PAU CASESNOVES
Centre Integrat de Formació Professional

Desenvolupament d'aplicacions web

Desenvolupament web a l'entorn client

UT 7.3: Fetch

Joan Pons Tugores

Índex de continguts

Promises....3
Excepcions....4
throw....4
try - catch....5
Fetch....7
Opcions del fetch....9
Recuperar el cos de la resposta....10
Fetch amb async i await....11

Promises

Una promesa és un objecte que torna un valor que esperam rebre en un futur indeterminat, no sabem exactament quan. Són molt adequades per treballar amb peticions asíncrones.

Una promesa té tres estats:

- **Pendent**: Encara no hem rebut la resposta.
- **Completada**: Hem rebut la resposta que esperàvem.
- **Rebutjada**: Hi h hagut problemes i la promesa no ens ha tornat la resposta que esperàvem.

Per tant, en utilitzar una promesa hem de preveure que fer en aquests casos:

- .**then**(funció): funció que s'executa quan la promesa s'ha completat correctament.
- .**catch**(funció): funció que s'executa quan la promesa ha acabat amb error.

Si desbarat() és una funció que s'executa asíncronament i torna una promesa la utilitzaríem de la següent manera:

```
desbarat()
  .then((resultat)=>{})
  .catch((error)=>{})
```

o amb funcions anònimes:

```
desbarat()
  .then (function(resultat){})
  .catch(function(error){})
```

La funció, *callback*, que passam al ***then*** té un paràmetre on rebem el resultat que torna la promesa.

La funció, *callback*, que passam al ***catch*** té un paràmetre on rebem l'error, *exception*, que torna la promesa si falla.

Excepcions

Amb les promeses necessitam les excepcions. Javascript té un sistema d'excepcions semblant al de Java, encara que no igual.

throw

En detectar una errada podem llençar un error amb *throw*.

A diferència de Java, podem llençar qualsevol cosa:

```
throw 'Error2';   // String
throw 42;         // Number
throw true;       // Boolean
throw {toString: function() { return "I'm an object!"; } };
```

try - catch

Si tancam el codi que pot llençar una excepció dins un *try-catch* quan salti una excepció s'executarà el bloc de codi del catch. El catch evidentment no distingeix el tipus de l'excepció.

```
function getMonthName(mo) {
  mo = mo - 1; // Adjust month number for array index (1 = Jan, 12 = Dec)
  let months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
                'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
  if (months[mo]) {
    return months[mo];
  } else {
    throw 'InvalidMonthNo'; // throw keyword is used here
  }
}

try { // statements to try
  monthName = getMonthName(myMonth); // function could throw exception
} catch (e) {
  monthName = 'unknown';
  logMyErrors(e); // pass exception object to error handler (i.e. your own
            function)
}
```

A l'exemple anterior si el més no és correcte la funció *getMonthName* llença una excepció i el codi que l'ha cridada, com que està dins d'un try catch captura aquest error i el tracta.

Podem utilitzar la classe Error. Té un constructor amb un paràmetre per un missatge

```
throw new Error("Missatge de l'error");
```

D'aquesta manera podem utilitzar el missatge:

```
}catch (e){
  alert(e.message);
}
```

La classe Error té una sèrie de subclasses. Per distingir-les dins el *catch* hauríem d'utilitzar *instanceof.* Les classes són:

- EvalError: Un error produït en utilitzar *eval()*.
- RangeError: Error amb un valor numèric que es troba fora del rang vàlid.
- ReferenceError: Hem trobat una referència invàlida.
- SyntaxError: Error de sintàxi.
- TypeError: Una variable o paràmetre no és del tipus esperat.
- URIError: Les funcions *encodeURI()* o *decodeURI()* han rebut arguments invàlids.
- AggregateError: Representa diversos errors agrupats en un de sol.
- InternalError: Errors en el motor Javascript, per exemple "too much recursion"

La propietat *e.name* ens diu de quin tipus és l'error en concret.

```
}catch (e){
  console.log(e.name);
  console.log(e.message);
}
```

Les promeses ja executen el seu mètode *catch* sense tancar-les dins un *try*.

Fetch

Fetch és el substitut del *XmlHttpRequest*. Permet fer crides asíncrones al servidor i torna una promesa.

La versió més bàsica d'un *fetch* seria la següent:

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

El resultat que torna la promesa del fetch és un objecte representant la resposta del servidor.

Com a curiositat, els mètodes que tornen el cos de la resposta, *response.json(), response.formData, response.text()*, ... tornen una altra promesa, per això en aquest exemple tenim dos *then*:

- El primer és per la resposta del *fetch*
- El segon és per la resposta de la promesa del *json()*.

Si volem que les dades tornades per el json arribin al següent *then* la funció ha de fer un return. Recordau que l'exemple anterior és equivalent a

```
fetch('http://example.com/movies.json')
  .then(function(response) {return response.json()})
  .then(function(data){console.log(data)});
```

El fetch només torna un error capturable pel *catch* quan hi ha una errada en la petició. Que el servidor torni un error distint de OK i derivats (tots els errors amb codi entre 200 i 299) no és un error, no arribarà al *catch*.

Podem controlar si la resposta ha estat ok o no amb el *response.ok*

```
fetch('http://example.com/movies.json')
  .then(function(response) {
        if(response.ok){
              return response.json();
        }else{
              throw new Error("Error en el fetch: "+response.status+" - "+
                    response.text);
        }
  })
  .then(function(data){console.log(data)})
.catch(error => alert(error);
```

Opcions del fetch

El primer paràmetre del *fetch* és la url amb la que volem treballar, fins ara és l'únic que hem utilitzat. Per defecte el fetch envia un *get* a la url especificada.

Si volem utilitzar un altre verb, passar informació al body, especificar capçaleres, ... necessitam utilitzar el segon paràmetre del fetch, un objecte amb els paràmetres d'inicialització.

```
fetch('http://example.com/movies.json',
        {
                method:'GET',
                headers:{
                        'Content-Type': 'application/json',
                        'Accept':'application/json'
                },
                body: JSON.stringify(dades)
        }
)
.then(response => response.json())
.then(data => console.log(data));
```

Alguns del atributs d'aquest objecte són:

- **method**: El verb http: GET, POST, PUT, DELETE, ...
- **headers**: Les capçaleres de la petició, qualsevol de les que pugui enviar el navegador: *Content-Type*, *Accept*, *Accept-language*, ...
- **body**: El cos de la petició. El *json*, *xml*, *formdata*, ... que enviau al servidor.

Recuperar el cos de la resposta

L'objecte *response* té diversos mètodes per recuperar el cos de la resposta, depenent del tipus de les dades que conté. Tots aquests mètodes tornen una promesa amb el contingut.

- .json(): Torna el resultat d'aplicar JSON.parse al contingut del cos. Per tant torna un objecte o array Javascript.
- .text(): Torna un String amb el contingut del cos. Sempre es decodifica utilitzant UTF-8.
- .formData(): Torna el contingut del cos com un objecte FormData. El tipus mime és "multipart/form-data"
- ...

Fetch amb async i await

await és una paraula reservada de Javascript que permet aturar l'execució del codi esperant que una funció asíncrona, per exemple un *fetch*, acabi. Permet programar el codi d'una manera més "seqüencial" que utilitzant el *then.*

await atura el codi. Per minimitzar-ne els efectes s'ha d'incloure el codi que l'utilitza dins una funció marcada amb la paraula clau *async*.

Un exemple de petició Rest amb await podria ser el següent:

```
async function ambAwait() {
    const resposta = await fetch("http://52.178.39.51:8080/llibres", {
        headers: {accept: 'application/json'},
        method: 'get'
    });
    if (resposta.ok) {
        const llibres = await resposta.json();
        llibres.forEach(llibre => {
            const p = document.createElement("p");
            p.appendChild(document.createTextNode(llibre.titol));
            document.getElementById("llibres").appendChild(p);
        });
    }
}
```

En aquest cas tenim una funció *async*, és a dir, que la seva execució serà asíncrona,que fa una petició Rest amb un fetch, però aquesta vegada el decoram amb *await*.

```
async function ambAwait() {
```

En aquest cas marcam el fetch amb *await* i assignam la resposta que torna, la promise, a la constant *resposta*.

```
const resposta = await fetch("http://52.178.39.51:8080/llibres", {
    headers: {accept: 'application/json'},
    method: 'get'
});
```

Si la resposta ha arribat amb un codi Ok hem de recuperar el contingut amb el mètode *json()*, que torna una altra promesa. Per tant, tornarem a assignar el resultat de *json()* a una constant marcant-lo amb await.

```
if (resposta.ok) {
    const llibres = await resposta.json();
```

Quan arriba la promesa, dins *llibres* tendrem les dades Javascript i podrem fer el tractament que faci falta.

```
    llibres.forEach(llibre => {
        const p = document.createElement("p");
        p.appendChild(document.createTextNode(llibre.titol));
        document.getElementById("llibres").appendChild(p);
    });
```

El mateix codi sense async i await seria:

```
function senseAwait() {
    fetch("http://52.178.39.51:8080/llibres",
            {headers: {accept: 'application/json'}, method: 'get'})
        .then(response => response.json())
        .then(llibres =>
            llibres.forEach(llibre => {
                const p = document.createElement("p");
                p.appendChild(document.createTextNode(llibre.titol + " (" +
                        llibre.idLlib + ")"));
                document.getElementById("llibres").appendChild(p);
            })
        );
```