10. ASYNC/AWAIT JAVASCRIPT

IES ESTACIÓ CURS 2021- 2022

Índex

1. Introducció.

- El codi asíncron és aquell que ix del fil principal d'execució de Javascript, executant-se en paral·lel. Si bé Javascript és un llenguatge síncron per naturalesa, existeixen unes certes funcionalitats en els entorns d'execució de Javascript o en els navegadors que permeten l'execució de codi asíncron.

- Inicialment, el codi asíncron en Javascript s'executava únicament gràcies a l'ús de **Callbacks**. No obstant això, en la versió ES2015 de Javascript es van introduir les **Promeses** per a evitar uns certs problemes que feien que l'ús de **Callbacks** es complicara bastant quan es niaven. Però amb l'objectiu de facilitar encara més el seu ús, es va introduir la sintaxi **async/await** en la versió ES2017 de Javascript.

- Aquesta sintaxi permet crear funcions que usen tant **Promises** com generadors, sent una abstracció d'alt nivell de les promeses de Javascript. L'única diferència és que les funcions que utilitzen **async/await**, gràcies a l'ús de generadors, poden pausarse a si mateixes i continuar la seua execució en un altre moment.

2. Què és la sintaxi async/away.

- Quan les **Promises** van aparéixer en la versió ES2015 de Javascript, resolien els problemes que es donaven amb l'ús de **Callbacks** a l'hora de gestionar el codi asíncron.

- El major era el **Callback Hell**, que es donava quan es niaven diversos **Callbacks** que executaven codi asíncron, ja que el codi es tornava complicat i poc llegible.

- No obstant això, les **Promises** eren millorables, ja que també agregaven la seua pròpia complexitat i una sintaxi difícil d'entendre.

- Per això, van aparéixer les funcions asíncrones, que usen la sintaxi **async/await**, permetent seguir un estil de programació síncron encara creant funcions asíncrones.

3. Com crear una funció asíncrona amb async.

- Crear una funció asíncrona amb **async**:

```
async function obtenerDatos() {
  return 'hola!';
}
```

- Aquesta funció no gestiona codi asíncron, però funciona d'un mode diferent de les funcions tradicionals, ja que quan l'executes podràs comprovar que retorna una promesa. De fet, si inspecciones el codi, podràs comprovar que la promesa que retorna inclou les propietats PromiseStatus i PromiseValue. Per comprovar-lo executa la funció obtenerDatos:

```
console.log(obtenerDatos());
```

- Resultat per consola:

```
__proto__: Promise
[[PromiseStatus]]: "fulfilled"
[[PromiseValue]]: Object
```

3. Com crear una funció asíncrona amb async.

- Açò significa que podràs gestionar qualsevol funció que use **async** com una promesa, podent utilitzar **then**, **catch** o **finally** igual que ho faries en una promesa:

```
obtenerDatos().then(response => console.log(response));
```

- En executar el codi anterior hauries de veure el següent resultat per la consola: `hola!`
- Quan utilitzes **async** amb una funció, es retornarà una promesa encara que no la retornes explícitament, ja que l'entorn d'execució de Javascript s'encarregarà que així siga. Per això, les dues funcions que veus a continuació són equivalents:

```
const funcionA = async () => {
  return 'Hola!';
}

funcionA().then((resultado) => console.log(resultado));

const functionB = async () => {
  return Promise.resolve('Hola!');
}

funcionB().then((resultado) => console.log(resultado));
```

4. Com usar await en una funció assíncrona.

- Les funcions asíncrones permeten executar promeses en el seu interior mitjançant l'operador **await**, que esperarà que la promesa regire un valor abans de continuar l'execució del codi.
- Posarem com a exemple la següent promesa:

```
const tareaAsincrona = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('Tarea completada'), 3000);
  })
}
```

- Mitjançant l'ús d'**await** en una funció asíncrona, podrem executar la promesa i obtenir el seu resultat, pausant l'execució de la funció fins que la promesa retorne un resultat.

```
async function gestionarTareas() {
  const response = await tareaAsincrona();
  console.log(response);
}
```

4. Com usar await en una funció assíncrona.

- Es pot usar **await** en les funcions asíncrones declarades amb **async**.
- De la mateixa manera, també pots fer servir **await** amb la funció **fetch** per a realitzar una petició a una API.
- En el següent exemple obtenim les dades d'un usuari des de **GitHub** emprant **l'API Fetch** en una funció asíncrona:

```
async function obtenerUsuario() {
        const response = await fetch('https://api.github.com/users/edulazaro’);
        const data = await response.json();
        console.log(data);
}
obtenerUsuario();
```

4. Com usar await en una funció assíncrona.

- L'operador **await** que anteposem a la funció **fetch** a l'interior de la funció **obtenerUsuario** evitarà que la línia següent s'execute fins que no obtinguem un valor de tornada.
- D'aquesta manera ens assegurem que la constant data no estiga buida.
- Com veus, no és necessari usar **then**, com sí que fem amb les promeses, per a obtenir el resultat.
- Aquest serà el resultat que es mostre per la consola:

```
blog: "https://www.neoguias.com"
twitter_username: "neeonez"
type: "User"
url: "https://api.github.com/users/edulazaro"
updated_at: "2021-01-14T11:22:45Z"
....
```

4. Com usar await en una funció assíncrona.

- No obstant això, existeix una gran diferència entre l'ús d'**await** i l'ús de **then()** amb una promesa.
- L'ús d'**await** suspén l'execució de la funció actual fins que el codi de context superior s'acabe d'executar, mentre que usant **then()**, el cos de la promesa es continuarà executant.
- Ho veurem amb un exemple:

```
const miFuncion= () => Promise.resolve('Tres');

async function  miFuncion() {
  console.log('Dos');
  const res = await getC();
  console.log(res);
}

console.log('Uno')
miFuncion();
console.log('Cuatro');
```

4. Com usar await en una funció assíncrona.

- En l'exemple anterior, primer es mostrarà **Un** per la consola, després **Dues**, seguidament **Quatre** i finalment **Tres**.
- Això és pel fet que, quan s'executa la funció **miFuncion()**, s'obté el resultat de **getC()**, però abans de ser assignat a la constant **res**, atés que hem usat **await**, es finalitzarà primer tot el codi dels contextos superiors.
- Per això, abans que es mostre **Tres** per pantalla, s'executarà primer el **console.log('Quatre')** i la resta del codi d'aquest context, d'haver-lo.
- En finalitzar, es reprén l'execució de la funció **miFuncion()**, assignant-se **Tres** al constant **res** i executant-se finalment el **console.log(res).**
- Si haguérem utilitzat un **await** amb la funció **miFuncion(),** en cas d'estar la crida a l'interior d'una funció asíncrona, el resultat seria un altre.
- Primer es mostraria **Un** per la consola, després **Dues**, seguidament **Tres** i finalment **Quatre**.

5. Com gestionar errors amb catch.

- Atés que estem treballant amb funcions asíncrones, també podem gestionar els errors d'aquestes.

- Quan s'utilitza una promesa, tens disponible el mètode **catch**, usat al costat del mètode **then**, que et permet capturar els possibles errors que ocórreguen durant la seua execució.

- No obstant això, en aquest cas farem servir una sentència **try/catch** per a obtindre el mateix efecte.

5. Com gestionar errors amb catch.

- Gestionarem les possibles excepcions de l'exemple de l'apartat anterior, en el qual obteníem un usuari de **GitHub** usant **fetch**:

```
async function obtenerUsuario() {
        try{
                // Si todo ha ido bien
                const response = await fetch('https://api.github.com/users/edulazaro');
                const data = await response.json(); console.log(data);
        } catch(error) {
                // Ha ocurrido algún errorconsole.log(error);
        }
}
obtenerUsuario();
```

- El codi anterior saltarà fins a la sentència **catch** en cas que reba un error, mostrant-lo per la consola.

6. Per què has d'usar async/await.

- Tal com hem vist, el codi és molt més fàcil d'entendre quan utilitzes **async/await** en comparació amb l'ús de promeses.
- No obstant això, aquest fenomen s'accentua més quan en una promesa a l'utilitzar diverses sentències **then** en cadena, es quan més percebes els seus avantatges.
- Veurem un exemple un poc més complex en el qual obtindrem un arxiu JSON per a després fer diverses tasques amb ell.
- Primer usarem una promesa i després async/await.
- En el codi que es veu a continuació obtenim un arxiu JSON.
- Després obtenim el primer element del mateix i seguidament realitzem una petició a una API per a obtenir un recurs, per a finalment transformar el resultat a format JSON:

6. Per què has d'usar async/await.

```
const obtenerPrimeraFila = () => {
  return fetch('/filas.json')
    .then(res => res.json())
    .then(filas => filas[0])
    .then(fila => fetch(`/filas/${fila.id}`))
    .then(res => res.json());
}

obtenerPrimeraFila();
```

6. Per què has d'usar async/await.

- Si ara usem **async/await**, comprovaràs que el codi és més senzill:

```
const obtenerPrimeraFila = async () => {
  const res = await fetch('/filas.json');
  const filas = await res.json();
  const fila = filas[0];
  const resFila = await fetch(`/filas/${fila.id}`);
  const datos = await resFila.json();
  return datos;
}

obtenerPrimeraFila();
```

- Un altre avantatge és que la depuració del codi serà molt senzilla en comparació amb l'ús de promeses.
- Això és pel fet que per defecte, el depurador no pararà la seua execució amb el codi asíncron, però sí que el farà quan uses **async/await**, ja que s'executa com el codi síncron.

7. Com encadenar funcions asíncrones.

- Les funcions asíncrones també poden encadenar-se igual que les promeses, i a més amb una sintaxi més fàcil d'entendre, ja que bastarà amb usar l'operador +.
- A continuació pots veure un exemple:

```
const comerFruta = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('Me gusta comer fruta'), 1000);
  });
}

const comerFrutaVerdura = async () => {
  const fruta = await comerFruta();
  return fruta + ' y verdura';
}

comerFrutaVerdura().then(res => {
  console.log(res);
});
```

7. Com encadenar funcions asíncrones.

- El codi anterior mostrarà el següent missatge per la consola quan executes la funció **comerFruta()**:

```
comerFruta(); // Me gusta comer fruta y verdura
```

8. Limitacions de l'ús de async/await.

- Actualment resulta molt més freqüent l'ús de async/await que l'ús de promeses.
- No obstant això, les promeses disposen de funcionalitats addicionals que no podràs aconseguir amb async/await.
- Un exemple d'això és la combinació de diverses promeses mitjançant **Promise.all()** o el mètode **Promise.race().**
- En el fons, quan uses **async/await** estàs usant promeses juntament amb generadors, que són capaços de pausar l'execució del codi, fent que el codi siga més flexible.

9. Alternatives a async/await.

- En l'actualitat, el recomanable és que uses **async/await** sempre que siga possible en lloc d'usar promeses directament, ja que el codi serà més llegible i fàcil d'entendre per altres desenvolupadors.

- No obstant això, a vegades pot ser que siga necessari utilitzar **callbacks** o **promeses**.