
epam
Ajax
Part 2
UA Resource Development Unit
2020
epam
CONFIDENTIAL

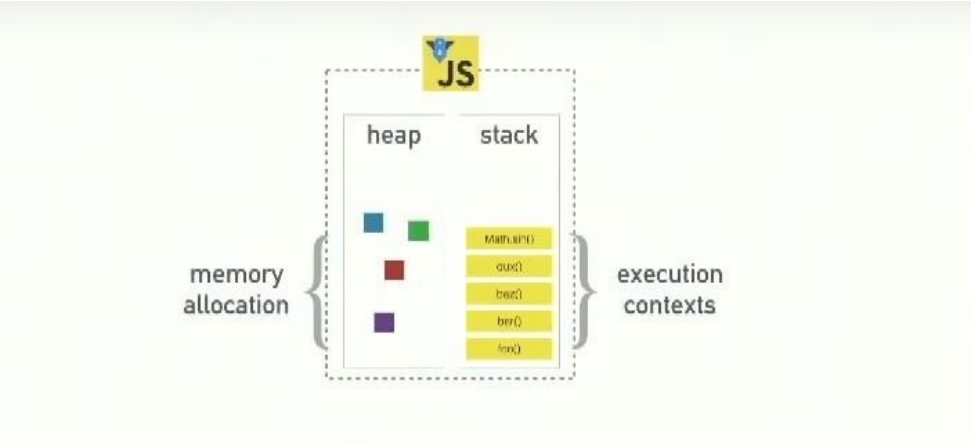
AGENDA

1. Event loop
2. Promise
3. Cross-domain restrictions
4. Comet
5. WebSocket

EVENT LOOP

JS What are you?

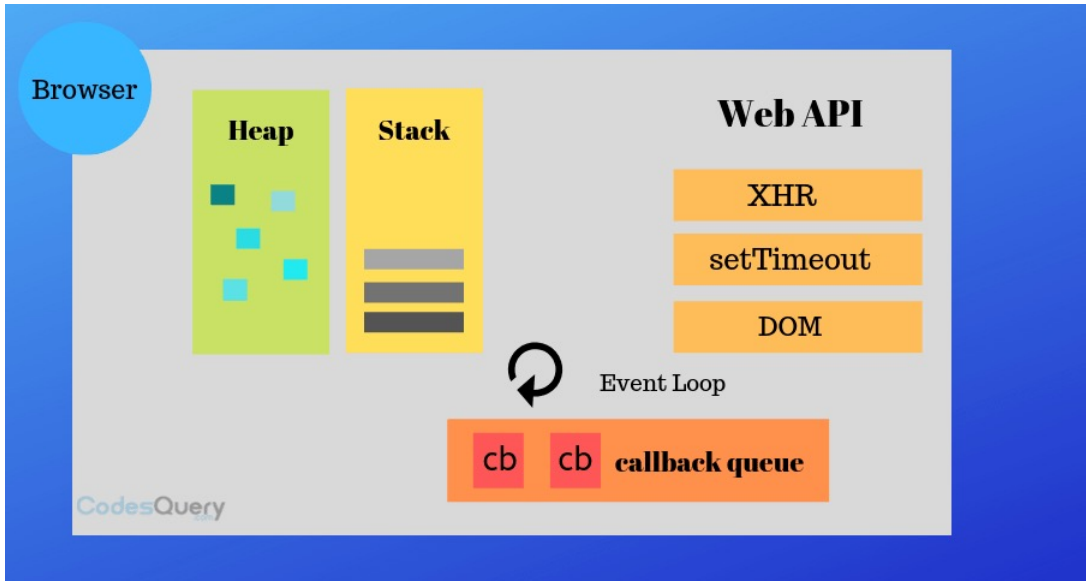
- Single-threaded
- Non-blocking
- Asynchronous
- Concurrent



Event Loop

Javascript runtime is single threaded which means that it can execute one piece of code at a time. In order to understand the concurrency model and the event loop in Javascript we have to first get around with some common terms that are associated with it. First let's learn what is a call stack.

A call stack is a simple data structure that records where in the code we are currently. So if we step into a function that is a function invocation it is pushed to the call stack and when we return from a function it is popped out of the stack.



Event Loop

In the simple term, Event Loop in javascript is a constantly running process that checks if the call stack is empty. If call stack is empty then it looks into the event queue. If the event queue has something that is waiting then it is moved to the call stacks and if not then nothing happens.

In the above paragraph, we used terms like call stacks, event queue. There are few other terms besides these which are heap, web APIs which is used to execute a javascript code.

Let's understand these terms in details then we will have a clear idea about event loop in javascript.

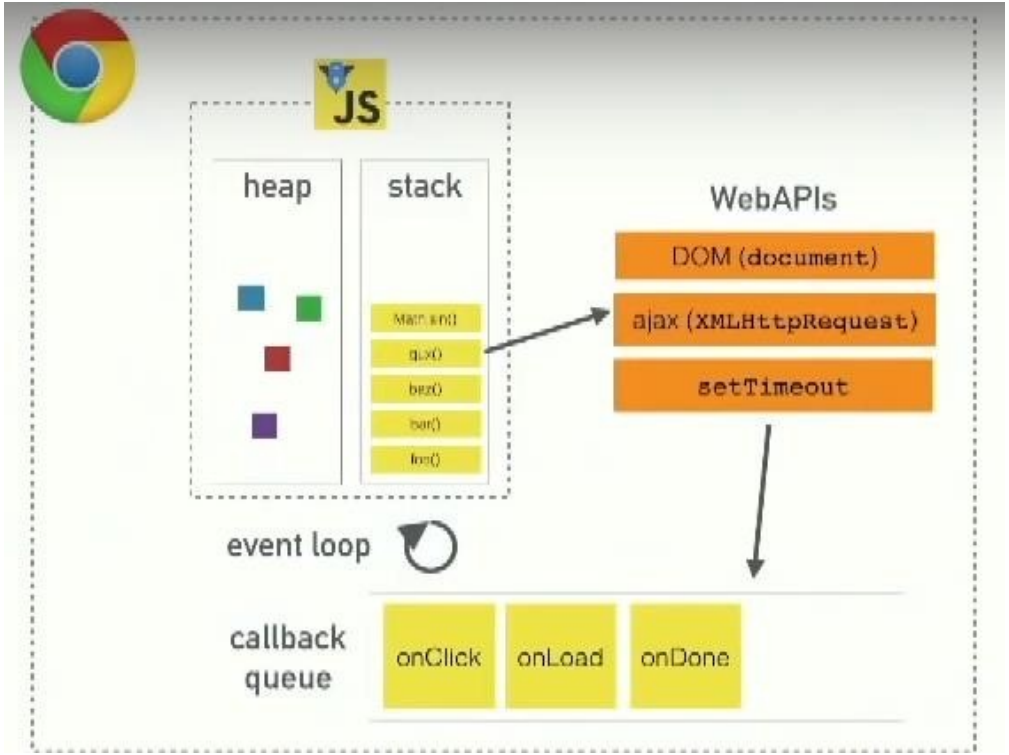
Heap

Heap in the javascript is a place where objects stores when we define them in our code.

Stacks

A Stack is a data structure which works in Last In First Out(LIFO) manners. The stack holds all the javascript function call. On each call, it pushed on the top of the stack.

stack


JS
heap
stack
WebAPIs
DOM (document)
ajax (XMLHttpRequest)
setTimeout
event loop
callback queue
onClick
onLoad
onDone

Event Loop

Web API

Browsers have defined APIs which developers can be used to make the complex process which knows as web API. You can find the whole list of the web API from here.

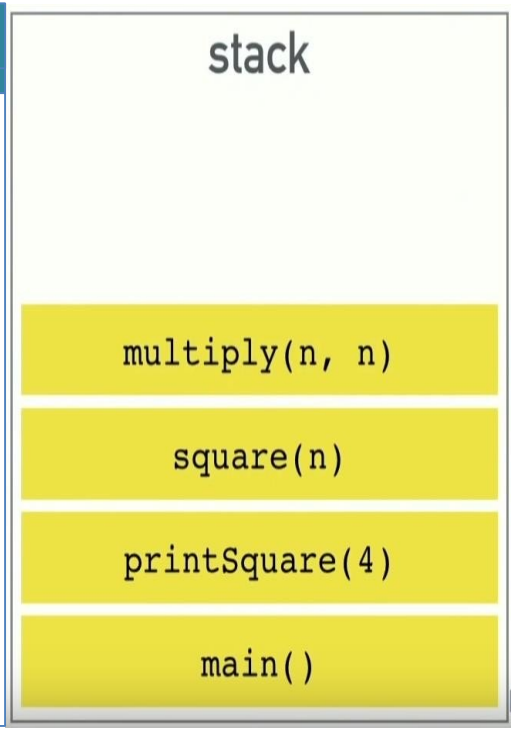
Callback Queue

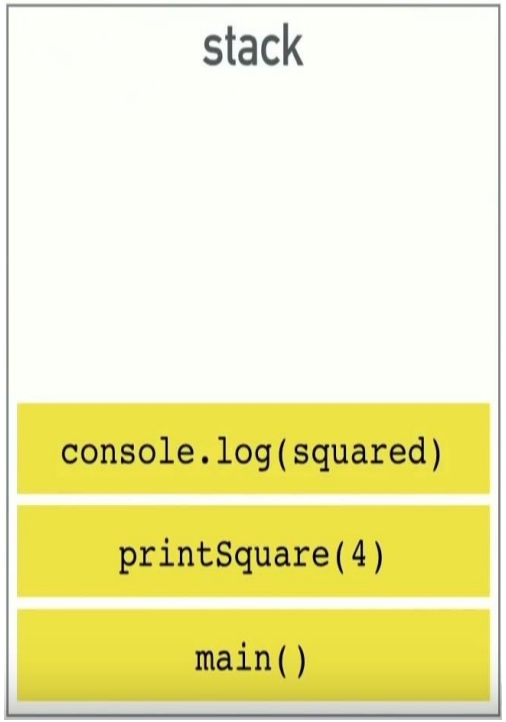
When a process such as XHR, setTimeout, setInterval, promises to finish its job then they pushed into the callback queue. The callback queue is triggered by the event loop process after our stack is empty. When our stacks have no function call then a process is popped out from the callback queue and pushed it to stack.

Event Loop

EXAMPLE

```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```





Event Loop & setTimeout

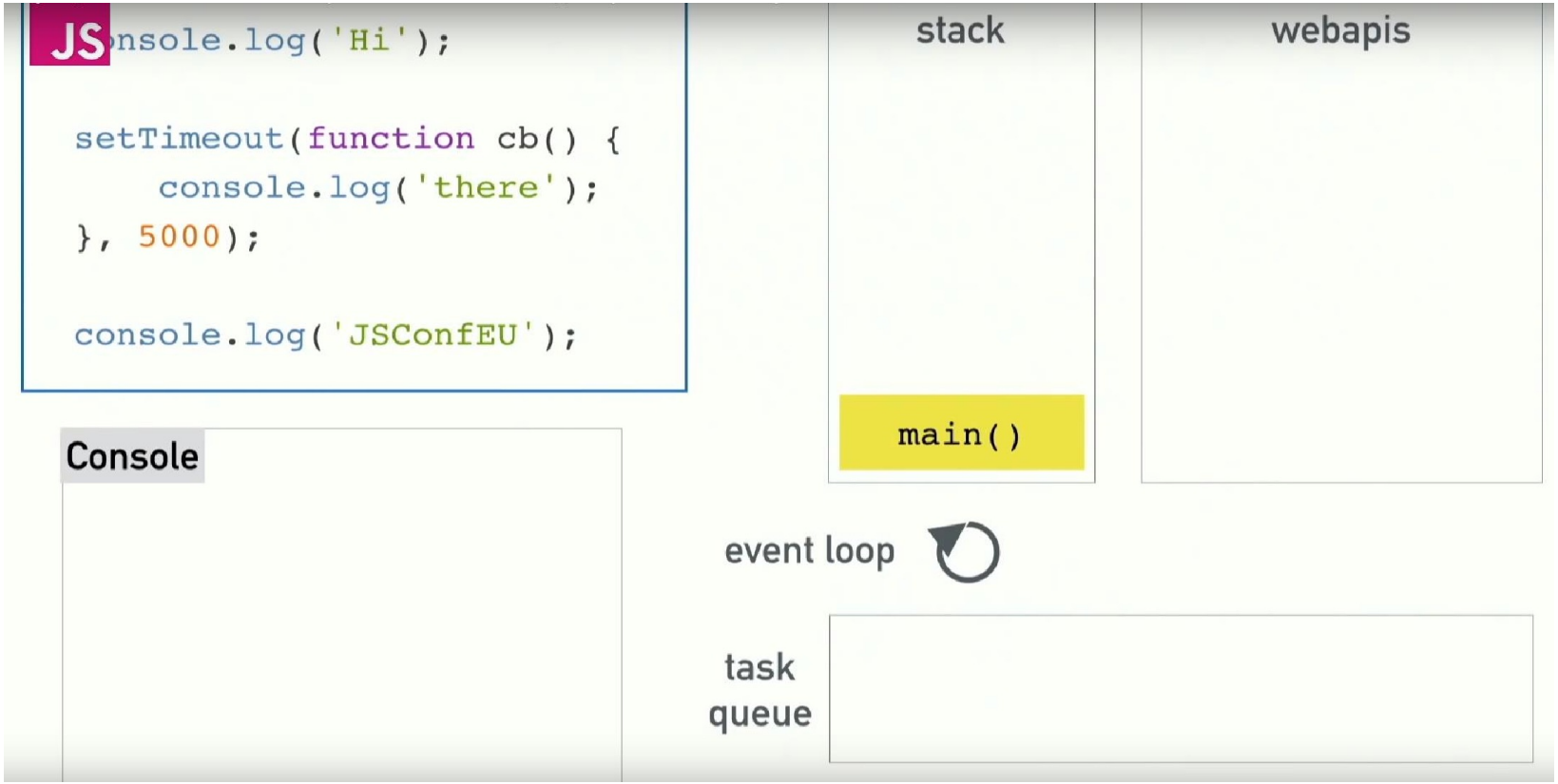
EXAMPLE

```
console.log(1);

setTimeout(function(
) {
    console.log(2);
}, 1000);

console.log(3);
```

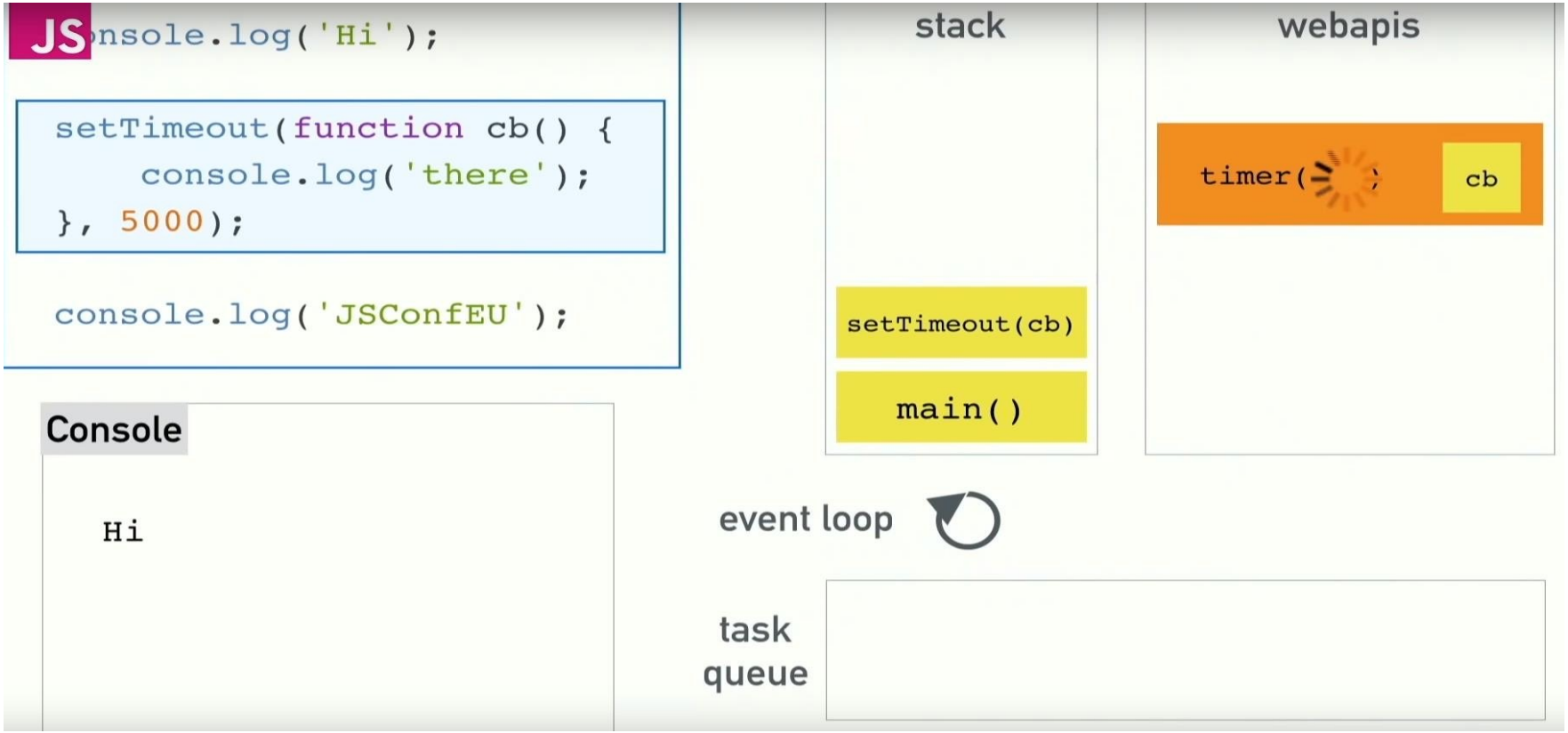
EXAMPLE

```
console.log(1);

setTimeout(function() {
    console.log(2);
}, 0);

console.log(3);
```

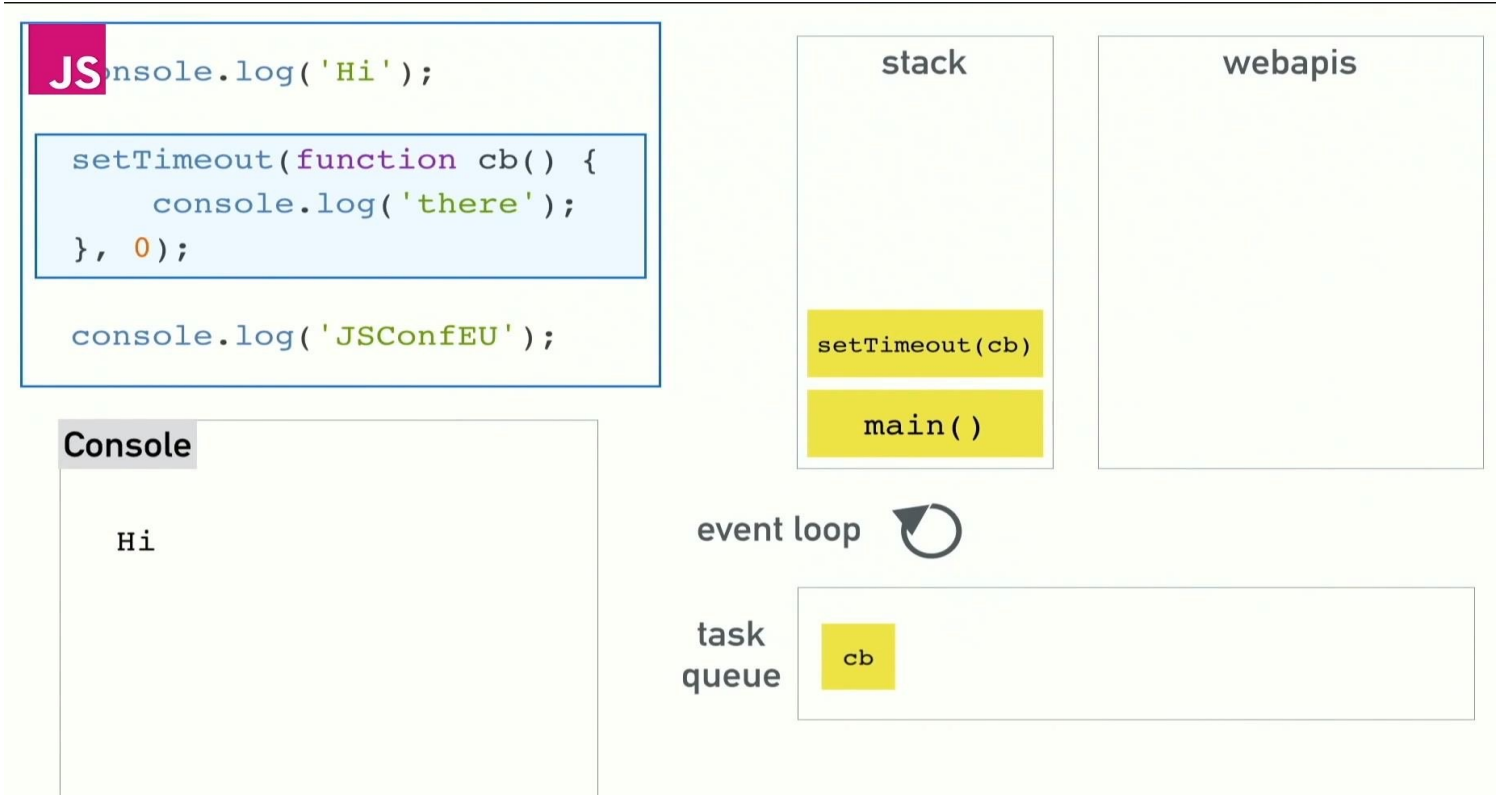
Concurrency & the Event Loop



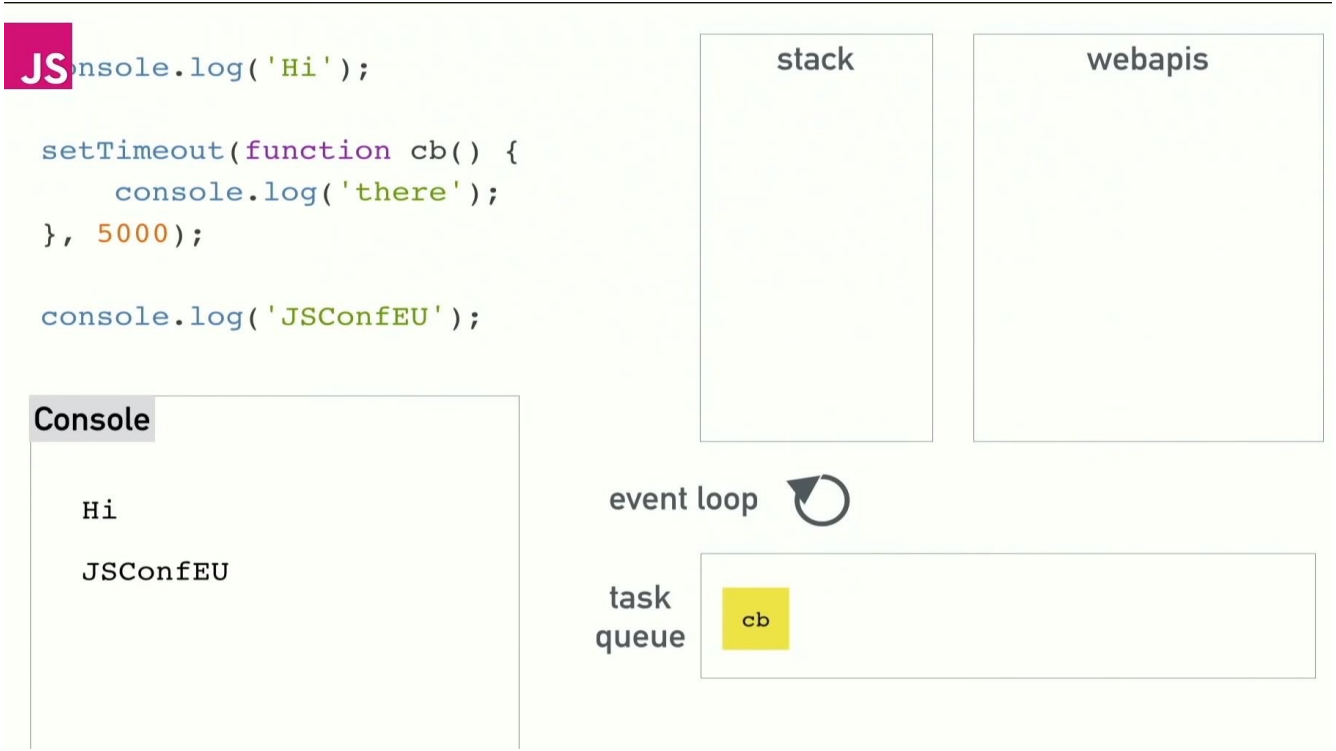
Concurrency & the Event Loop



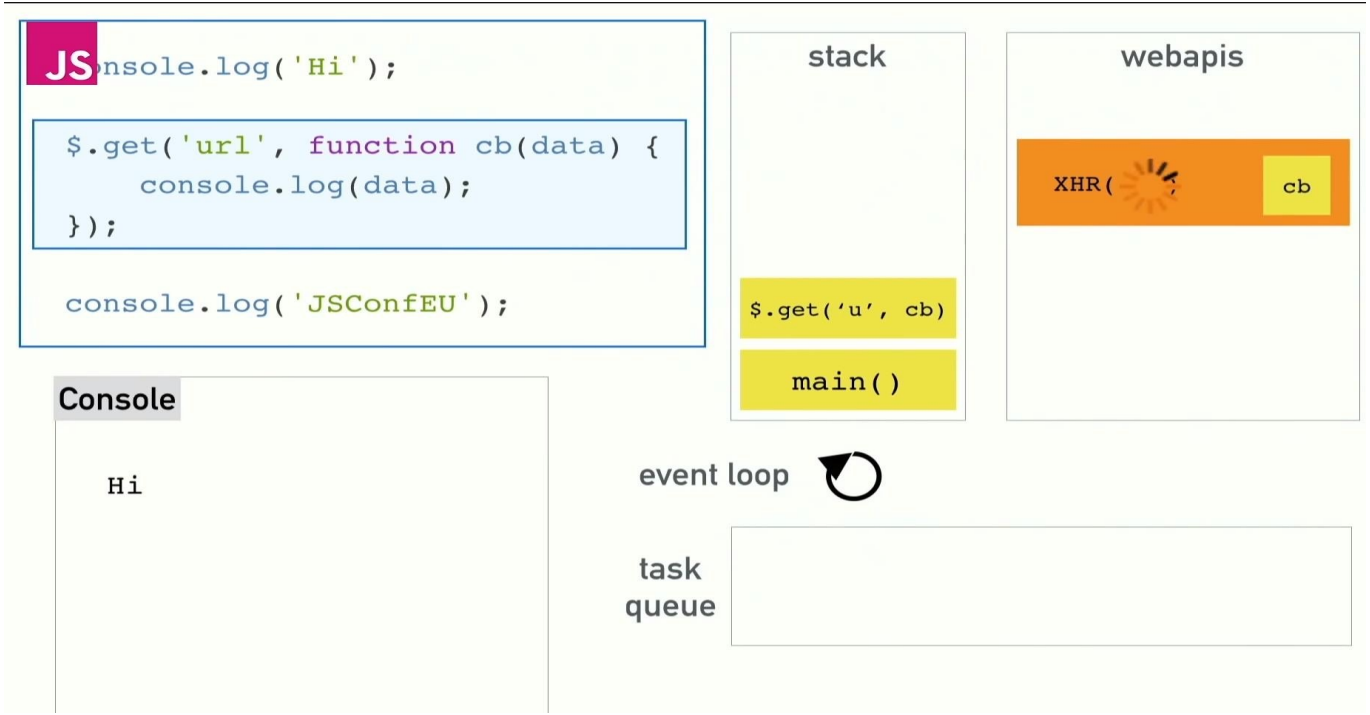
Concurrency & the Event Loop



Concurrency & the Event Loop



Concurrency & the Event Loop



setTimeout & setInterval

EXAMPLE

```
function sayHi() {
    console.log('Hi');
}

setTimeout(sayHi, 1000);
```

```
function sayHi(phrase, who) {
    console.log(`${phrase}, ${who}!`);
}

setTimeout(sayHi, 2000, "Привет", "Вася1");
```

setTimeout & setInterval

EXAMPLE

```
function sayHi(phrase, who) {
    console.log( `${phrase}, ${who}!`);
}

var timerId = setTimeout(sayHi, 500, "Привет", "Вася5");

console.log(timerId);
clearTimeout(timerId);
```

setTimeout & setInterval

EXAMPLE

```
setTimeout(function tick() {

    console.log( "тик" );
    setTimeout(tick, 500);

}, 500);
```

setTimeout & setInterval

EXAMPLE

```
console.log(1);

setTimeout(function() {
    console.log(2);
}, 1000);

console.log(3);
```

EXAMPLE

```
var i;

for (i = 0; i < 10; i++) {

    setTimeout(function() {
        console.log(i);
    }, 100);

}
```

Asynchronous operations

Asynchronous operations

According to Wikipedia: *Asynchrony in computer programming refers to the occurrence of events independently of the main program flow and ways to deal with such events.*

In programming languages like e.g Java or C# the “main program flow” happens on the main thread or process and “the occurrence of events independently of the main program flow” is the spawning of new threads or processes that runs code in parallel to the “main program flow”.

This is not the case with JavaScript. That is because a JavaScript program is single threaded and all code is executed in a sequence, not in parallel. In JavaScript this is handled by using what is called an *“asynchronous non-blocking I/O model”*. What that means is that while the execution of JavaScript is blocking, I/O operations are not. I/O operations can be fetching data over the internet with Ajax or over WebSocket connections, querying data from a database such as MongoDB or accessing the filesystem with the NodeJs “fs” module. All these kind of operations are done in parallel to the execution of your code and it is not JavaScript that does these operations; to put it simply, the underlying engine does it.

Callbacks

For JavaScript to know when an asynchronous operation has a result (a result being either returned data or an error that occurred during the operation), it points to a function that will be executed once that result is ready. This function is what we call a “callback function”. Meanwhile, JavaScript continues its normal execution of code. This is why frameworks that does external calls of different kinds have APIs where you provide callback functions to be executed later on.

Registering event listeners in a browser with “addEventListener” is example of common API that uses callbacks.

Here is an example of fetching data from an URL using a module called “request”:

```
let result;
request('http://www.somepage.com', function (error, response, body)
{
    if(error){
        // Handle error.
    }
    else {
        result = body;
    }
});
console.log(result);
```

Callbacks

So if we want to do a second request based on the result of a first one we have to do it inside the callback function of the first request because that is where the result will be available:

```
request('http://www.somepage.com', function (firstError,
firstResponse, firstBody) {
    if(firstError){
        // Handle error.
    }
    else {
        request(`http://www.somepage.com/${firstBody.someValue}`,
function (secondError, secondResponse, secondBody) {
            if(secondError){
                // Handle error.
            }
            else {
                // Use secondBody for something
            }
        });
    }
});
```

Callbacks are a good way to declare what will happen once an I/O operation has a result, but what if you want to use that data in order to make another request? You can only handle the result of the request (if we use the example above) within the callback function provided.

PROMISE

Promises

A promise is an object that wraps an asynchronous operation and notifies when it's done. This sounds exactly like callbacks, but the important differences are in the usage of Promises. Instead of providing a callback, a promise has its own methods which you call to tell the promise what will happen when it is successful or when it fails. The methods a promise provides are "then(…)" for when a successful result is available and "catch(…)" for when something went wrong.

There are lots of frameworks for creating and dealing with promises in JavaScript, but all of the examples below assumes that we are using native JavaScript promises as introduced in ES6.

Using a promise this way looks like this:

```
someAsyncOperation(someParams)
.then(function(result){
    // Do something with the result
})
.catch(function(error){
    // Handle error
});
```

Promises

The true power of promises is shown when you have several asynchronous operations that depend on each other, just like in the example above under “Callback Hell”. So let’s revisit the case where we have a request that depends on the result of another request. This time we are going to use a module called “axios” that is similiar to “request” but it uses promises instead of callbacks. This is also to point out that callbacks and promises are not interchangeable.

Using axios, the code would instead look like this:

```
const axios = require(‘axios’);


axios.get(‘http://www.somepage.com')
.then(function (response) { // Reponse being the result of the first
request
    // Returns another promise to the next .then(..) in the chain
    return
axios.get(`http://www.somepage.com/${response.someValue}`);
})
.then(function response { // Reponse being the result of the second
request
    // Handle response
})
.catch(function (error) {
    // Handle error.
});
```

Promise

new Promise(executor);

new Promise(function(resolve, reject) { ... });

EXAMPLE

```
var promise = new Promise(function(resolve, reject)
{
  // This function will be called automatically
});

// onFulfilled will work if successful
promise.then(onFulfilled)

// onRejected will work if an error occurs
promise.then(null, onRejected) // catch
```

Promise

In order to put the handler only on an error,
Instead
.then(null, onRejected)

.catch(onRejected)

Promise

EXAMPLE

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject("error");
        resolve("result");
        reject("error");
    }, 1000);
});

promise
    .then(
        result => {
    console.log("Fulfilled: " + result);
    },
    error => {
        console.log("Rejected: " + error);
    }
);
```

Promise

EXAMPLE

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject("My error");
        resolve("My result");
        reject("My error");
    }, 1000);
});

promise
    .then(
            result => {
        console.log("Fulfilled: " + result);
    })
    .catch(
        error => {
        console.log("Rejected: " + error);
    });
```

Promise

EXAMPLE

```
var promise1 = Promise.resolve(1);

  va promise2 =                                    {
  r  promise1.then(function(value)
     return Promise.resolve(value * 3);
  })
  ;
var promise3 =                                     {
     promise1.then(function(value)
     return Promise.resolve(value + 1);
  });
  promise3.then((value) => console.log(value));
```

Promise

EXAMPLE

```
var promise1 = Promise.resolve(1);

var promise2 = promise1.then(function(value {
        return )
    }).then(function(value) {
        return Promise.resolve(value * 3);
        value = value + 1;
    }).then((value) => console.log(value));
```

Promise

EXAMPLE

```
var promise = new Promise(function(resolve, reject) {
    setTimeout(resolve, 100);
    setTimeout(reject, 50);
});
promise.then(function() {
        console.log('ok');
    },
    function() {
        console.log('ko');
    });
```

Async/Await

Async/Await is a language feature that is a part of the ES8 standard. It was implemented in version 7.6 of NodeJs. If you are new to JavaScript this concept might be a bit hard to wrap your head around, but I would advise that you still give it a try. You don't have to use it if you don't want to. You will be fine with just using Promises.

Async/Await is the next step in the evolution of handling asynchronous operations in JavaScript. It gives you two new keywords to use in your code: "async" and "await". Async is for declaring that a function will handle asynchronous operations and await is used to declare that we want to "await" the result of an asynchronous operation inside a function that has the async keyword.

A basic example of using async/await looks like this:

```
async function getSomeAsyncData(value){
    const result = await fetchTheData(someUrl, value);
    return result;
}
```

Error handling with async/await

Inside the scope of an async function you can use try/catch for error handling and even though you await an asynchronous operation, any errors will end up in that catch block:

```
async function getSomeData(value){
    try {
        const result = await fetchTheData(value);
        return result;
    }
    catch(error){
        // Handle error
    }
}
```

CROSS-DOMAIN RESTRICTIONS

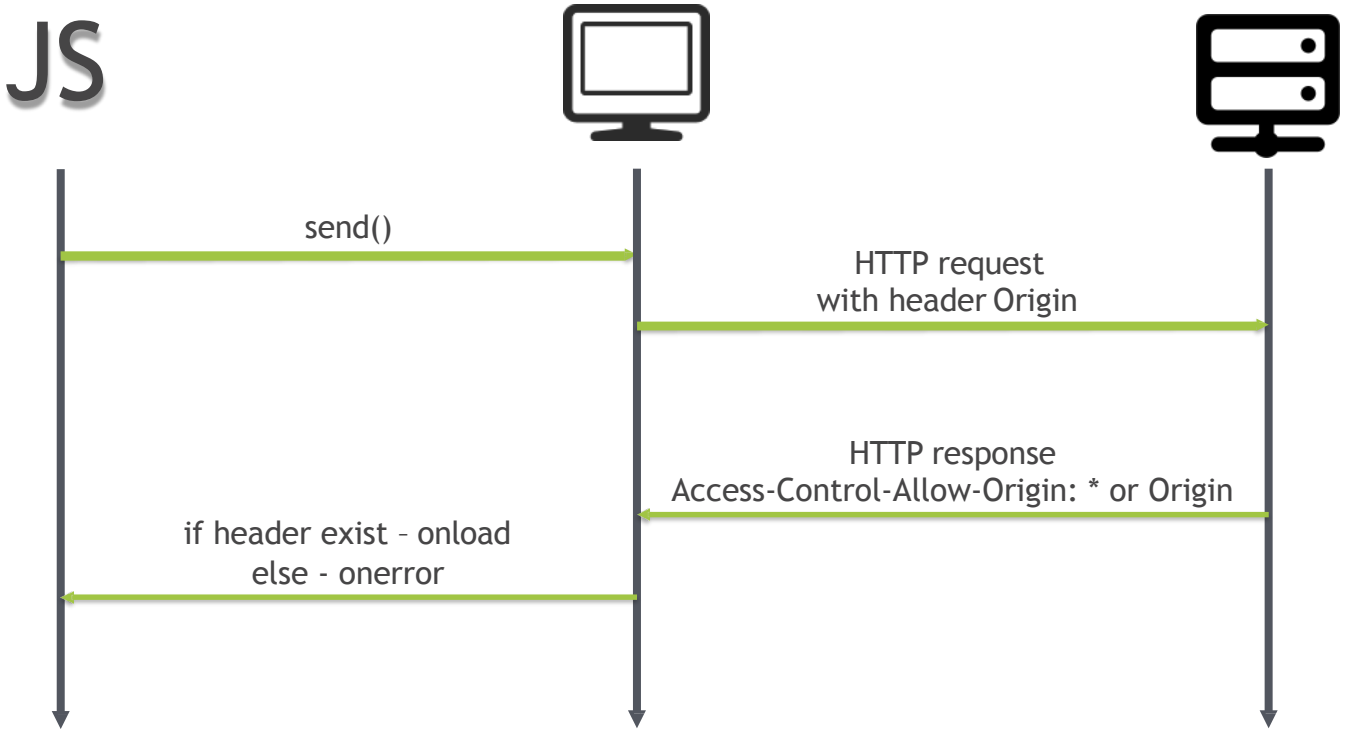
POSSIBLE SOLUTIONS

Server Proxy

JSONP

CORS

SIMPLE CORS REQUEST



COMET

COMET TECHOLOGIES

Implementing

request a server in some interval

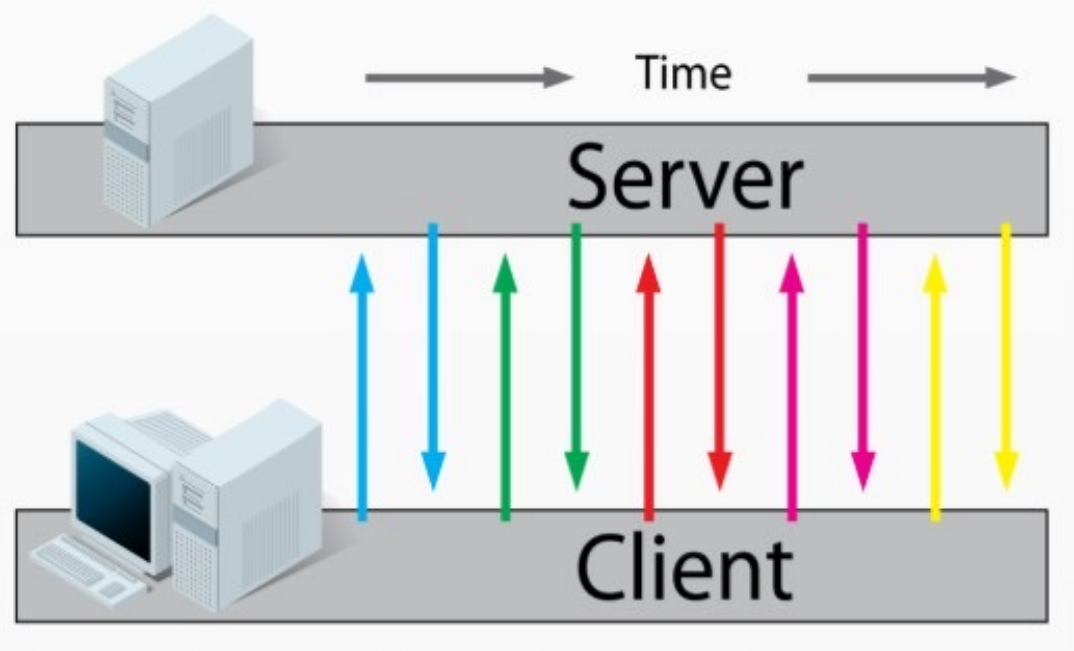
the server may hold a single request as long as it want

full-duplex communication channels over a single TCP connection

POLLING

EXAMPLE

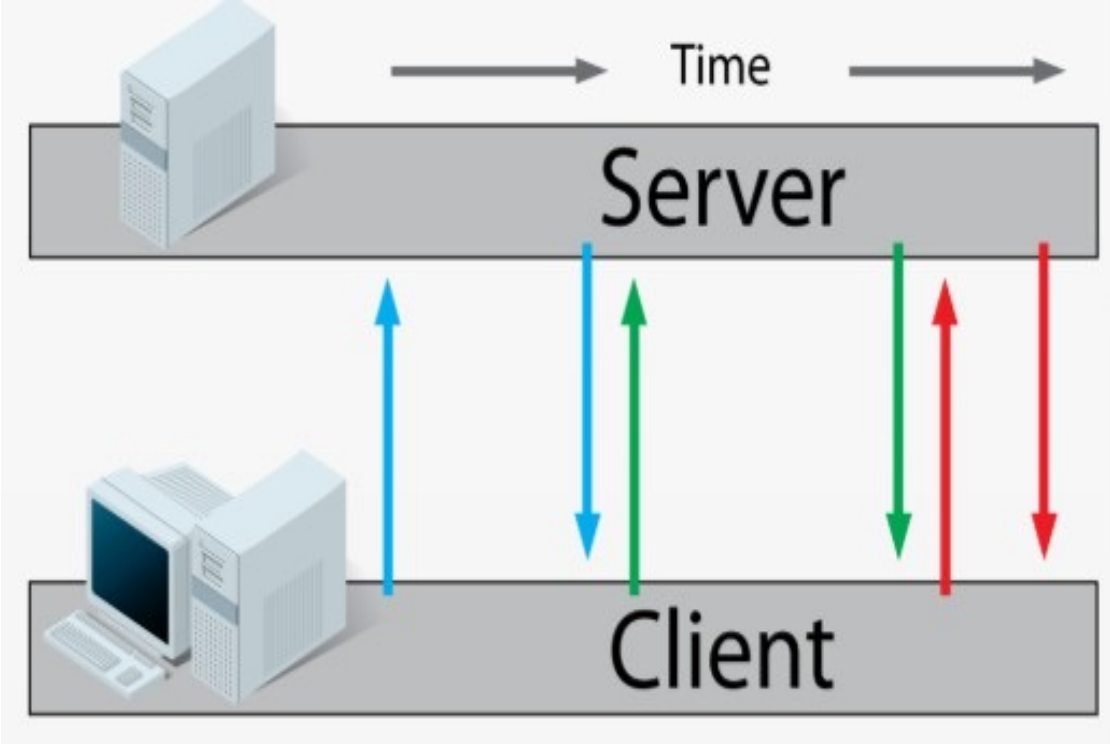
```
setInterval(function() {
    var xhr = new XMLHttpRequest();

    xhr.onload = function() {
        onMessage(this.responseText);
    }
    xhr.onerror = function() {
        onError(this);
    }

    xhr.open('GET', url, true);
    xhr.send();
}, 2000);
```



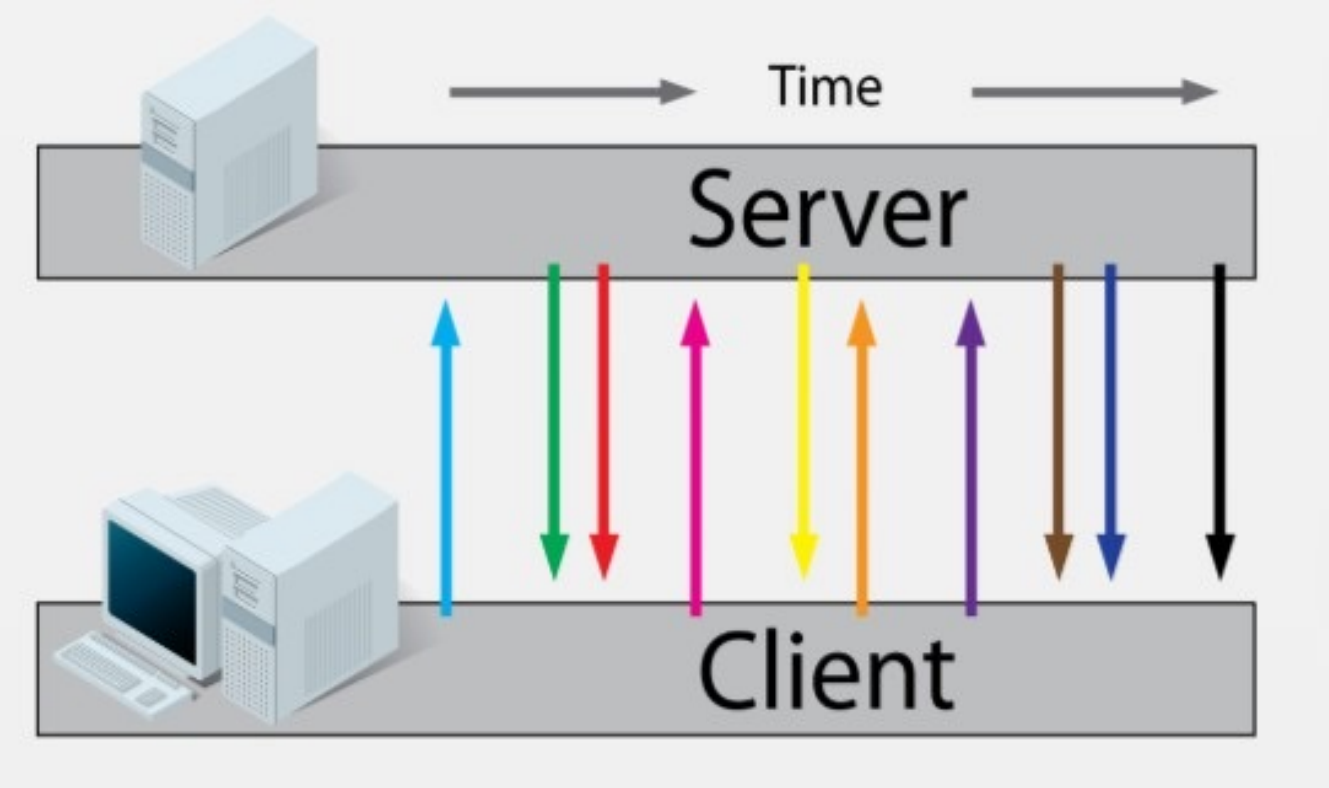
LONG-POLLING

EXAMPLE

```
function subscribe(url) {
    var xhr = new XMLHttpRequest();

    xhr.onload = function() {
        onMessage(this.responseText);
}
    xhr.onerror = function() {
        onError(this);
}
    xhr.onloadend = function() {
        subscribe(url);
    }

    xhr.open('GET', url, true);
    xhr.send();
}
```



WEBSOCKETS



WEBSOCKETS

EXAMPLE

```
var socket = new WebSocket("ws://javascript.ru/ws");
socket.onopen = function() {
    console.log('Connected');
};
socket.onclose = function(event) {
    if (event.wasClean) {
        console.log('The connection is closed cleanly');
    } else {
        console.log('Connection failure');
    }
    console.log('Code: ' + event.code + ' reason : ' +
    event.reason);
};
socket.onmessage = function(message) {
    console.log("Message received " + message.data);
};
socket.onerror = function(error) {
    console.log('Error!',  error.message);
    console.dir(error);
};
```

FE Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

serhii_shcherbak@epam.com

With the note [FE Online UA Training Course Feedback]

Thank you.

Q&A

UA Frontend Online LAB