ASYNCHRONOUS JAVASCRIPT

INTRODUCTION

CALLBACKS

PROMISES

ASYNC AND AWAIT

ASYNCHRONOUS ITERATIONS

INTRODUCTION

ASINCRONÍA

- ¿Qué significa que algo es asíncrono?
- Entornos industriales -> continuado
- Otros entornos -> event-driven

CALLBACKS

Qué son

- Funciones que pasas a otra función para que se invoque cuando una condición o evento ocurra.
- **Timers**

```
setTimeout(checkForUpdates, 60000);
```

- En **ms**
- **clearInterval()**
- **clearTimeout()**

```
// Call checkForUpdates in one minute and then again every minute afi
let updateIntervalId = setInterval(checkForUpdates, 60000);

// setInterval() returns a value that we can use to stop the repeatec
// invocations by calling clearInterval(). (Similarly, setTimeout()
// returns a value that you can pass to clearTimeout())
function stopCheckingForUpdates() {
    clearInterval(updateIntervalId);
}
```

CALLBACKS

Qué son

- **Events:**

```
// Ask the web browser to return an object representing the HTML
// <button> element that matches this CSS selector
let okay = document.querySelector('#confirmUpdateDialog button.okay');

// Now register a callback function to be invoked when the user
// clicks on that button.
okay.addEventListener('click', applyUpdate);
```

- Un timer es un evento de tiempo
- Eventos de user (botones, recibir información de internet…)

CALLBACKS

Qué son

- **Network Events:**

```
function getCurrentVersionNumber(versionCallback) { // Note callback
    // Make a scripted HTTP request to a backend version API
    let request = new XMLHttpRequest();
    request.open("GET", "http://www.example.com/api/version");
    request.send();

    // Register a callback that will be invoked when the response ar
    request.onload = function() {
        if (request.status === 200) {
            // If HTTP status is good, get version number and call ca
            let currentVersion = parseFloat(request.responseText);
            versionCallback(null, currentVersion);
        } else {
            // Otherwise report an error to the callback
            versionCallback(response.statusText, null);
        }
    };
    // Register another callback that will be invoked for network er
    request.onerror = request.ontimeout = function(e) {
        versionCallback(e.type, null);
    };
}
```

PROMISES

Qué es

- Una herramienta de javascript exclusive para trabajos asíncronos
- Representa el resultado de una ejecución asíncrona.
- Permite la encadenación de callbacks más sencilla.
- También permite gestionar excepciones durante las callbacks
- **No** se **debe utilzar en eventos repetitivos**

NOTA: Merece la pena investigar sobre las promises para entenderlas adecuadamente.

PROMISES

Funcionamiento

- La palabra **then** permite definir la función a la que se le llamará después de la function inicial.
- Permite gestionar excepciones con **catch**

```
// Suppose you have a function like this to display a user profile
function displayUserProfile(profile) { /* implementation omitted */ }

// Here's how you might use that function with a Promise.
// Notice how this line of code reads almost like an English sentence
getJSON("/api/user/profile").then(displayUserProfile);
```

```
getJSON("/api/user/profile").then(displayUserProfile).catch(handleProfileError)
;
```

PROMISES

Encadenamiento de promesas

- Cada then debe **devolver** un **objeto Promise** también

```
fetch(documentURL)                          // Make an HTTP request
    .then(response => response.json())      // Ask for the JSON body of
    .then(document => {                     // When we get the parsed JS
        return render(document);            // display the document to t
    })
    .then(rendered => {                     // When we get the rendered
        cacheInDatabase(rendered);          // cache it in the local dat
    })
    .catch(error => handle(error));         // Handle any errors that oc
```

PROMISES

Encadenamiento de promesas

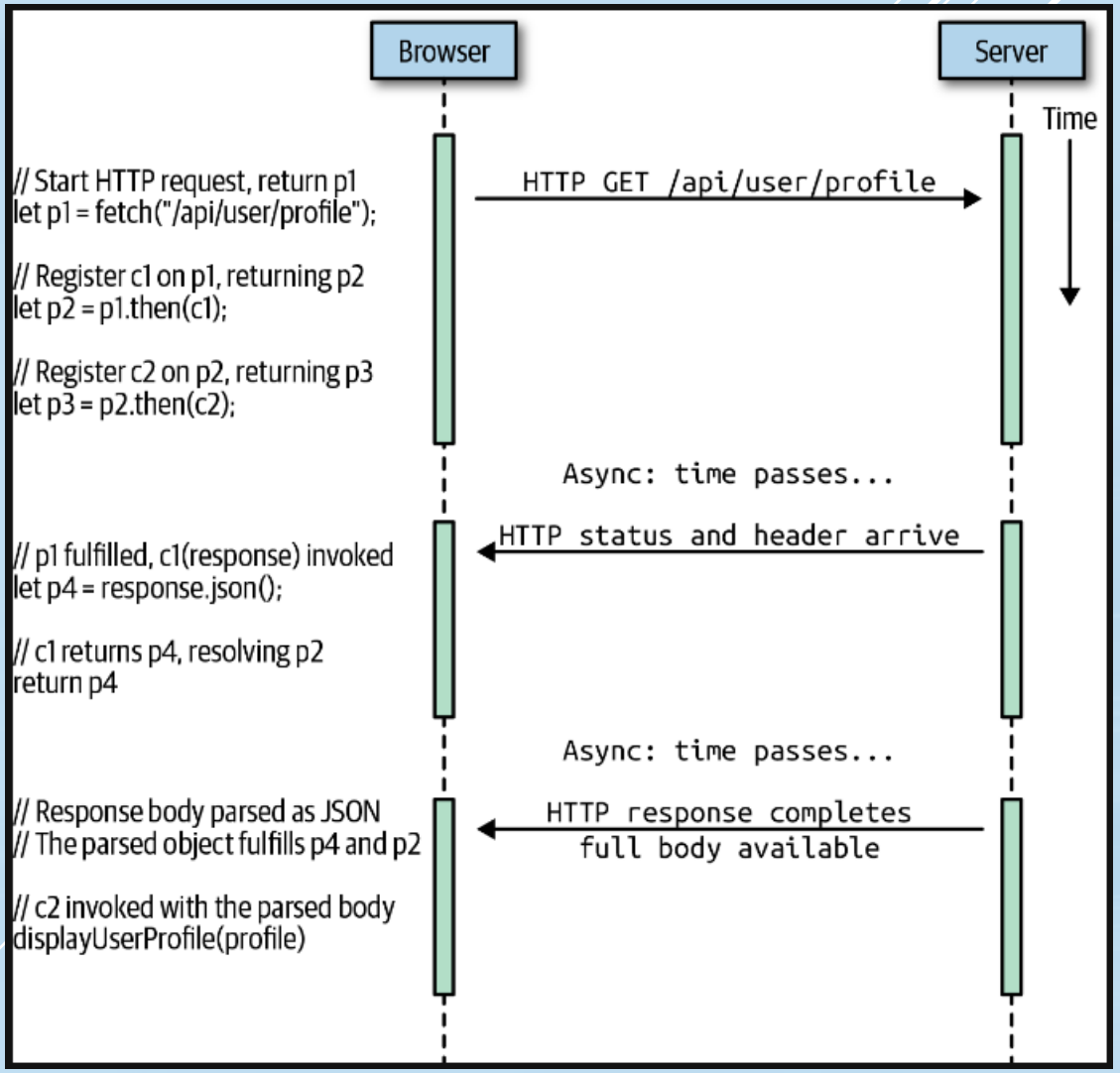
- Similar ejemplo por partes

```
function c1(response) {                 // callback 1
    let p4 = response.json();
    return p4;                          // returns promise 4
}

function c2(profile) {                  // callback 2
    displayUserProfile(profile);
}

let p1 = fetch("/api/user/profile");    // promise 1, task 1
let p2 = p1.then(c1);                   // promise 2, task 2
let p3 = p2.then(c2);                   // promise 3, task 3
```

PROMISES

Encadenamiento de promesas

- Ejemplo en el transcurso del tiempo



PROMISES

Ejemplos return promises

- wait

```
function wait(duration) {
    // Create and return a new Promise
    return new Promise((resolve, reject) => { // These control the P
        // If the argument is invalid, reject the Promise
        if (duration < 0) {
            reject(new Error("Time travel not yet implemented"));
        }
        // Otherwise, wait asynchronously and then resolve the Promi
        // setTimeout will invoke resolve() with no arguments, which
        // that the Promise will fulfill with the undefined value.
        setTimeout(resolve, duration);
    });
}
```

PROMISES

Ejemplos return promises

- getJson

```
const http = require("http");

function getJSON(url) {
    // Create and return a new Promise
    return new Promise((resolve, reject) => {
        // Start an HTTP GET request for the specified URL
        request = http.get(url, response => { // called when response
            // Reject the Promise if the HTTP status is wrong
            if (response.statusCode !== 200) {
                reject(new Error(`HTTP status ${response.statusCode}`
                response.resume();  // so we don't leak memory
            }
            // And reject if the response headers are wrong
            else if (response.headers["content-type"] !== "applicati
                reject(new Error("Invalid content-type"));
                response.resume();  // don't leak memory
            }
            else {
```

```
            else {
                // Otherwise, register events to read the body of the
                let body = "";
                response.setEncoding("utf-8");
                response.on("data", chunk => { body += chunk; });
                response.on("end", () => {
                    // When the response body is complete, try to par
                    try {
                        let parsed = JSON.parse(body);
                        // If it parsed successfully, fulfill the Pro
                        resolve(parsed);
                    } catch(e) {
                        // If parsing failed, reject the Promise
                        reject(e);
                    }
                });
            }
        });
        // We also reject the Promise if the request fails before we
        // even get a response (such as when the network is down)
        request.on("error", error => {
            reject(error);
        });
    });
}
```

ASYNC AND AWAIT

Simplificación de promesas

- **await** convierte un objeto promise en un valor de retorno
- Parece síncrono pero **NO** lo es

```
let response = await fetch("/api/user/profile");
let profile = await response.json();
```

- Es obligatorio que la función que utilice await tenga definido un **async** en la cabecera

ASYNC AND AWAIT

Simplificación de promesas

- **async** convierte cualquier función en un objeto promise

```
async function getHighScore() {
    let response = await fetch("/api/user/profile");
    let profile = await response.json();
    return profile.highScore;
}
```

- **Para paralelizar** varios await, irán en orden aunque no sea necesario. Para paralelizarlo mejor, utilizar **Promise.all**.

ASYNCHRONOUS ITERATIONS

Para qué

- Sirven para utilizar funciones que **sí se repiten**
- Se utiliza las keys **for/await**

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

async function parseFile(filename) {
    let stream = fs.createReadStream(filename, { encoding: "utf-8"});
    for await (let chunk of stream) {
        parseChunk(chunk); // Assume parseChunk() is defined elsewher
    }
}
```

ASYNCHRONOUS ITERATIONS

Ejemplo

- Actualizar varias urls de manera asíncrona.

```
for(const promise of promises) {
    response = await promise;
    handle(response);
}
```

```
for await (const response of promises) {
    handle(response);
}
```

ASYNCHRONOUS ITERATIONS

SUMMUM

- Generadores de funciones Asíncronas

- Busca más información en internet

```
// A Promise-based wrapper around setTimeout() that we can use await
// Returns a Promise that fulfills in the specified number of millise
function elapsedTime(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

// An async generator function that increments a counter and yields i
// a specified (or infinite) number of times at a specified interval.
async function* clock(interval, max=Infinity) {
    for(let count = 1; count <= max; count++) { // regular for loop
        await elapsedTime(interval);            // wait for time to p
        yield count;                            // yield the counter
    }
}

// A test function that uses the async generator with for/await
async function test() {                       // Async so we can use
    for await (let tick of clock(300, 100)) { // Loop 100 times every
        console.log(tick);
    }
}
```

BIBLIOGRAFÍA

1. JavaScript: The Definitive Guide, 7th Edition
2. Eloquent JavaScript, 3th edition, Marijn Haverbeke
3. Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)
4. Modern c++ Programming with Test-Driven Development, Jeff Langr
5. Refactoring: Improve the design of existing Code, Martin Fowler
6. Game programming patterns, Robert Nystrom