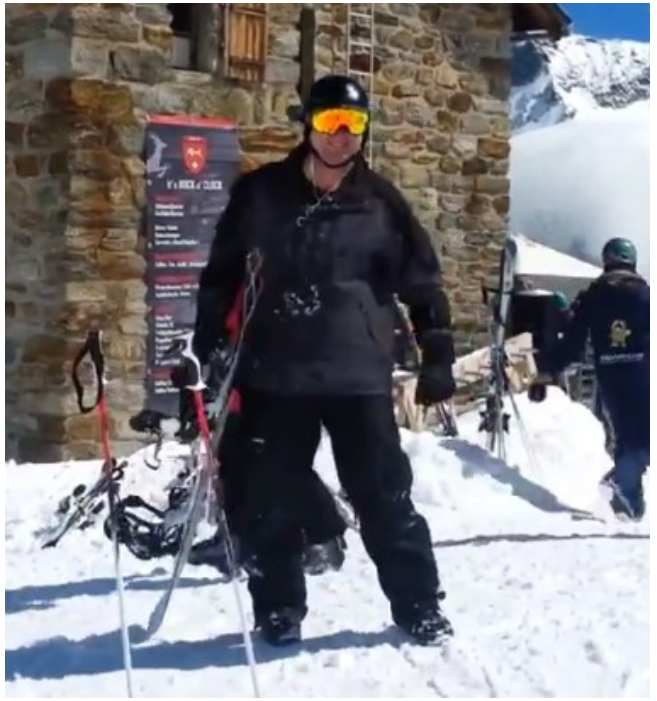
Asynchronous JS

Haim Michael

February 24th, 2020

Haim Michael Introduction

- Snowboarding. Learning. Coding. Teaching. More than 18 years of Practical Experience.



Haim Michael Introduction

- Professional Certifications

  Zend Certified Engineer in PHP

  Certified Java Professional

  Certified Java EE Web Component Developer

  OMG Certified UML Professional

- MBA (cum laude) from Tel-Aviv University

  Information Systems Management

Introduction

Introduction

- JavaScript is a single thread programming language that provides us with asynchronous mechanism and multi threading using web workers.

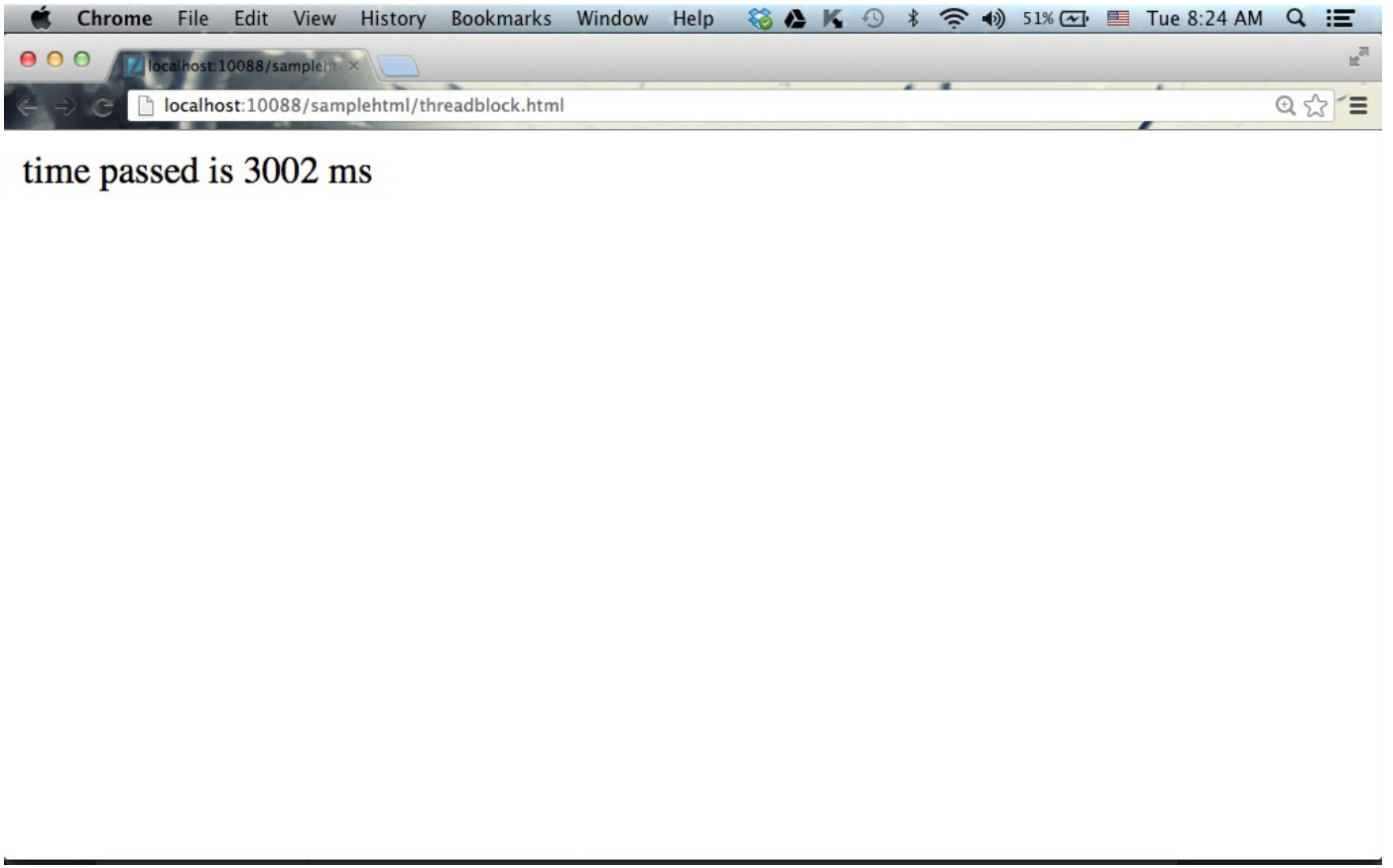
The Events Queue

- There is a queue of tasks the one and only thread needs to complete.

The Events Queue First Demo

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript Thread Block</title>
</head>
<body>
    <script type="text/javascript">
        var startTime = new Date();
        setTimeout(function()
        {
            document.write("time passed is " +
                (new Date() - startTime)+" ms");
        }, 500);
        while (new Date() - startTime < 3000) {};
    </script>
</body>
</html>
```

The Events Queue First Demo



The Events Queue

- The one single thread and its events queue is the reason for the unresponsiveness many web pages tend to show.

Asynchronous Functions

- When calling an asynchronous function in JavaScript we expect it to return immediately and we expect it to call the callback function we usually pass over (when it completes its operation).
- When calling a synchronous function (the common case) we expect it to return only when it completes its operation.

Asynchronous Functions

- In some cases we might encounter functions that might behave either in a synchronous or in an asynchronous way. One example is the $ function in jQuery. When passing it another function that other function will be invoked when the DOM loading completes. If the DOM was already loaded it will be invoked immediately.

JavaScript Loading

- When using the simple script element tag as in the following code sample the script will be loaded synchronously.

```
<script type="text/javascript" src=”lib.js”></script>
```

- Loading too many scripts using this script element in the `<head>` part of the page might delay the rendering.
- Loading the scripts by putting the script elements in the end of the `<body>` part might get us a static page with nonworking controls.

JavaScript Loading

- When the script is called from code that belongs to the body part of our page or when the script is responsible for the look of our page then we better load it in the `<head>` element.

JavaScript Loading

- We can load our script programmatically either by using a JavaScript library that was developed especially for that purpose or by writing our own code.

```
var head = document.getElementsByTagName('head')[0];
var script = document.createElement('script');
script.src = 'mylib.js';
head.appendChild(script);
```

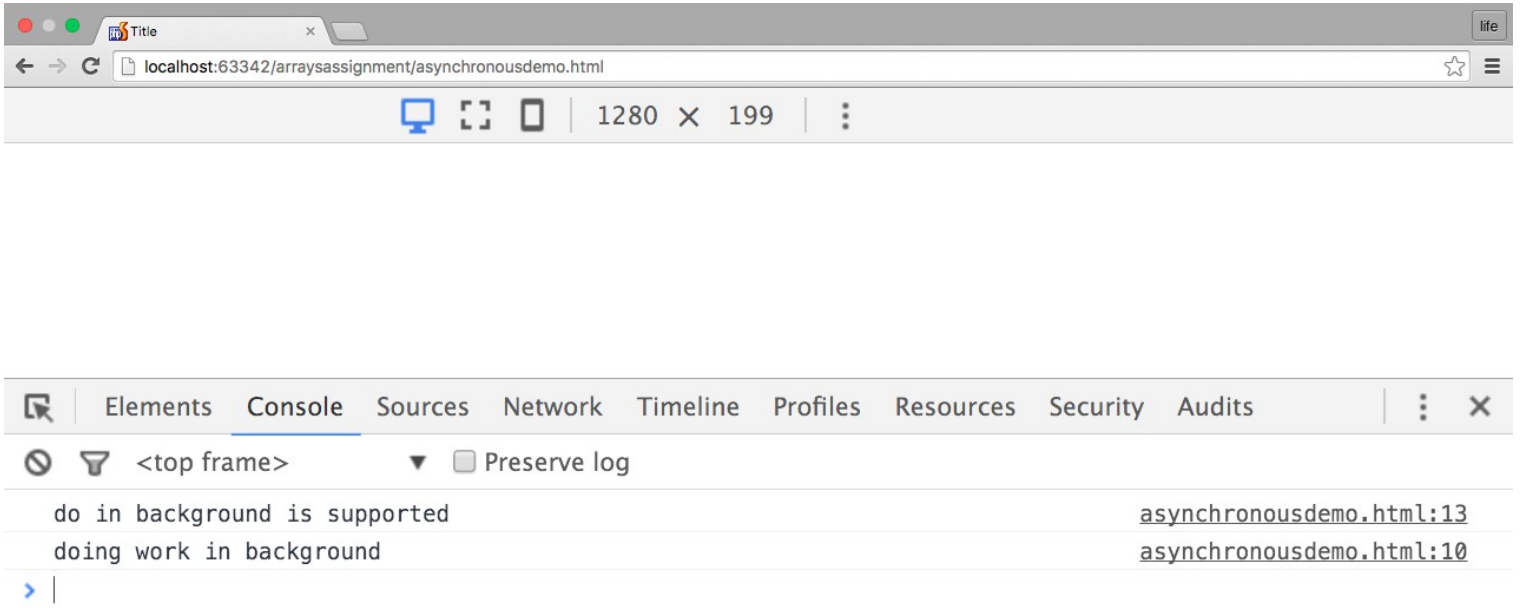
The `requestIdleCallback` Function

- Calling this function we should pass over the function we want to be invoked in the background in those moments when the one and only thread is free (not busy with other tasks).

The `requestIdleCallback` Function

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <script>
        function doInBackground() {
            console.log("doing work in background");
        }
        if (window.requestIdleCallback) {
            console.log("do in background is supported");
            requestIdleCallback(doInBackground);
        }
        else {
            setTimeout(doInBackground, 3);
        }
    </script>
</body>
</html>
```

The `requestIdleCallback` Function



Promises

Introduction

- ECMAScript 2015 provides us with the possibility to use promises in our code.
- Promises allow us to to write code that executes asynchronously. Promises allow us to write code that will be executed at a later stage and if succeeded or failed a notification will be generated accordingly.
- We can chain promises together based on success or failure in a simpler easy to understand way.

Single Thread

- JavaScript has a single thread that handles a queue of jobs. Whenever a new piece of code is ready to be executed it will be added to the queue.
- When the JavaScript engine completes the execution of specific job the event loop picks the next job in the queue and executes it.

The Event Loop

- The event loop is a separated thread inside the JavaScript engine.
- The event loop monitors the code execution and manages the jobs queue. The jobs on that queue are executed according to their order from the first job to the last one.

Events

- When a user clicks a button or presses a key on the keyboard an event is triggered.
- The triggered event can be hooked with the code we want to be executed whenever that event is triggered. The code we hooked with the event will be added as a new job to the queue.

Events

- The event handler code doesn't execute until the event fires, and when it does execute, it is executed as a separated job that will be added to the queue and waits for its execution.

```
var button = document.getElementById("mybt");
button.onclick = function(event) {
    console.log("click!");
};
```

The Callback Pattern

- The callback function is passed over as an argument. The callback pattern makes it simple to chain multiple calls together.

The Callback Pattern

```
func1("temp.txt", function(err, contents) {
    if (err) {
        console.log(“error...”)
    }

    func2("another.txt", function(err) {
        if (err) {
            console.log(“error...”)
        }

    });
});
```

Getting Location Sample

```
...
navigator.geolocation.getCurrentPosition(myfunc,myerrorfunc);
...
```

The Callback Hell Pattern

- When using the callback pattern and nesting too many callbacks it can easily result in code that is hard to understand and difficult to debug.

The Callback Hell Pattern

```
func1(function(err, result) {
    if (err) {
        console.log("error...");
    }
    func2(function(err, result) {
        if (err) {
            console.log("error...");
        }
        func3(function(err, result) {
            if (err) {
                console.log("error...");
            }
            func4(function(err, result) {
                if (err) {
                    console.log("error...");
                }
                func5(result);
            });
        });
    });
});
```

Problems of Higher Complexity

- When coping with problems of an higher complexity, such as having two asynchronous operations running in parallel and having another function we want to execute when the first two completes.

Promise Basics

- Promise is an object that represents the result of an asynchronous operation. Through the promise object it will be possible to get the result of the asynchronous operation when completed.

Promise Lifecycle

- Each and every promise goes through a short lifecycle. It starts in the pending state (the asynchronous operation has not yet completed).
- Once the asynchronous operation completes, the promise is considered settled and enters one of two possible states. Fulfilled (the asynchronous operation has completed successfully) or Rejected (the asynchronous operation did not complete successfully, due to some error or another cause).

Promise Lifecycle

- We can’t determine in which state the promise is in programmatically.
- We can take a specific action when a promise changes state by using the then() method.
- Each and every promise object has state property that is set to "pending", "fulfilled", or "rejected" in order to reflect the promise’s state.

The `then` Method

- The `then()` method is available on every promise object. It takes two arguments. The first argument is a function to call when the promise is fulfilled. The second argument is a function to call when the promise is rejected.
- Both the fulfillment function and the rejection function are passed over additional data related to the fulfillment or the rejection (accordingly).

The `Promise` Constructor

- We can create new promises by calling the `Promise` function constructor. The `Promise` function receives a single argument, which is the function that contains the code to be executed when the promise is added to the queue.
- The function we pass over to `Promise` should be with two parameters that receive two functions as arguments. The `resolve()` and the `reject()`.

The `Promise` Constructor

- The `resolve()` function should be called when the executor has finished successfully in order to signal that the promise is ready to be resolved.
- The `reject()` function should be called when the executor function fails and we want to indicate about it.

The `Promise` Constructor

```
var promise = new Promise(function(resolve, reject) {
    document.write("<br/>promise was created!");
    resolve();
});

promise.then(function() {
    document.write("<br/>the promise's executor completed... then one!");
}).then(function() {
    document.write("<br/>the promise's executor completed... then two!");
}).then(function() {
    document.write("<br/>the promise's executor completed... then three!");
});

document.write("<br/>simple output!");
```

The `Promise` Constructor

promise was created!
simple output!
the promise's executor completed... then one!
the promise's executor completed... then two!
the promise's executor completed... then three!

The `catch` Method

- The `catch()` function is called when the executor (represented by the promise object) fails. We should indicate about that failure by calling the `reject()` function.

The `catch` Method

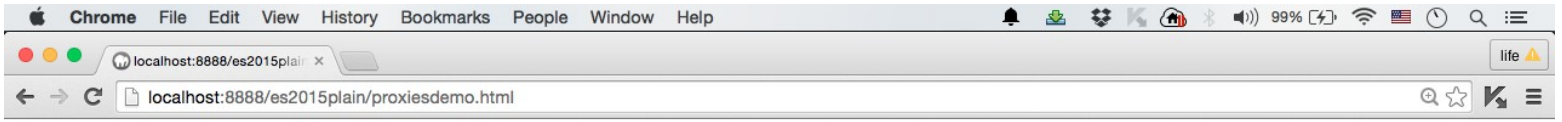
```
var promise = new Promise(function(resolve, reject) {
    document.write("<br/>promise was created!");
    //resolve();
    reject();
});

promise.then(function() {
    document.write("<br/>the promise's executor completed... then one!");
}).then(function() {
    document.write("<br/>the promise's executor completed... then two!");
}).then(function() {
    document.write("<br/>the promise's executor completed... then three!");
}).catch(function() {
    document.write("<br/>inside catch");
});

document.write("<br/>simple output!");
```

Practical Code Sample

https://github.com/mdn/js-examples/blob/master/promises-test/index.html

Code Sample for Asynchronously Loading of Image using Ajax

The `catch` Method




```
promise was created!
simple output!
inside catch
```

The Fetch API

Introduction

- The Fetch API provides an interface for fetching resources, whether on the network or not.
- The new Fetch API provides us with more capabilities comparing with using the XmlHttpRequest object.

The Request and Response Objects

- The Fetch API provides us with a generic definition of Request and Response objects.

The fetch Method

- The Fetch API provides us with a generic definition of Request and Response objects.
- Calling this method we should pass over the URL for the resource we want to fetch.
- The fetch method receives one mandatory argument. The path to the resource we want to fetch.

The fetch Method

- The fetch method returns a `Promise` object that resolves to the `Response` object. We will get the `Response` object in any case. Whether the response was successful or not.

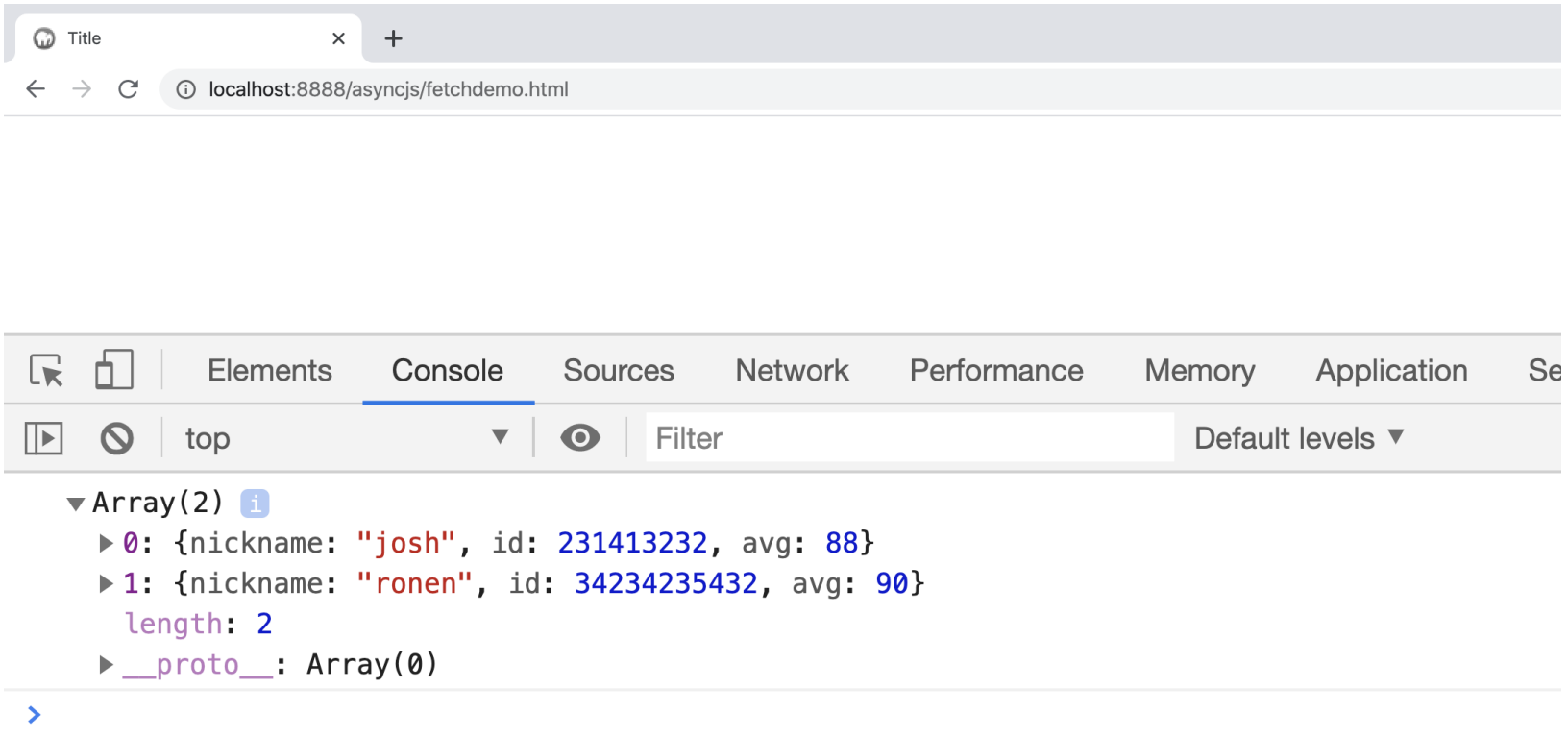
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Simple Code Sample for using The Fetch API

Simple Demo

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <script>
        fetch('students.json')
            .then((response) => {
                return response.json();
            })
            .then((data) => {
                console.log(data);
            });
    </script>
</body>
</html>
```

Simple Demo



The async Keyword

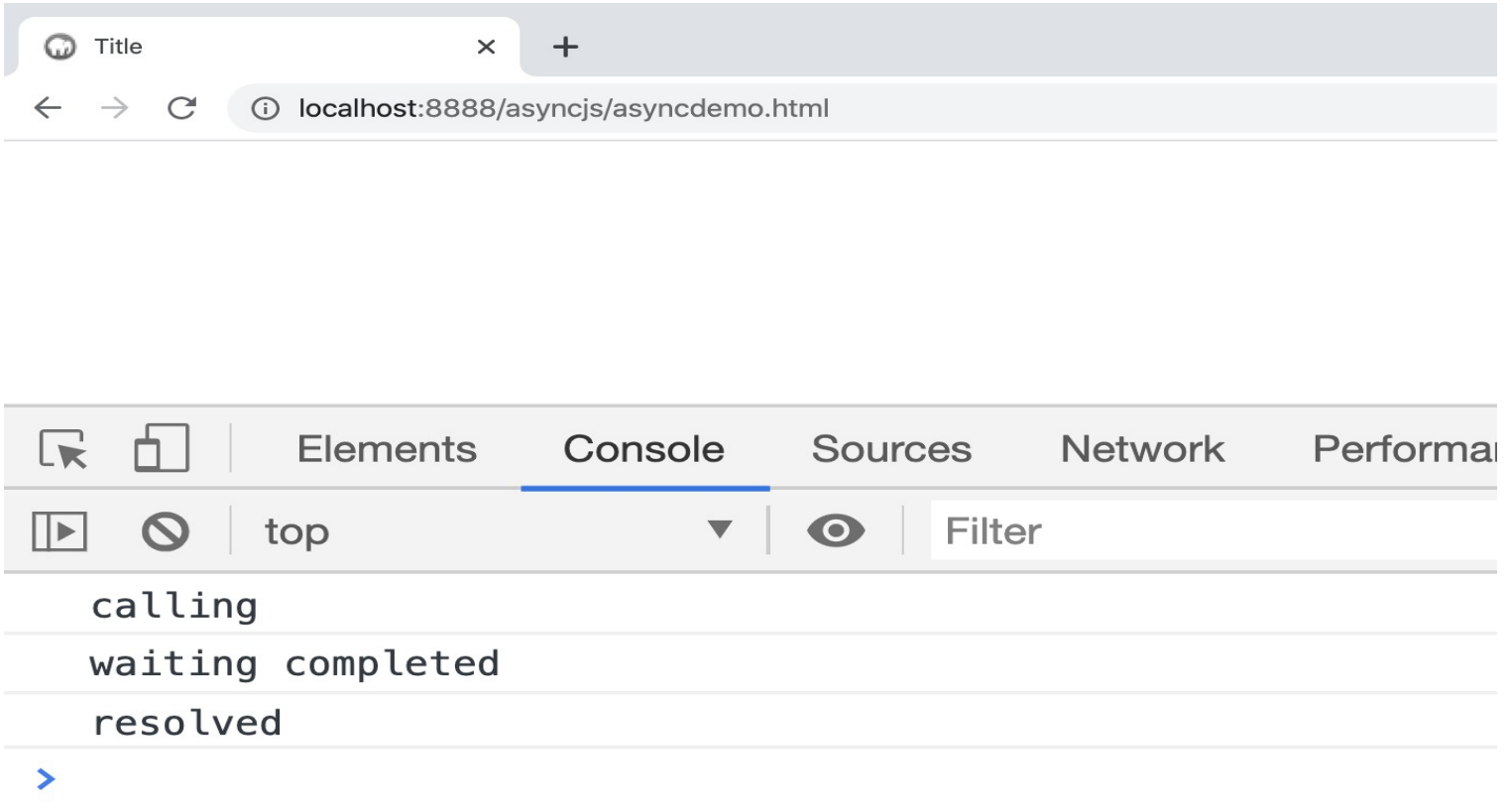
Introduction

- Functions we mark with the `async` keyword are asynchronous functions.
- Functions marked with the `async` keyword return an `AsyncFunction` object, which is implicitly a `Promise`.
- When calling a function that returns AsyncFunction object together with the `await` keyword we will get the AsyncFunction only after its operation was completed.

Code Sample

```
function functionWith3SecondsDelay() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('resolved');
        }, 3000);
    });
}

async function asyncFunction() {
    console.log('calling');
    const result = await functionWith3SecondsDelay();
    console.log("waiting completed");
    console.log(result);
    // expected output: 'resolved'
}

asyncFunction();
```

Code Sample



Questions & Answers

Thanks for Your Time!

Haim Michael
haim.michael@lifemichael.com
+972+3+3726013 ext:700

life michael