Sesión 3 – Semana 1

Peticiones con Axios y modularización

Carrera 43 A # 34 - 155. Almacentro. Torre Norte. Oficina 701
Medellín (Antioquia), Colombia

Contenido

1. Promesas
    1. Recordando que son promesas
    2. Métodos de promesas
2. Peticiones HTTPs
    1. Recordando que son métodos HTTPS
    2. Axios
3. Modularización

Promesas

1. Una promesa es una manera simple de manejar operaciones asíncronas en JavaScript. Se utiliza para averiguar si la operación asíncrona se completó con éxito o no.
2. Son fáciles de administrar cuando se trata de múltiples operaciones asincrónicas donde los callbacks pueden crear un callback hell que conducen a un código inmanejable.

¿Qué son?

Antes de las promesas, se usaban **eventos** y **callbacks**, pero:

1. Los **callbacks** tenían funcionalidades limitadas y creaban un código inmanejable
2. Los **eventos** no eran buenos para manejar operaciones asincrónicas.

Promesas

Beneficios y aplicaciones

1. Mejora la legibilidad del código
2. Mejor manejo de operaciones asíncronas
3. Mejor flujo de definición de control en lógica asíncrona
4. Mejor manejo de errores

1. Las promesas se utilizan para el manejo asíncrono de eventos.
2. Las promesas se utilizan para manejar solicitudes http asíncronas, entre otras.

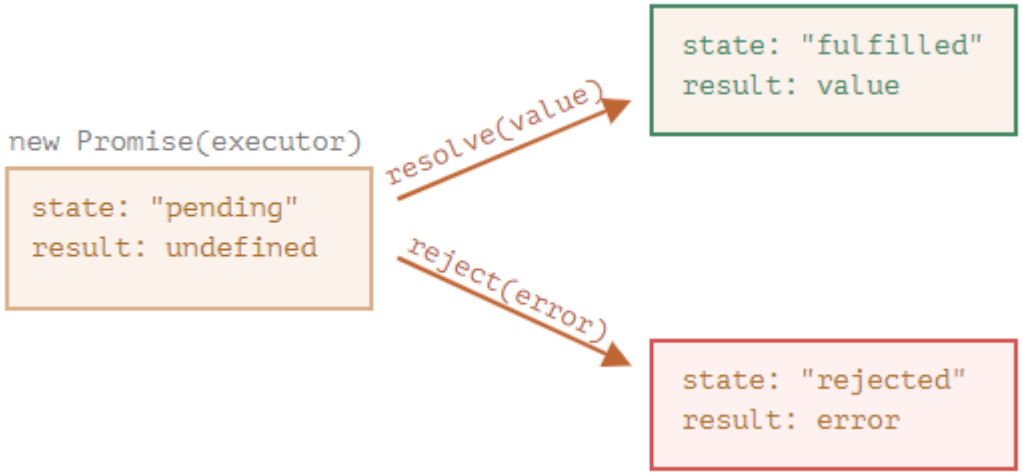
Promesas

Estados

El objeto **`Promise`** representa la eventual finalización (o falla) de una operación asincrónica y su valor resultante.

Comienza en un estado pendiente (pending). Eso significa que el proceso no está completo. Si la operación tiene éxito, el proceso finaliza en un estado **fulfilled**. Y, si ocurre un error, el proceso termina en un estado **rejected**.

Por ejemplo, cuando solicita datos del servidor mediante una promesa, estará en estado pendiente. Cuando los datos lleguen con éxito, estarán en un estado completo. Si ocurre un error, entonces estará en un estado rechazado.



Promesas

Estados

En resumen, una promesa puede presentar los siguientes estados:

1. **`fulfilled`**: La acción relacionada a la promesa se llevó a cabo con éxito.
2. **`rejected`**: La acción relacionada a la promesa falló.
3. **`pending`**: Aún no se ha determinado si la promesa fue **`fulfilled`** o **`rejected`**.
4. **`settled`**: Ya se ha determinado si la promesa fue **`fulfilled`** o **`rejected`**

También se suele usar el término **`thenable`**, para indicar que un objeto tiene disponible un método "**`then`**" (y que por tanto está relacionado con Promesas).

Promesas

¿Cómo crearlas?

Usando el constructor `new Promise`:

```
const myPromise = new Promise
((resolve, reject)=>{
    //Instrucciones
});
```

```
const myPromise = ()=>{
    return new Promise((resolve, reject)=>{
        //Instrucciones
    })
}
```

```
const myPromise = ()=> new
Promise((resolve, reject)=>{
        //Instrucciones
    })
```

MAKAIA

WWW.MAKAIA.ORG

Promesas

¿Cómo consumirlas?

Las promesas pueden ser consumidas empleando los métodos `.then()`, `.catch()` y `.finally()`.

`.then ()`

Se ejecuta cuando una promesa se resuelve o se rechaza.

Parámetros:

Toma dos funciones como parámetros:

1. La primera función se ejecuta si se resuelve la promesa y se recibe un resultado.
2. La segunda función se ejecuta si se rechaza la promesa y se recibe un error. (Es opcional y hay una mejor manera de manejar el error usando el método **`.catch()`**

```
.then(function(result){
        //handle success
    }, function(error){
        //handle error
    })
```

Sintaxis

Carrera 43 A # 34 - 155. Almacentro. Torre Norte. Oficina 701
Medellín (Antioquia), Colombia

MAKAIA
WWW.MAKAIA.ORG

Promesas

¿Cómo consumirlas?

Las promesas pueden ser consumidas empleando los métodos `.then()`, `.catch()` y `.finally()`.

`.catch ()`

Se ejecuta cuando se rechaza una promesa o se produce algún error en la ejecución. Se utiliza como controlador de errores cada vez que en cualquier paso existe la posibilidad de obtener un error.

Parámetros:

Toma una función como parámetros:

1. Función para manejar errores o rechazos de promesas. (El método **`.catch()`** llama internamente a **`.then(null, errorHandler)`**, es decir, **`.catch()`** es solo una forma abreviada de **`.then(null, errorHandler)`**)

```
.catch(function(error){
        //handle error
    })
```

Sintaxis

Carrera 43 A # 34 - 155. Almacentro. Torre Norte. Oficina 701
Medellín (Antioquia), Colombia

MAKAIA
WWW.MAKAIA.ORG

Promesas

¿Cómo consumirlas?

Las promesas pueden ser consumidas empleando los métodos `.then()`, `.catch()` y `.finally()`.

`.finally ()`

se ejecuta cuando la promesa se resuelve con éxito o se rechaza.

Parámetros:

Toma una función como parámetros.

```
.finally(()=>{
    //Instrucciones que se ejecutarán
    cuando la promesa retorne una
    respuesta
});
```

Sintaxis

Métodos del objeto promesas

1. `Promise.all()`: Recibe un iterable de promesas como entrada y devuelve una única promesa que se resuelve en una matriz de los resultados de las promesas de entrada. Esta promesa devuelta se cumplirá cuando se hayan cumplido todas las promesas de entrada, o si la iterable de entrada no contiene promesas. Se rechaza inmediatamente después de que cualquiera de las promesas de entrada sea rechazada o no arroje un error, y en este caso retornará el primer mensaje de rechazo/error.

   Sintaxis:

   ```
   Promise.all(iterable)
   ```

   recibe como parámetro un `objeto iterable` como un `Array`.

Métodos del objeto promesas

2. `Promise. allSettled()`: Devuelve una promesa que se cumple después de que todas las promesas dadas se hayan cumplido o rechazado, con una matriz de objetos que describen el resultado de cada promesa.

   Sintaxis:

   ```
   Promise.allSettled (iterable)
   ```

   recibe como parámetro un `objeto iterable` como un `Array` donde cada elemento sea una promesa.

Métodos del objeto promesas

3. `Promise. any():` Recibe un iterable de promesas como entrada y devuelve una única promesa que se cumple tan pronto como se resuelve alguna de las promesas en el iterable, con el valor de la promesa cumplida. Si no hay promesas en el cumplimiento iterable (si se rechazan todas las promesas dadas), la promesa devuelta se rechaza con un AggregateError, una nueva subclase de Error que agrupa errores individuales.

   Sintaxis:

   ```
   Promise.any (iterable)
   ```

   recibe como parámetro un `objeto iterable` como un `Array`.

Métodos del objeto promesas

4. `Promise. race(): Devuelve una promesa que cumple o rechaza tan pronto como una de las promesas en un iterable cumple o rechaza, con el valor o la razón de esa promesa.

Sintaxis:

```
Promise.race (iterable)
```

recibe como parámetro un `objeto iterable` como un `Array`.

Peticiones HTTPs

¿Qué son?

Define un conjunto de métodos de petición para indicar la acción que se desea realizar para un recurso determinado. Aunque estos también pueden ser sustantivos, estos métodos de solicitud a veces son llamados HTTP verbs.

¿Cuáles son esos métodos de petición?

GET, POST, PUT, PATCH y DELETE.

Axios

¿Qué es?

Axios es una biblioteca de código abierto para hacer solicitudes HTTP.

¿Cómo empezar?

Instalación

1. Usando npm:
   ```
   npm install axios
   ```
2. Usando JsDelivr CDN:
   ```
   <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
   ```
3. Usando un pkg CDN:
   ```
   <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
   ```

Si Axios es instalado usando el npm , entonces necesitaría importarlo

```
import axios from 'axios'
```

¿Cómo usar Axios?

vs `fetch()`

Petición GET

```
const Get = async (url) => {
    const { data } = await axios.get(url);
    return data;
}
```

```
const Get = async (url) => {
    const resp = await fetch(url);
    const data = await resp.json();
    return data
}
```

¿Cómo usar Axios?

Petición POST

```
const Post = async (url, object = {}) => {
    await axios.post(url, object);
}
```

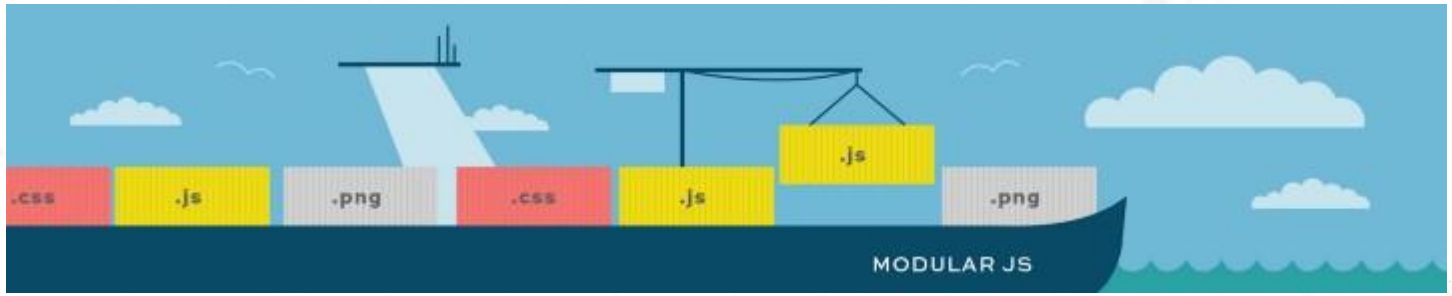
vs `fetch()`

```
const Post = async (url, obj) => {
    try {
        await fetch(url, {
            method: 'POST',
            headers: {
                "Content-Type": "application/json"
            },
            body: JSON.stringify(obj)
        })
    } catch (error) {
        console.log(error)
    }
}
```

Modularización

¿Qué es?

Es un método para separar funcionalidades independientes, de modo que podamos cargar cualquier función que necesitemos. Cada porción es un módulo que encapsulan una o varias funciones o tareas.



Carrera 43 A # 34 - 155. Almacentro. Torre Norte. Oficina 701
Medellín (Antioquia), Colombia

MAKAIA
WWW.MAKAIA.ORG

Modularización

Características y ventajas

1. Los módulos son “pequeños” fragmentos de código muy especializados
2. Los módulos son “independientes” y reemplazables por otros
3. Los módulos tienen dependencias entre sí, en ocasiones cruzadas
4. Los módulos permiten encapsular mucha lógica en forma de cajas negras
5. Una buena modularización permite subir el nivel de abstracción de una aplicación
6. Permiten la reutilización del código
7. Mantienen la calidad del código

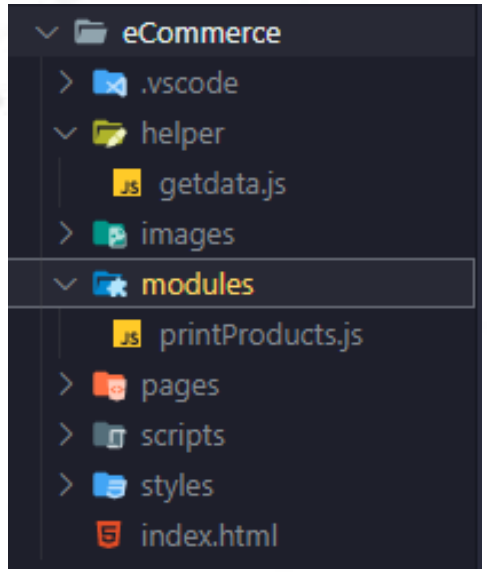
Modularización

¿Cómo hacerlo?

1. Cuando trabajamos con módulo lo primero que debemos hacer es incluir en la etiqueta `</script>` de nuestro `archivo.html` el atributo `type="module"`.

   ```
   <script src="./scripts/index.js" type="module"></script>
   ```

2. En la estructura de carpetas de nuestro repositorio, incluimos carpetas (llamadas **`helpers`** para albergar los módulos que nos permitan realizar peticiones HTTPs y **`modules`** para albergar paquetes de código independientes de funcionalidades auxiliares que permitan interactuar con el DOM, por ejemplo) que alojarán los diferentes archivos scripts (o módulos) diferenciados por su funcionalidad y alcance.



Modularización

¿Cómo hacerlo?

3. Desde los scripts se deben emplear las directivas especiales **`export`** e **`import`** para intercambiar funcionalidad y llamar a funciones de un módulo de otro:
   - La palabra clave **`export`** etiqueta las variables y funciones que deberían ser accesibles desde fuera del módulo actual.
   - **`import`** permite importar funcionalidades desde otros módulos.

Modularización

import & export

Las directivas export e import tienen varias variantes de sintaxis como:

Cuando export está antes de las sentencias

```
// exportar un array
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec'];

// exportar una constante
export const MODULES_BECAME_STANDARD_YEAR = 2022;

// exportar una función
export const sayHi=(user)=>{
    alert(`Hello, ${user}`);
}
```

import se emplea de estas maneras:

```
//Para llamar funciones, variables o constantes
específicas
import {sayHi, months} from './module.js'

sayHi('Jhon');
const lengthArray = months.length;

//Para importar todo como un objeto
import * as module from './module.js'

module.sayHi('Jhon');
const lengthArreglo = module.months.length;
```

Modularización

import & export

Cuando empleamos `export default`

```
/ exportar un array
export default let months = ['Jan', 'Feb', 'Mar','Apr',
'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];


// exportar una función
const sayHi=(user)=>{
    alert(`Hello, ${user}`);
}

export default sayHi;
```

`import` se emplea de estas maneras:

```
//Para llamar funciones, variables o constantes
específicas
import months from './module.js'

const lengthArray = months.length;

//Para importar todo como un objeto
import sayHi from './module.js'

sayHi('Jhon');
```

Fuentes

1. https://www.programiz.com/javascript/promise
2. https://www.geeksforgeeks.org/javascript-promises/
3. https://programacionymas.com/blog/promesas-javascript
4. https://developer.mozilla.org/es/docs/Web/HTTP/Methods
5. https://axios-http.com/docs/intro
6. https://medium.com/@sebastianpaduano/modularizaci%C3%B3n-en-javascript-538bd6c75fa
7. https://es.javascript.info/import-export
8. https://es.javascript.info/modules-intro

MAKAIA

WWW.MAKAIA.ORG

Info: comunicaciones@makaia.org

Corporación MAKAIA
Medellín, Colombia
Carrera 43A – 34-155. Almacentro
Torre Norte, Oficina 701
Teléfono: (+574) 448 03 74
Móvil: (+57) 320 761 01 76

@makaiaorg