11. PETICIONS HTTP AMB FETCH

IES ESTACIÓ CURS 2021- 2022

Índex

1. ¿Què és Fetch?

- Fetch és el nom d'una nova API per a Javascript amb la qual podem realitzar peticions HTTP asíncrones utilitzant promeses i de manera que el codi siga un poc més senzill.

- La manera de realitzar una petició és molt senzilla, bàsicament es tracta de cridar a fetch i passar-li per paràmetre la URL de la petició a realitzar:

```
// Realizamos la petición y guardamos la promesa
const request = fetch("/robots.txt");

// Si es resuelta, entonces ejecuta esta función...
request.then(function(response) { ... });
```

1. ¿Què és Fetch?

- **fetch()** retornarà una promesa que serà acceptada quan reba una resposta i només serà rebutjada si falla la xarxa o si per alguna raó no es va poder completar la petició.
- La manera més habitual de manejar les promeses és utilitzant .**then().**
- Això se sol reescriure de la següent forma, que queda molt més simple:

```
fetch("/robots.txt")
  .then(function(response) {
    /** Código que procesa la respuesta **/
  });
```

1. ¿Què és Fetch?

- Al mètode **.then()** li passem una funció **callback** on el seu paràmetre **response** és l'objecte de resposta de la petició que hem realitzat.
- En el seu interior realitzarem la lògica que vulguem fer amb la resposta a la nostra petició. A la funció **fetch(url, options)** li passem per paràmetre la **url** de la petició i, de manera opcional, un objecte **options** amb opcions de la petició HTTP.
- Examinarem un codi on es veu un poc millor com fer la petició amb **fetch**:

```
// Opciones de la petición (valores por defecto)
const options = {
  method: "GET"
};

// Petición HTTP
fetch("/robots.txt", options)
  .then(response ⇒ response.text())
  .then(data ⇒ {
    /** Procesar los datos **/
  });
```

2. Paràmetre options.

- Paràmetre opcional **options** de la petició HTTP.
- En aquest objecte podem definir diversos detalls:

Campo	Descripción
STRING `method`	Método HTTP de la petición. Por defecto, `GET`. Otras opciones: `HEAD`, `POST`, etc...
OBJECT `body`	Cuerpo de la petición HTTP. Puede ser de varios tipos: `String`, `FormData`, `Blob`, etc...
OBJECT `headers`	Cabeceras HTTP. Por defecto, `{}`.
STRING `credentials`	Modo de credenciales. Por defecto, `omit`. Otras opciones: `same-origin` e `include`.

2. Paràmetre options.

- El primer, i més habitual, sol ser indicar el mètode HTTP a realitzar en la petició.
- Per defecte, es realitzarà un **GET**, però podem canviar-los a **HEAD, POST, PUT** o qualsevol altre tipus de mètode.
- En segon lloc, podem indicar objectes per a enviar en el bodi de la petició, així com modificar les capçaleres en el camp **headers**:

```
const options = {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify(jsonData)
};
```

2. Paràmetre options.

- Finalment, el camp **credentials** permet modificar la manera en què es realitza la petició.
- Per defecte, el valor omés fa que no s'incloguen credencials en la petició, però és possible indicar els valors **same-origin**, que inclou les credencials si estem sobre el mateix domini, o **include** que inclou les credencials fins i tot en peticions a altres dominis.

- Recorda que estem realitzant peticions relatives al mateix domini. En el cas de realitzar peticions a dominis diferents obtindríem un problema de **CORS** (Cross-Origin Resource Sharing) similar al següent:

> Access to fetch at 'https://otherdomain.com/file.json' from origin 'https://domain.com/' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

3. Headers.

- Encara que en l'exemple anterior hem creat les capçaleres com un objecte genèric de Javascript, és possible crear un objecte Headers amb el qual treballar:

```
const headers = new Headers();
headers.set("Content-Type", "application/json");
headers.set("Content-Encoding", "br");
```

3. Headers.

- Per a això, a part del mètode **.set()** podem utilitzar altres per a treballar amb capçaleres, comprovar la seua existència, obtenir o canviar els valors o fins i tot eliminar-los:

Método	Descripción
BOOLEAN .has(STRING name)	Comprueba si la cabecera name está definida.
STRING .get(STRING name)	Obtiene el valor de la cabecera name.
.set(STRING name, STRING value)	Establece o modifica el valor value a la cabecera name.
.append(STRING name, STRING value)	Añade un nuevo valor value a la cabecera name.
.delete(STRING name)	Elimina la cabecera name.

3. Headers.

- Com molts altres objectes iterables, podem utilitzar els mètodes **.entries(), .keys()** i/o **.values()** per a recórrer-los:

```
for ([key, value] of headers.entries()) {
  console.log("Clave: ", key, "valor: ", value);
}
```

- Per a peticions amb poques capçaleres no és major problema, però en peticions més complexes utilitzar **Headers** és una bona pràctica.

4. Resposta de la petició HTTP.

- Si tornem al nostre exemple de la petició amb **fetch**, observarem que en el primer **.then()** tenim un objecte **response**. Es tracta de la resposta que ens arriba del servidor web al moment de rebre la nostra petició:

```
// Petición HTTP
fetch("/robots.txt", options)
  .then(response ⇒ response.text())
  .then(data ⇒ {
    /** Procesar los datos **/
  });
```

- Encara que en aquest exemple, simplement estem utilitzant una **arrow function** que fa un **return** implícit de la **promesa** que retorna el mètode **.text(),** aquest objecte **response** té una sèrie de propietats i mètodes que poden resultar-nos útils en implementar el nostre codi.

4. Resposta de la petició HTTP.

- Tenim les següents propietats:

Propiedad	Descripción
NUMBER .status	Código de error HTTP de la respuesta (100-599).
STRING .statusText	Texto representativo del código de error HTTP anterior.
BOOLEAN .ok	Devuelve `true` si el código HTTP es `200` (o empieza por `2`).
OBJECT .headers	Cabeceras de la respuesta.
STRING .url	URL de la petición HTTP.

4. Resposta de la petició HTTP.

- Les propietats **.status** i **statusText** ens retornen el codi d'error HTTP de la resposta en format numèric i cadena de text respectivament.
- No obstant això, existeix una novetat respecte a XHR, i és que tenim una propietat **.ok** que ens retorna **true** si el codi d'error de la resposta és un valor del rang **2xx**, és a dir, que tot ha anat correctament.
- Així doncs, tenim una forma pràctica i senzilla de comprovar si tot ha anat bé en realitzar la petició:

```
fetch("/robots.txt")
  .then(response ⇒ {
    if (response.ok)
      return response.text()
  })
```

- Finalment, tenim la propietat **.headers** que ens retorna les capçaleres de la resposta i la propietat **.url** que ens retorna la URL completa de la petició que hem realitzat.

5. Mètodes de processament.

- D'altra banda, la instància **response** també té alguns mètodes interessants, la majoria d'ells per a processar mitjançant una promesa les dades rebudes i facilitar el treball amb ells:

Método	Descripción
STRING `.text()`	Devuelve una promesa con el texto plano de la respuesta.
OBJECT `.json()`	Idem, pero con un objeto `json`. Equivalente a usar `JSON.parse()`.
OBJECT `.blob()`	Idem, pero con un objeto `Blob` (binary large object).
OBJECT `.arrayBuffer()`	Idem, pero con un objeto `ArrayBuffer` (buffer binario puro).
OBJECT `.formData()`	Idem, pero con un objeto `FormData` (datos de formulario).
OBJECT `.clone()`	Crea y devuelve un clon de la instancia en cuestión.
OBJECT `Response.error()`	Devuelve un nuevo objeto `Response` con un error de red asociado.
OBJECT `Response.redirect(url, code)`	Redirige a una `url`, opcionalmente con un `code` de error.

5. Mètodes de processament.

- Observa que en els exemples anteriors hem utilitzat **response.text().**
- Aquest mètode indica que volem processar la resposta com a dades textuals, per la qual cosa aquest mètode retornarà una **promise** amb les dades en text pla, facilitant treballar amb ells de manera manual:

```
fetch("/robots.txt")
  .then(response ⇒ response.text())
  .then(data ⇒ console.log(data));
```

- Observa que en aquest fragment de codi, després de processar la resposta amb **response.text(),** retornem una amb el contingut en text pla.
- Aquesta es processa en el segon **.then(),** on gestionem aquest contingut emmagatzemat en data.

5. Mètodes de processament.

- Tingues en compte que tenim diversos mètodes similars per a processar les respostes. Per exemple, el cas anterior utilitzant el mètode **response.json()** en lloc de **response.text()** seria equivalent al següent fragment:

```
fetch("/contents.json")
  .then(response ⇒ response.text())
  .then(data ⇒ {
    const json = JSON.parse(data);
    console.log(json);
  });
```

- Com es pot veure, amb **response.json()** ens estalviaríem haver de fer el **JSON.parse()** de manera manual, per la qual cosa el codi és un poc més directe.

6. Exemple utilitzant promeses.

- El que veiem a continuació seria un exemple un poc més complet de tot el que hem vist fins ara:
  - Comprovem que la petició és correcta amb **response.ok.**
  - Utilitzem **response.text()** per a processar-la.
  - En el cas de produir-se algun error, llancem excepció amb el codi d'error.
  - Processem les dades i les mostrem en la consola.
  - En el cas que la **promesa** siga rebutjada, capturem l'error amb el **catch**.

6. Exemple utilitzant promeses.

```
// Petición HTTP
fetch("/robots.txt")
  .then(response => {
    if (response.ok)
      return response.text()
    else
      throw new Error(response.status);
  })
  .then(data => {
    console.log("Datos: " + data);
  })
  .catch(err => {
    console.error("ERROR: ", err.message)
  });
```

6. Exemple utilitzant promeses.

- De fet, podem refactoritzar un poc aquest exemple per a fer-lo més llegible.
- Creem la funció **isResponseOk()** per a processar la resposta (invertint el condicional per a fer-ho més directe).
- Després, els **.then()** i **.catch()** els utilitzem amb una **arrow function** per a simplificar-los:

```
const isResponseOk = (response) => {
  if (!response.ok)
    throw new Error(response.status);
  return response.text()
}

fetch("/robots.txt")
  .then(response => isResponseOk(response))
  .then(data => console.log("Datos: ", data))
  .catch(err => console.error("ERROR: ", err.message));
```

- No obstant això, encara que és bastant comú treballar amb promeses utilitzant **.then()**, també podem fer ús de **async/await** per a manejar **promeses**, d'una forma més directa.

7. Exemple utilitzant async/await.

- Utilitzar async/await no és més que usar una cosa visualment més agradable, però que per davall fa la mateixa tasca.
- Per a això, el que hem de tindre sempre present és que un await només es pot executar si està dins d'una funció definida com async.
- En aquest cas, el que fem és el següent:

  - Creem una funció request(url) que definim amb async.
  - Cridem a fetch emprant await per a esperar i resoldre la promesa.
  - Comprovem si tot ha anat bé fent ús de response.ok.
  - Cridem a response.text() fent servir await i retornem el resultat.

7. Exemple utilitzant async/await.

```
const request = async (url) => {
  const response = await fetch(url);
  if (!response.ok)
    throw new Error("WARN", response.status);
  const data = await response.text();
  return data;
}

const resultOk = await request("/robots.txt");
const resultError = await request("/nonExistentFile.txt");
```

- Una vegada fet això, podem cridar a la nostra funció **request** i emmagatzemar el resultat, usant novament await.
- Al final, utilitzar **.then()** o **async/await** és una qüestió de gustos i pots utilitzar el que més t'agrade.