Javascript Advanced

Autonomous Software Agents - Lab

Marco Robol - marco.robol@unitn.it

Contents

- **Advanced Asynchronous Programming** - Promises and async/await
- **The Node.js Event Loop** - Promises and process.nextTick() microtasks vs. setTimeout(), setImmediate() and IO macrotasks
  https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
- **Node.js EventEmitter** - Listening and emitting events.
  https://nodejs.dev/en/learn/the-nodejs-event-emitter/
  https://nodejs.org/api/events.html
- **Socket.IO** - Bidirectional and event-based communication between a client and a server. It is built on top of the WebSocket protocol and provides additional guarantees like fallback to HTTP long-polling or automatic reconnection.
  https://socket.io/

Advanced Asynchronous Programming

Promises

There are different ways to handle asynchronous operations in JavaScript. Execution of I/O operations is demanded to external processes or to the OS and then the result is handled once available. There are different ways to handle the asynchronous code in JavaScript which are:

- Callbacks
- **Promises** https://web.dev/promises/
- Async/Await

> https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

A promise is an object having three possible states:

- **Pending**: Initial State, before the event has happened.
- **Resolved**: After the operation completed successfully.
- **Rejected**: If the operation had error during execution, the promise fails.

A Promise is an assurance that something will be done. Promise is used to keep track of whether the asynchronous event has been executed or not and determines what happens after the event has occurred.

```
// when not yet resolved
console.log( myPendingPromise );          //Promise { <pending> }
// once resolved
console.log( Promise.resolve(123) );      //Promise { 'any value' }
// or once rejected
console.log( Promise.reject('error') );   //Promise { <rejected> }
```

Chaining `.then()` `.catch()` `.finally()`

- `.then()` is used to handle a successfully resolved promise
- `.catch()` is used for rejected promise and handling errors
- `.finally()` runs once after either .then() or .catch(), regardless of the state of the promise.

```
promise
.then(function (value) {
    console.log("Promise resolved successfully with", value);
})
.catch(function (err) {
    console.log("Promise is rejected with", err);
});
```

> https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

Transforming any *callback-based* API to a *promise-based* one

The `Promise()` constructor is primarily used to wrap functions that do not support promises. To take advantage of the readability improvement offered by promises.

```
var myPromise = new Promise( executor );
```

The `executor` ties a callback outcome to a promise. You write the executor. `resolveFunc` and `rejectFunc` are functions that accept a single parameter of any type.

```
// It receives two functions as parameters: `resolveFunc` and `rejectFunc`.
function executor( resolveFunc, rejectFunc ) {
    // Typically, some asynchronous operation that accepts a callback,
    // like the `readFile` or `setTimeout`
    ... resolveFunc(value); ... // call on resolved
    ... rejectFunc(reason); ... // call on rejected
} // return value is ignored
```

An example: setTimeout

Suppose we want a sequence of timeouts. With callback-based setTimeout() we have:

```
setTimeout(()=>{
    console.log('1000ms');          // 1000ms
    setTimeout(()=>{
            console.log('5000ms')   // 5000ms
        }, 5000)                    // then, wait for another 5 seconds
}, 1000)                            // first, wait 1 second
```

We can promisify the setTimeout API as follows:

```
promisifiedTimeout = function (time) {
    return new Promise( (res) => setTimeout( ()=>res(time+'ms'), time) )
}
```

```
promisifiedTimeout(1000)                                    // first wait for 1 second
.then( resolvedValue=>console.log(resolvedValue) )          // '1000ms'
.then( resolvedValue=>{return promisifiedTimeout(5000)} )   // then wait for additional 5
.then( console.log )                                        // '5000ms'
```

An example: readFile

```
const fs = require('fs');

const readPromisify = function (file) {
    return new Promise(function (resolve, reject) {
        fs.readFile(file, (err, data) => {
            if (err) throw reject(err);
            resolve(data); data
        });
    });
}

readPromisify('/file.md')
.then(function (data) {
      console.log("Promise resolved successfully");
})
.catch(function (err) {
    console.log("Promise is rejected");
});
```

Asynch/await

https://javascript.info/async-await

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

```
async function f() { return 1; }
f().then(alert); // 1
```

Await works only inside async functions. It waits for a promise to resolve, then return resolved value.

```
async function f() {
  let result = await promise; // wait until the promise resolves (*)
  alert(result); // "done!"
}
```

> https://www.geeksforgeeks.org/difference-between-promise-and-async-await-in-node-js/

Timeout example

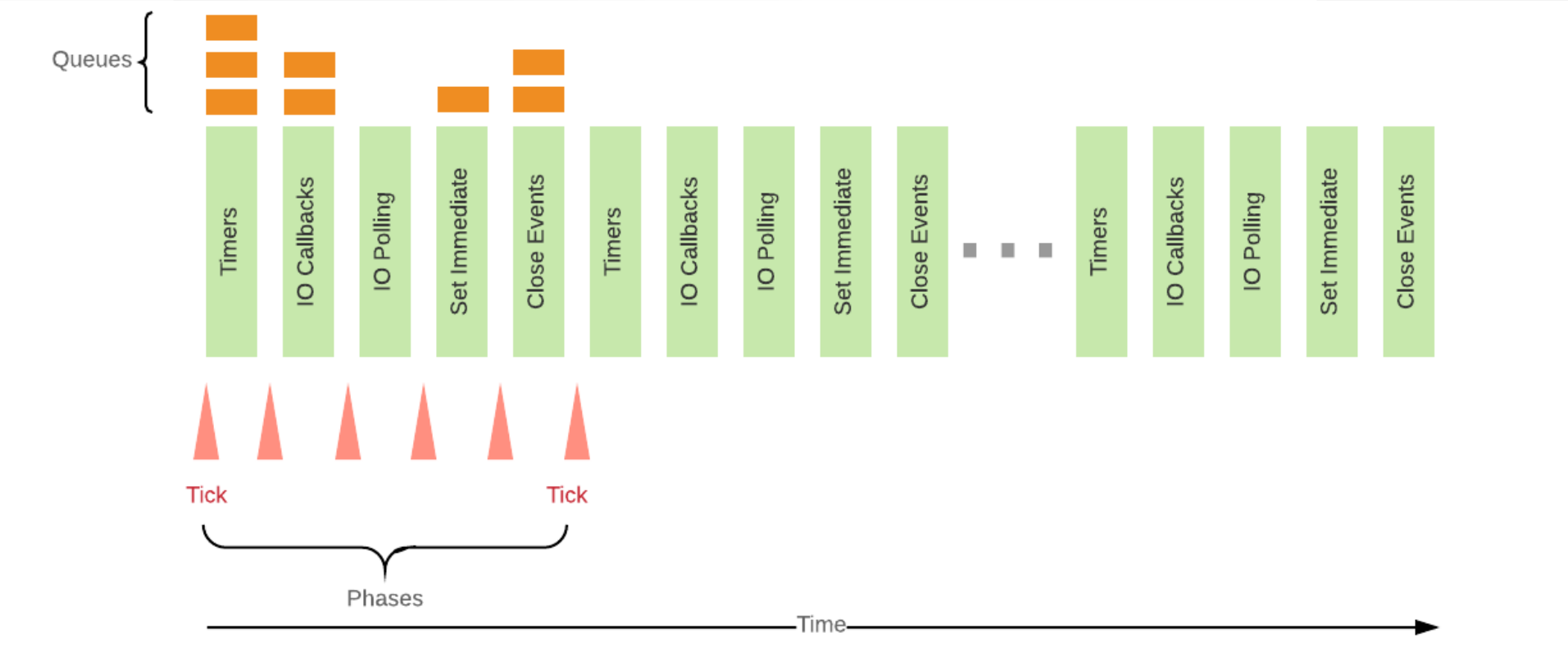
```
setTimeout(()=>{
    setTimeout(()=>{
            console.log('done')                    // done
        }, 5000)                                   // then wait for another 5 seconds
}, 1000)                                           // first wait 1 second
```

```
function example() {
    promisifiedTimeout(1000)                       // first wait for 1 second
    .then(()=>{return promisifiedTimeout(5000)})   // then wait for additional 5
    .then(()=>console.log('done');)                // done
}
```

```
async function asyncExample() {
    await promisifiedTimeout(1000);                // first wait for 1 second
    await promisifiedTimeout(5000);                // then wait for additional 5
    console.log('done');                           // done
}
```

The Node.js Event Loop

> The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.
>
> Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.
>
> https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

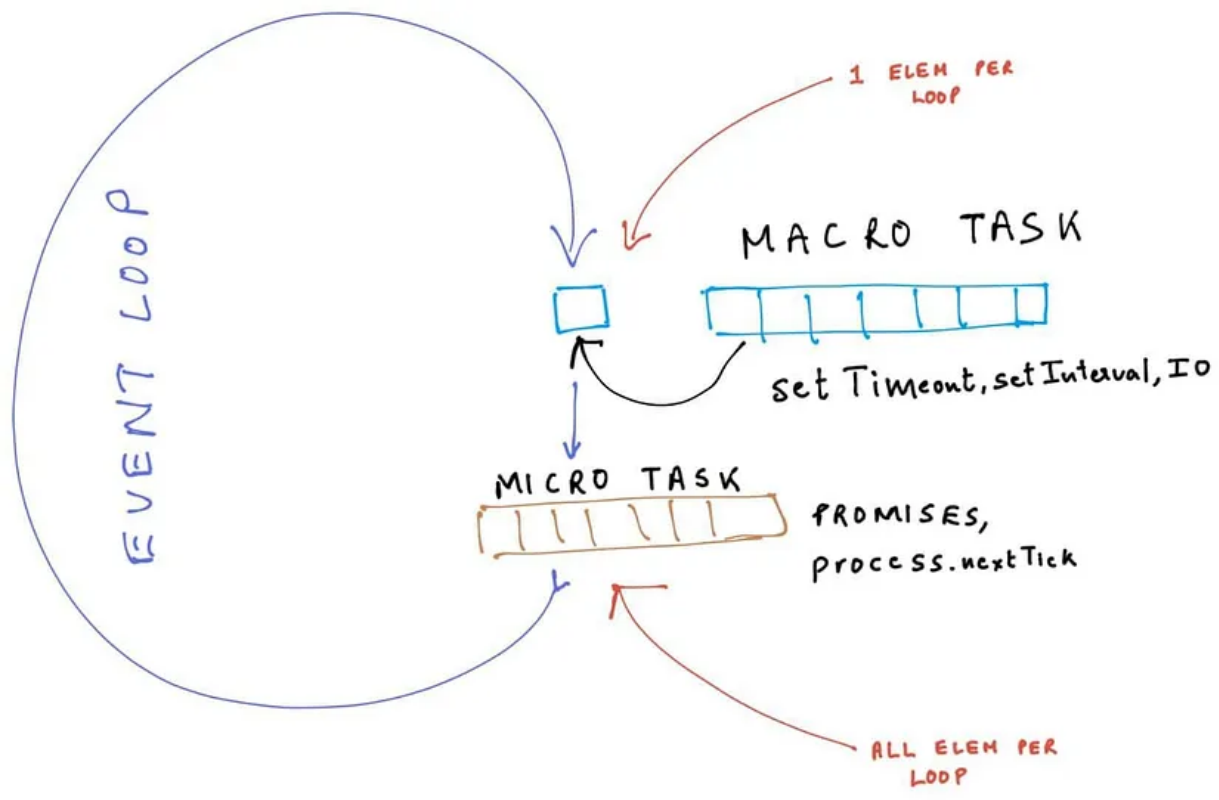


https://medium.com/the-node-js-collection/what-you-should-know-to-really-understand-the-node-js-event-loop-and-its-metrics-c4907b19da4c

Event loop executes tasks in `process.nextTick queue` first, and then executes `promises microtask queue`, and then executes `macrotask queue`.

> Each phase has a **FIFO queue** of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue **until the queue has been exhausted** or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.
>
> https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Macrotasks vs. Microtasks



https://medium.com/dkatalis/eventloop-in-nodejs-macrotasks-and-microtasks-164417e619b9.

Microtasks `process.nextTick` `Promise.then`

A `process.nextTick` callback is added to **process.nextTick** queue.

A `Promise.then` callback is added to promises **microtask** queue.

Both are executed on the current iteration of the event loop, after current operations.

> Part of the *asynchronous API*. Not technically part of the event loop.
> https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

If **microTasks** continuously add more elements to microTasks queue, **macroTasks** will stall and won't complete event loop in shorter time causing event loop delays.

```
// microtask, this would still postpone all timers and IO from being executed!!!
Promise.resolve().then( () => console.log('promise callback')  );
process.nextTick(       () => console.log('nextTick callback') );
```

Macrotasks `setTimeout` `setImmediate`

A `setTimeout` `setImmediate` callback is added to **macrotask** queue.

> How is `setImmediate()` different from `setTimeout(() => {}, 0)` (passing a 0ms timeout), and from `process.nextTick()` and `Promise.then()`?
> https://nodejs.dev/en/learn/understanding-setimmediate/

Event loop executes tasks in process.nextTick queue first, and then executes promises microtask queue, and then executes **macrotask** queue.

```
// macrotask, queues with other timers and IO
setImmediate( () => console.log('setImmediate callback')    )
setTimeout(   () => console.log('setTimeout 0 callback') , 0)
```

An example

```
// micromacro.js
setImmediate(           () => console.log('setImmediate')  );  // macrotask
setTimeout(             () => console.log('setTimeout') , 0);  // macrotask
Promise.resolve().then( () => console.log('promise')       );  // microtask
process.nextTick(       () => console.log('nextTick')      );  // microtask
console.log('main')
```

```
$ node micromacro.js
main
nextTick
promise
setTimeout
setImmediate
```

An example: Which recursive function is going to block the whole script?

```
setTimeout( () => console.log('setTimeout 100') , 100);

function recursive () {
    setImmediate( () => {
        console.log('setImmediate'); // macrotask
        recursive();
    } );
} recursive();

function recursive2 () {
    process.nextTick( () => {
        console.log('nextTick')      // microtask
        recursive2();
    } );
} recursive2();

function recursive3 () {
    console.log('main');             // main
    recursive3();
} recursive3();
```

An example: In which order will these executes?

```
for ( let i=0; i<2 ; i++) {
    process.nextTick( () => {
        console.log('nextTick');                          // microtask
        setImmediate( () => console.log('setImmediate') ); // macrotask
    } )
    console.log('main');                                  // main
}
```

Hands-on

Javascript promises - mastering the asynchronous - **codingame.com**

> Steps 5 to 17: https://www.codingame.com/playgrounds/347/

Node.js EventEmitter

> If you worked with JavaScript in the browser, you know how much of the interaction of the user is handled through events: mouse clicks, keyboard button presses, reacting to mouse movements, and so on.
>
> On the backend side, Node.js offers us the option to build a similar system using the events module.
>
> This module, in particular, offers the EventEmitter class, which we'll use to handle our events.

https://nodejs.dev/en/learn/the-nodejs-event-emitter/
https://nodejs.org/api/events.html

EventEmitter - quick start

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();

eventEmitter.on('start', () => {
    console.log('started');
});
eventEmitter.emit('start');
```

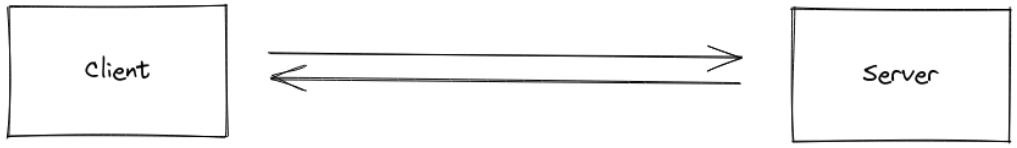
```
eventEmitter.on('start', (start, end) => {
console.log(`started from ${start} to ${end}`);
});

eventEmitter.emit('start', 1, 100);
```

> https://nodejs.dev/en/learn/the-nodejs-event-emitter/

Socket.IO

Socket.IO is a library that enables low-latency, bidirectional and event-based communication between a client and a server.



> The bidirectional channel between the Socket.IO server (*Node.js*) and the Socket.IO client (*browser, Node.js, or another programming language*) is established with a **WebSocket** connection whenever possible (HTTP long-polling as fallback).

> **WebSocket** is a communication protocol which provides a full-duplex and low-latency channel between the server and the browser.
> https://en.wikipedia.org/wiki/WebSocket

Quick start

The Socket.IO API is inspired from the Node.js **EventEmitter**, which means you can emit events on one side and register listeners on the other:

```
// Server
io.on("connection", (socket) => {
    socket.emit("hello", "world");
});
```

```
// Client
socket.on("hello", (arg) => {
    console.log(arg); // world
});
```

> https://socket.io/docs/v4/emitting-events/#basic-emit

Quick start ...This also works in the other direction:

```
// Server
io.on("connection", (socket) => {
  socket.on("hello", (arg) => {
    console.log(arg); // world
  });
});
```

```
// Client
socket.emit("hello", "world");
```

You can send any number of arguments, and all serializable data structures are supported, including binary objects like Buffer or TypedArray.

> Check out getting started project on https://socket.io/get-started/chat

Questions?

marco.robol@unitn.it