

Web 2.0

Lecture 1: Asynchronous I/O

doc. Ing. Tomáš Vitvar, Ph.D.

tomas@vitvar.com • @TomasVitvar • <http://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <http://vitvar.com/courses/w20>



Modified: Mon Feb 17 2020, 10:36:27
Humla v0.3

Overview

- **Asynchronous I/O Overview**
- Asynchronous I/O in JavaScript
- JavaScript Language Overview

Recall: Application Server

- Environment that runs an application logic
 - Client communicates with AS via an application protocol
 - Client – Browser, application protocol – HTTP
- Terminology
 - Application Server × Web Server × HTTP Server
 - AS is a modular environment; provides technology to realize enterprise systems
 - AS contains a Web server/HTTP server
 - We will deal with Web server only
- Two major models to realize communication
 - Blocking I/O (also called synchronous I/O)
 - Non-blocking I/O (also called asynchronous I/O)
- A technology we will work with
 - Node.js – runs server-side Javascript

Programming Models

- Concurrency
 - Multiple tasks have the ability to run in an overlapping manner
 - Concurrency does not imply parallelism!
- Multiprocessing
 - CPU-bounded tasks
 - Allows to process multiple processes on different CPUs
- Multithreading
 - I/O bound tasks
 - Multiple threads execute tasks
 - A process may contain multiple threads
 - It uses **preemptive multitasking**
 - OS decides how long a task should run (no tasks cooperation)
 - context switching
 - Threads can access shared memory; you need to control this

Asynchronous I/O

- Asynchronous I/O
 - *A style of concurrent programming; it is not a parallelism*
 - *Single-threaded, single process design*
 - *It uses **cooperative multitasking***
- Asynchronous processing of a task
 - *Tasks are running in so called **event loop***
 - *A task is able to "pause" when they wait for some result*
 - *A task let other tasks to run*
 - *Asynchronous code facilitates concurrent execution*
 - *It gives the "look and feel" of concurrent execution*

Overview

- Asynchronous I/O Overview
- **Asynchronous I/O in JavaScript**
- JavaScript Language Overview

Web 2.0 Application Architecture

Web Application

client-side technologies for presentation and user interactions



Web 2.0 Application



JavaScript

- Lightweight, interpreted, object-oriented language
- Client-side (browser) and server-side (node.js, AppsScript)
- Standard
 - Current stable release is ECMAScript 2017 (standard ECMA-262)
- Major characteristics
 - Function is an Object
 - passing functions as arguments to other functions
 - returning functions as values from other functions
 - assigning functions to variables
 - storing functions in data structures.
 - Anonymous functions
 - declared without any named identifier to refer to it
 - Arrow functions
 - Closures

Event Loop

Event Loop:

```
while (queue.waitForMessage()) {  
  queue.processNextMessage()  
}
```

- *Message = data + callback to be processed*
- *Messages are process completely one by one*
 - *No "clashes" across messages' processing*
 - *Processing should not block for a long time – Workers*
- *Browser adds a new message when an event occurs and there is an event listener*

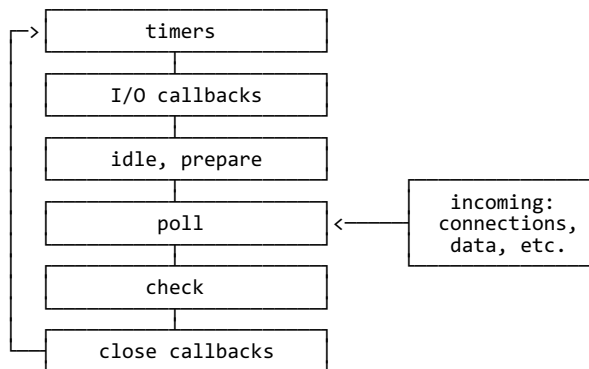
Node.js

- Node.js [🔗](#)
 - *Web server technology, very efficient and fast!*
 - *Event-driven I/O framework, based on JavaScript V8 engine*
 - *Any I/O is non-blocking (it is asynchronous)*
 - *One worker thread to process requests*
 - *You do not need to deal with concurrency issues*
 - *More threads to realize I/O*
 - *Open sourced, @GitHub [🔗](#), many libraries [🔗](#)*
 - *Future platform for Web 2.0 apps*
- Every I/O as an event
 - *reading and writing from/to files*
 - *reading and writing from/to sockets*

```
1 // pseudo code; ask for the last edited time of a file  
2 stat( 'somefile', function( result ) {  
3   // use the result here  
4 } );
```

Node.js Event Loop

- Allows Node.js to perform asynchronous I/O operations.



- Six phases, each phase has a FIFO queue of callbacks to execute.
 - **timers** – executes callbacks scheduled by `setTimeout()` and `setInterval()`
 - **I/O callbacks** – executes all I/O callbacks except close callbacks.
 - **idle/prepare** – used internally
 - **poll** – retrieve new I/O events
 - **check** – invokes `setImmediate()` callbacks
 - **close callbacks** – executes close callback, e.g. `socket.on('close', ...)`.

HTTP Server in Node.js

- HTTP Server implementation
 - server running at **127.0.0.1**, port **8080**.

```
1  const http = require('http');
2
3  const hostname = '127.0.0.1';
4  const port = 3000;
5
6  const server = http.createServer((req, res) => {
7    res.statusCode = 200;
8    res.setHeader('Content-Type', 'text/plain');
9    res.end('Hello World');
10 });
11
12 server.listen(port, hostname, () => {
13   console.log(`Server running at http://${hostname}:${port}/`);
14 });
```

Google Apps Script

- Google Apps Script
 - *JavaScript cloud scripting language*
 - *easy ways to automate tasks across Google products and third party services*
- You can
 - *Automate repetitive processes and workflows*
 - *Link Google products with third party services*
 - *Create custom spreadsheet functions*
 - *Build rich graphical user interfaces and menus*

```
1  // create spreadsheet menu
2  function onOpen() {
3      var ss = SpreadsheetApp.getActiveSpreadsheet();
4      var menuEntries = [ {name: "Say Hi", functionName: "sayHi"},
5                          {name: "Say Hello", functionName: "sayHello"} ];
6      ss.addMenu("Tutorial", menuEntries);
7  }
8
9  function sayHi() {
10     Browser.msgBox("Hi");
11 }
12
13 function sayHello() {
14     Browser.msgBox("Hello");
15 }
```

Overview

- Asynchronous I/O Overview
- Asynchronous I/O in JavaScript
- JavaScript Language Overview

Objects and Arrays

- Objects and Arrays

```
1 // objects - key/value pairs
2 var obj = { name: "Tomas", "main-city" : "Innsbruck", value : 3 };
3 obj.name = "Peter"; // assign the name property another value
4 obj["main-city"] = "Prague"; // another way to access object's values; it's not an array
5
6 // arrays
7 var a = ["Tomas", "Peter", "Alice"];
8 for (var i = 0; i < a.length; i++)
9     // do something with a[i]
10
11 // combinations of arrays and objects
12 var obj_a = [
13     { name: "Tomas", city: "Innsbruck" },
14     { name: "Peter", city: "Prague" },
15     { name: "Alice", cities: ["Prague", "Brno"] } ];
16
17 for (var j = 0; j < obj_a.length; j++)
18     // do something with obj_a[j].name, ...
```

- Functions

```
1 // assign a function to a variable
2 var minus = function(a, b) {
3     return a - b;
4 }
5
6 // call the function;
7 // now you can pass 'minus' as a parameter to another function
8 var r2 = minus(6, 4);
```

Functions

- Function Callbacks

– *You can use them to handle asynchronous events occurrences*

```
1 // function returns the result through a callback, not directly;
2 // this is not a non-blocking I/O, just demonstration of the callback
3 function add(a, b, callback) {
4     callback(a + b);
5 }
6
7 // assign the callback to a variable
8 var print = function(result) {
9     console.log(result);
10 };
11
12 // call the function with callback as a parameter
13 add(7, 8, print);
```

- Functions as values in object

```
1 var obj = {
2     data : [2, 3, "Tomas", "Alice", 4 ],
3
4     getIndexDof : function(val) {
5         for (var i = 0; i < this.data.length; i++)
6             if (this.data[i] == val)
7                 return i;
8         return -1;
9     }
10 }
11
12 obj.getIndexDof(3); // will return 1
```


Closures

- Closures

– *A function value that references variables from outside its body*

```
1  function adder() {
2      var sum = 0;
3      return function(x) {
4          sum += x;
5          return sum;
6      }
7  }
8
9  var pos = adder();
10
11 console.log(pos(3)); // returns 3
12 console.log(pos(4)); // returns 7
13 console.log(pos(5)); // returns 12
```

Objects

- **this** problem

– *A new function defines its own **this** value.*

```
1  function Person() {
2      // The Person() constructor defines `this` as an instance of itself.
3      this.age = 0;
4
5      setInterval(function growUp() {
6          // the growUp() function defines `this` as the global object,
7          // which is different from the `this`
8          // defined by the Person() constructor.
9          this.age++;
10     }, 1000);
11 }
12
13 var p = new Person();
```

– *Solution*

```
1  function Person() {
2      var that = this;
3      that.age = 0;
4
5      setInterval(function growUp() {
6          // The callback refers to the `that` variable of which
7          // the value is the expected object.
8          that.age++;
9      }, 1000);
10 }
```

Arrow Functions

- Arrow function expression
 - defined in ECMAScript 2015
 - shorter syntax than a function expression
 - non-binding of **this**

```
1  function Person(){
2    this.age = 0;
3
4    setInterval(() => {
5      this.age++; // |this| now refers to the person object
6    }, 1000);
7  }
8
9  var p = new Person();
```

- Syntax, function body

```
1  // concise body syntax, implied "return"
2  var func = x => x * x;
3
4  // with block body, explicit "return" required
5  var func = (x, y) => { return x + y; };
6
7  // object literal needs to be wrapped in parentheses
8  var func = () => ({foo: 1});
```

Callback Hell

- Callback in callback

```
1  loadScript('/my/script1.js', function(script) {
2
3    loadScript('/my/script2.js', function(script) {
4
5      loadScript('/my/script3.js', function(script) {
6        // ...continue after all script 1,2 and 3 are loaded
7      });
8    });
9  });
10
11 });
```

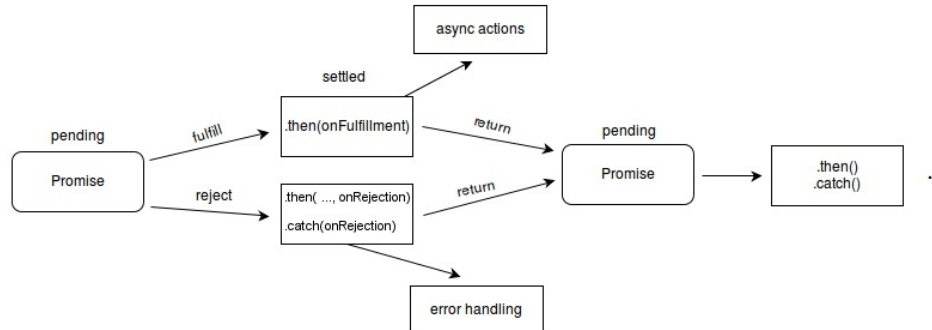
– Complex asynchronous code is hard to understand and manage

- Solution
 - Promise – a proxy to a "future" value of the function
 - Async/await – language constructs to work with asynchronous code

Promise Object

- Promise

- An object representing completion or failure of an asynchronous operation.
- A proxy for a value not necessarily known when the promise is created.



Callback Hell Example

- A callback in a callback

```
1  const request = require('request');
2
3  request("http://w20.vitvar.com/toc.json", { json: true },
4  (err, res, body) => {
5    if (err)
6      console.log("error: " + err)
7    else {
8      console.log(body)
9      request("http://mdw.vitvar.com/toc.json", { json: true },
10     (err, res, body) => {
11       if (err)
12         console.log("error: " + err)
13       else
14         console.log(body)
15     })
16   }
17 })
```

Promise Example

- A chain of Promise objects

```
1  const request = require('request');
2
3  function get_json(url) {
4      return new Promise((resolve, reject) => {
5          request(url, { json: true }, (err, res, body) => {
6              if (err)
7                  reject(err)
8              else
9                  resolve(body)
10             })
11         })
12     };
13
14     get_json('http://w20.vitvar.com/toc.json')
15     .then((data) => {
16         console.log(data)
17         return get_json('http://mdw.vitvar.com/toc.json')
18     })
19     .then((data) => {
20         console.log(data)
21     })
22     .catch((err) => {
23         console.log("error: " + err)
24     })
25 }
```

async/await

- **async**

- the function always returns a Promise
- if there is no Promise, the returned value is wrapped into Promise

```
1  async function f() {
2      return 1;
3  }
4
5  f().then((v) => alert(v));
```

- **await**

- makes program to wait until the promise is resolved or rejected
- it returns the resolved value and throws an exception when the promise is rejected
- can only be used inside **async** function

```
1  async function f() {
2      var promise = new Promise((resolve, reject) => {
3          setTimeout(() => resolve("done!"), 1000)
4      });
5
6      var result = await promise; // wait untill the promise is resolved
7
8      alert(result);
9  }
10
11  f();
```