Comunicació amb el servidor

Índex

- Ajax
- Execution Context i Call Stack
- Event Loop, pila i cua
- XMLHttprequest:
  - Json
  - Formularis
  - XHR Més fàcil
- APIs
- Promeses
  - XHR en promeses
- Fetch

AJAX

- Un conjunt de tecnologies:
  - Javascript
  - XHTML i CSS
  - XML o JSON
  - XMLHttpRequest
- Amb XMLHttpRequest, Javascript demana o envía un XML o un JSON al servidor sense recarregar la pàgina.
- Javascript utilitza el XML o JSON rebut del servidor per a modificar la web.
- El servidor sol tindre un API Rest o similar per a atendre les peticions.
- Millora el rendiment i l'experiència d'usuari al no recarregar tot.
- Empitjora el SEO i és més difícil de desenvolupar.
- Amb AJAX es poden fer webs SPA.

El motor de Javascript

Dins del navegador s'encarrega de:

- Compilar i executar el codi Javascript
- Manejar la pila de funcions. (call stack)
- Manejar l'allotjament dels objectes en memòria (heap)
- Recolector de fem per als objectes que ja no es necessiten.
- Proporcionar una API amb utilitats del navegador, xarxa, asíncrones entre altres.

Entorn d’execució

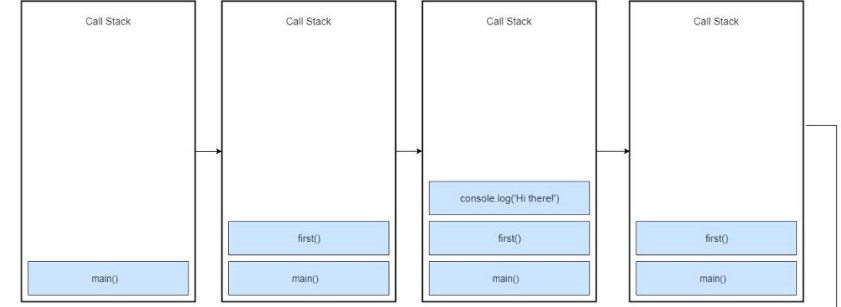
- Javascript sols pot tindre un fil d’execució. (en principi)
- Si demanem alguna cosa al servidor de forma síncrona, tota la web es para fins que arriba.
- Els navegadors tenen un entorn d’execució (runtime enviroment) que permeten que Javascript demane de forma asíncrona i continue fent coses.
- Eixes peticions asíncrones poden ser Callbacks, Promeses o Asinc/Await
- Per entendre cóm funcionen les peticions asíncrones en JS cal entendre els conceptes de Context d’execució i de Pila de Cridades (Execution Context and Call Stack)

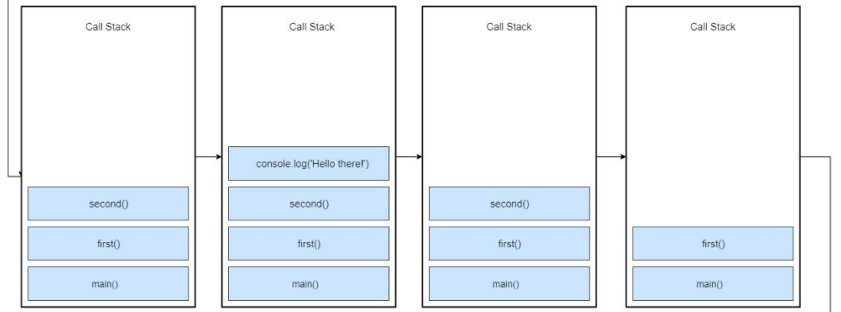
Execution Context i Call Stack

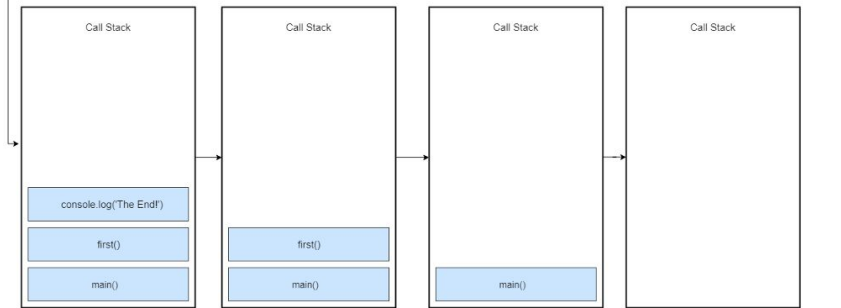
- Execution Context
  - És l'entorn en el qual JS s'avalúa i executa.
  - Pot ser global i cada funció té el seu.
- Call Stack
  - Una pila amb estructura LIFO on anar posant els contexts d'execució.

Exemple de Call stack

```
const second = () => {
  console.log('Hello there!');
}
const first = () => {
  console.log('Hi there!');
  second();
  console.log('The End');
}
first();
```

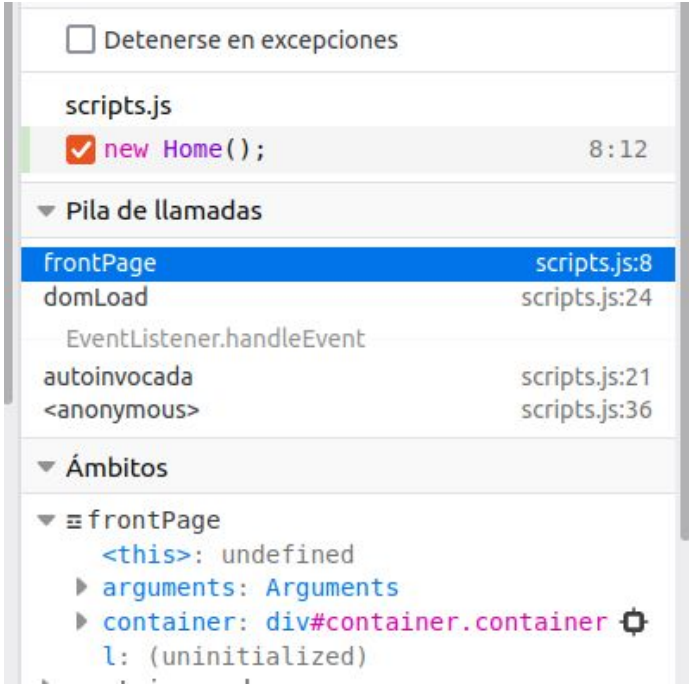






Analitzar Call Stack

- En firefox:
  - F12
  - Depurador
  - Ficar punt de ruptura
  - Executar i analitzar la pila de cridades i l'entorn d'execució de les funcions.
- console.trace()



Web APIs

- Els navegadors tenen uns serveis asíncrons que pot demanar Javascript.
- Alguns exemples són setTimeOut, Esdeveniments del DOM o XMLHttpRequest
- Aquestes APIs no són específiques de Javascript, sino dels navegadors (Node.js les fa d'una altra manera)

```
const networkRequest = () => {
  setTimeout(() => {
    console.log('Async Code');
  }, 2000);
 };
console.log('Hello World');
networkRequest();
console.log('The End');
```

Event Loop, pila i cua


```
// La segona funció té codi asíncron
function first() {console.log(1)}
function second() {
      setTimeout(() => {
         console.log(2)}, 0)} // 0 segons
function third() {console.log(3)}
first();
second();
third();
// Sempre eixirà 1 3 2
```

- Afegir first() a la pila, imprimir 1 i llevar first de la pila.
- Afegir second() a la pila, afegir setTimeout() a la pila, afegir la funció fletxa a la cua, llevar setTimeout() i second() de la pila.
- Afegir third() a la pila, imprimir 3 i llevar third() de la pila.
- Recorrer la cua, afegir la funció fletxa a la pila, imprimir 2 i llevar la funció fletxa de la pila.
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

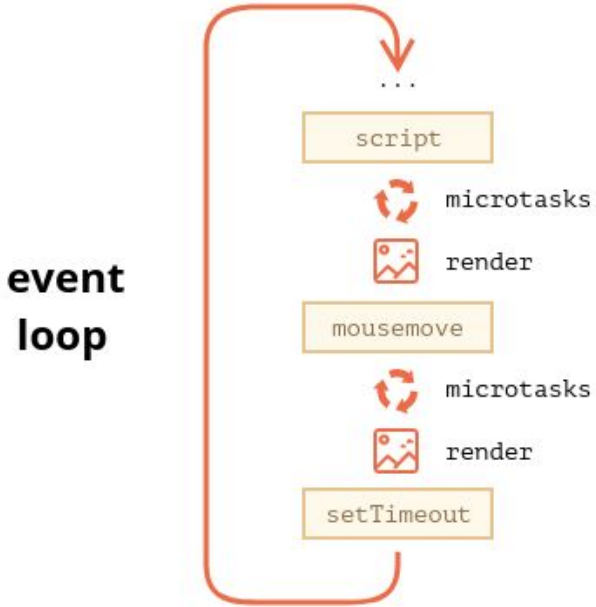
Callback

- Si volem que s'execute una funció després de que es complete una operació asíncrona es pot fer en callbacks
- Són funcions passades com a argument a altres funcions.
- Es pot produir un "Callback Hell" o "Pyramid of Doom"

```
// La segona funció tè codi asíncron
function first() {console.log(1)}
function second(callback) {setTimeout(() => {console.log(2); callback(); }, 0)}
function third() {console.log(3)}
first();
second(third);
// 1 2 3
```

Tasques i Microtasques

- Javascript no gestiona una única cua de tasques. Algunes cues tenen més prioritat i es gestionen diferent:
- La cua de les promeses es diu “microtask queue”
- Els esdeveniments i setTimeOut van a la “macrotask queue”
- Després de qualsevol macrotasca, es fan totes les microtasques.



Aprofitar les tasques


```
let start = Date.now();

function count() {
 // do a heavy job
 for (let j = 0; j < 1e9; j++) {
   i++;
 }
 console.log("Done in " + (Date.now() - start) + 'ms');
}

// count();   // Aquest bloqueja el navegador
setTimeout(count,0);
```

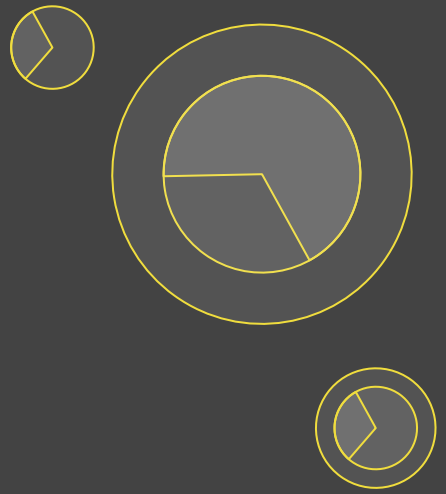
Fer una barra de progrés


```
document.addEventListener("DOMContentLoaded", () => {
 let progress = document.querySelector("#progress");
 let i = 0;
 function count() {
   // do a piece of the heavy job (*)
   do {
     i++;
     progress.innerHTML = i;
   } while (i % 1e3 != 0);

   if (i < 1e7) {      setTimeout(count);    }
 }
 count();
});
```

S'ha de dividir en varis per a que puga renderitzar en mig

XMLHttpRequest



XMLHttpRequest

- Fa les peticions asíncrones i ens avisa quan estan fetes.
- A pesar del seu nom, pot acceptar qualsevol tipus de dades. (Nosaltre utilitzarem més JSON)
- Té molts atributs i mètodes que anirem utilitzant poc a poc.
- Ara necessitem un servidor HTTP per a fer les peticions. Com que el servidor no estarà en el mateix domini que la web de proves, cal desactivar la seguretat de cross-origin al servidor.

Iniciar XMLHttpRequest


```
var req = new XMLHttpRequest();
req.open('GET', 'http://www.mozilla.org/', true);
req.onreadystatechange = function (aEvt) {
if (req.readyState == 4) {
   if(req.status == 200)
    dump(req.responseText);
   else
    dump("Error loading page\n");
}
};
req.send(null);
```

Inicialitzar l'objecte en navegadors moderns.

Indiquem que volem que siga asíncrona amb el true

Quan canvie l'estat a 4 mirem si ha arribat i el mostrem.

Mètodes i atributs XMLHttpRequest

- Open(mètode,URL,true): Diu quina URL i si s'ha de comunicar en GET o POST
- Send(dades): Diu quines dades s'han d'enviar. S'ha de fer sempre després del open().
- setRequestHeader(): Diu el format de les capçaleres.
- readyState: Informa al programa de l'estat de les peticions:
    - 0 Objecte iniciat
    - 1 Oberta la connexió (open)
    - 2 Al fer una petició (send)
    - 3 Rebent informació del servidor.
    - 4 Petició completada

FormData

- Objecte predefinit de Javascript per crear parells clau-valor per enviar formularis per XMLHttpRequest.

```
var formElement = document.getElementById("myFormElement"); // Un formulari html
let formData = new FormData(formElement); // Constructor de formData amb un formulari
formData.append("serialnumber", serialNumber++); // Afegir més dades
formData.append("afile", fileInputElement.files[0]); // afegir un fitxer

var xhr = new XMLHttpRequest();
xhr.open("POST", "http://foo.com/submitform.php");
xhr.send(formData);  // Enviar el formulari per POST
```

XMLHttpRequest i POST JSON

https://attacomsian.com/blog/xhr-json-post-request

Cal afegir al xhr açò:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

Si volem enviar dirèctament no cla fer res més. Si volem convertir abans en JSON:

```
JSON.stringify(Object.fromEntries(formData));
```

XHR més fàcil


```
function makeRequest (method, url, done) {
 var xhr = new XMLHttpRequest();
 xhr.open(method, url);
 xhr.onload = function () {    done(null, xhr.response); };
 xhr.onerror = function () {    done(xhr.response); };
 xhr.send();
}

makeRequest('GET', 'http://example.com', function (err, datums) {
 if (err) { throw err; }
 console.log(datums);
});
```

Els navegadors actuals suporten onload()

Encara millor si ho fem en addEventListener

APIs

- La comunicació amb el servidor pot ser:
  - JS demana un HTML estàtic o dinàmic i inserta el resultat.
  - JS demana o envía les dades en XML o JSON a una API.
- Les APIs poden ser:
  - SOAP (Complicat i no optimitzat per a HTTP)
  - REST (Basat en HTTP i en les URL)
  - Grapql (Com REST Amb més possibilitats per a les consultes)
  - gRPC (Més nou i per a HTTP/2)

API REST

- Utilitza les peticions HTTP com a verbs del protocol: GET, PUT, DELETE, POST, PATCH
- Utilitza les rutes de la URL per als elements a consultar o modificar.
- Utilitza el números de resposta HTTP per veure si ha funcionat.
- Les dades (Payload) es poden enviar en XML o JSON.
- Pot ser RESTfull si és totalment estricta en les característiques REST.

API GraphQL

- En la URL envía un JSON amb la consulta a realitzar.
- Permet més control en les peticions i una major granularitat.
- Una vegada implementat, les peticions són fàcils d'entendre pels humans.
- No sols funciona en HTTP.
- Utilitza el IDL Schema Definition Language

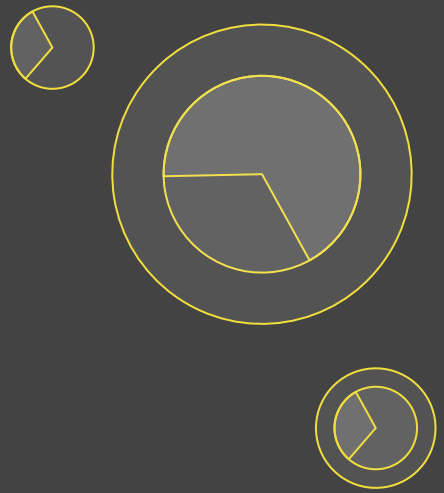
SDKs

- Les APIs poden ser molt complexes.
- Ferramentes com Firebase, MongoDB Realm, Supabase... tenen ferramentes per autenticar usuaris o fer consultes avançades.
- Encara que publiquen en REST o GraphQL i publiquen els protocols, programar la comunicació cada vegada pot ser costós.
- Aquestes bases de dades com a Backend solen tindre un SDK (Software Development Kit) que són biblioteques que simplifiquen les tasques comuns.

Els SDK no són estàndards i dependen del proveidor. Per a ser més genèrics i aprendre la base anem a evitar-los en classe.

Esperar peticions asíncrones

- XMLHttpRequest té distints estats i es pot assignar una funció al canvi d'estat.
- Mentre arriba la informació, JS pot mostrar un gif o un buit on anirà.
- La càrrega remota asíncrona no para el funcionament de l'aplicació.
- La forma tradicional de gestionar això és amb Callbacks, però pot implicar problemes.
- JQuery i altres biblioteques tenen ferramentes per simplificar la gestió d'aquests esdeveniments.
- JS a partir de ES6 incorpora les promeses de forma nativa.



Promises

https://developers.google.com/web/fundamentals/primers/promises?hl=es
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar_promesas
https://developer.mozilla.org/es/docs/Web/JavaScript/EventLoop#Ejecutar_hasta_completar
https://www.digitalocean.com/community/tutorials/understanding-the-event-loop-callbacks-promises-and-async-await-in-javascript
https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/async%20%26%20performance/ch3.md

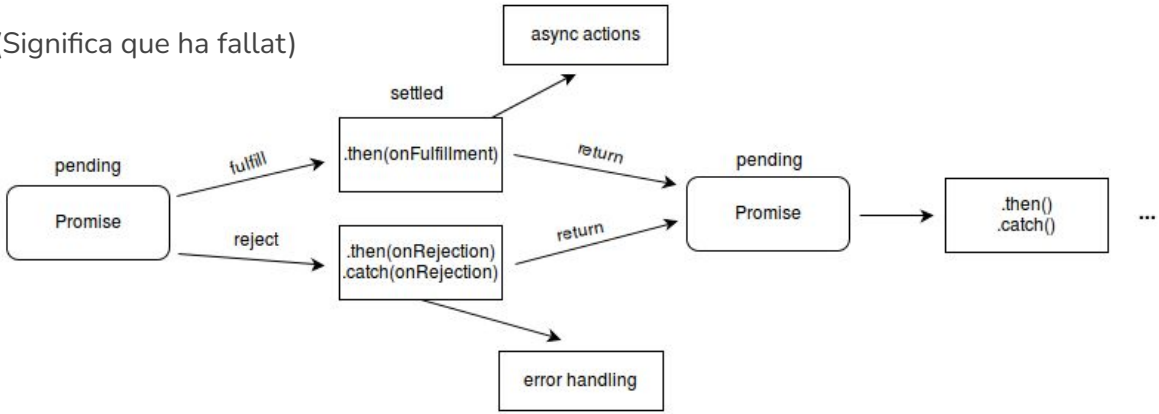
Promises

- Objectes que representen a un valor que pot estar disponible ara, en el futur, o mai.
- Tenen una funció executor que accepta una funció resolve i una reject.

```
const promise = new Promise((resolve, reject) => { // Funció executor
  setTimeout(() => {
    if (Math.random() > 0.5) { resolve("Resolving an asynchronous request!"); }
    else { reject("Rejecting an asynchronous request!"); }
  }, 2000);
 });
  promise.then((response) => { //.then si resol
    console.log(response);
  }).catch((response) => { // .catch si falla
    console.log(response);
  });
```

Promises

- Tornen un objecte de forma síncrona amb el que es pot treballar.
- Aquest objecte és la promesa d'un objecte futur, ja siga exitosa o no la promesa.
- Permet llançar peticions asíncrones i no esperar al resultat per poder continuar.
- Les promeses poden estar en aquests estats:
  - Pendent ***Pending*** (Estat inicial, que encara no s'ha complit o rebutjada)
  - Complida ***fullfilled***
  - Rebutjada ***rejected*** (Significa que ha fallat)



Encadenar promeses

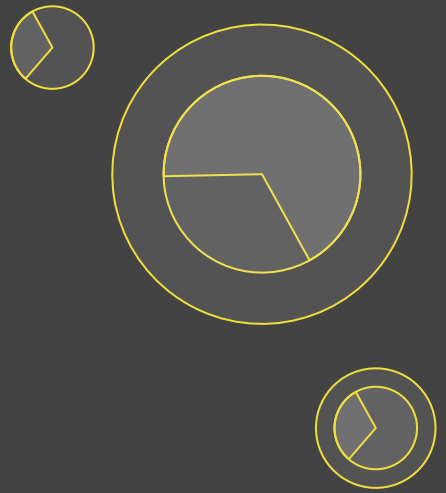
- .then() es pot encadenar i garantitza que s'executarà seqüencialment encara que les funcions siguen asíncrones.
- Es poden fer moltes peticions asíncrones o no i ficar un then() al final amb Promise.all()
- Es poden encadenar .then() a .catch() per a fer coses funcione o no la promesa.

Convertir XHR en promeses


```
function makeRequest (method, url) {
 return new Promise(function (resolve, reject) {
   var xhr = new XMLHttpRequest();
   xhr.open(method, url);
   xhr.onload = function () {
     if (this.status >= 200 && this.status < 300) {
       resolve(xhr.response);
     } else {
       reject({ status: this.status, statusText: xhr.statusText });
     }
   };
   xhr.onerror = function () {
     reject({ status: this.status, statusText: xhr.statusText });
   };
   xhr.send();
 }); }
```

```
makeRequest('GET', 'http://example.com')
.then(function (datums) {
 console.log(datums);
})
.catch(function (err) {
 console.error('Augh, there was an error!',
err.statusText);
});
```

Fetch



fetch

- Funciona paregut a XHR però amb promeses i sintaxi més fàcil.

```
fetch('http://127.0.0.1:5500/datos.json')
 .then(
   function(response) {
     if (response.status !== 200) {
       console.log('Looks like there was a problem. Status Code: ' +
         response.status);
       return;      }
     response.json().then(function(data) {
       console.log(data);
     });    } )
 .catch(function(err) {
   console.log('Fetch Error :-S', err);
 });
```

Response

- Si funciona, retorna una resposta que és un objecte ‘stream’ que té algunes funcions i atributs útils:
  - response.status -> L’estat de la descàrrega.
  - response.json() -> Transforma la resposta json en un objecte.

```
fetch('users.json').then(function(response) {
   console.log(response.headers.get('Content-Type'));
   console.log(response.headers.get('Date'));

   console.log(response.status);
   console.log(response.statusText);
   console.log(response.type);
   console.log(response.url);
});
```

Guardar les dades

- Fetch permet, entre altres coses, obtindre un text o un objecte de la resposta.
- L'objecte response té dos funcions que retornen la promesa d'obtenir un text o un objecte si la resposta és un json.
- No es poden fer les dos coses d'una mateixa petició.

```
fetch("https://dwec-daw-default-rtdb.firebaseio.com/productos.json")
       .then(response => response.json())
       .then(data => console.log(data));

fetch("https://dwec-daw-default-rtdb.firebaseio.com/productos.json")
       .then(response => response.text())
       .then(data => console.log(data));
```

Encadenant promeses


```
function status(response) {
   if (response.status >= 200 && response.status < 300) {
     return Promise.resolve(response)
   } else {
     return Promise.reject(new Error(response.statusText))
   }
 }
function json(response) { return response.json()  }
fetch('datos.json')
   .then(status)
   .then(json)
   .then(function(data) {
     console.log('Request succeeded with JSON response', data);
   }).catch(function(error) {
     console.log('Request failed', error);
   }); https://stackoverflow.com/a/43082995
```

POST en fetch


```
fetch(url, {
    method: 'post',
    headers: {
      "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
    },
    body: 'foo=bar&lorem=ipsum'
  })
  .then(json)
  .then(function (data) {
    console.log('Request succeeded with JSON response', data);
  })
  .catch(function (error) {
    console.log('Request failed', error);
  });
```

POST JSON en fetch


```
let datos = {username: 'example'};
fetch(url, {
        method: 'post',
        headers: {
          "Content-type": "application/json; charset=UTF-8"
        },
        body: JSON.stringify(datos)
      })
      .then(json)
      .then(function (data) {
        console.log('Request succeeded with JSON response', data);
      })
      .catch(function (error) {
        console.log('Request failed', error);      });
```

Async Await

- Una manera de treballar en promeses més curta.
- Una funció por ser async i retorna una promesa, per tant, es pot encadenar un .then()
- Dins d'una funció async, es poden cridar promeses en await i espera a que acabe per continuar.

```
async function getUser() {
 const response = await fetch('https://api.github.com/users/octocat ')
 const data = await response.json()
 console.log(data)
}
// Execute async function
getUser()
```

Carregar imatges en segon pla


```
<img src="placeholder.png" alt="${name}">

fetch(image_url)
.then(response => response.status == 200 ? response :
Promise.reject(response.status))
.then(response => response.blob())
.then(imageBlob => {
   let imageURL = URL.createObjectURL(imageBlob);

   divCard.querySelector('img').src = URL.createObjectURL(imageBlob);
}).catch(error => console.log(error));
```

Tractament de les dades en Javascript

Passant dades a Json


```
class Apple {
  constructor(type){
     this.type = type;
     this.color = "red";
  }
}
var apple1 = new Apple('Golden'); // Es crea una instància
appleJson=JSON.stringify(apple1);
console.log(appleJson);
```

Convertint de JSON a objectes


```
class Hero{
  constructor(name,car){
     this.name = name; this.car=car;
  }}
let heroJSON = '{"name":"Max","car":"V8"}';
let heroObject = JSON.parse(heroJSON);
let heroClass = Object.assign(new Hero, heroObject);
console.log(heroObject,heroClass);
```

Emmagatzemament en el costat del client

- Javascript no té accés directe ni al sistema d'arxius del client ni del servidor.
- Per poder guardar informació, els navegadors proporcionen ferramentes a Javascript.
- Les principals són les **Cookies i LocalStorage**

Cookies

- Informació associada a un domini web que guarda el navegador.
- Es guarden en un fitxer de text i Javascript té comandaments específics per a manipular-les.
- Es guarden variables i el seu valor.
- Les Cookies sols es poden modificar desde el domini que les ha creades.
- Les Cookies són les que, per exemple, mantenen una sessió o ajuden a la publicitat contextual.
- Es poden veure les cookies amb F12

```
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2021 12:00:00 UTC; path=/";
```

Manipular Cookies


```
var x = document.cookie;  // Llegir totes les cookies
// Modificar una cookie és sobreescriure
document.cookie = "username=John Smith; expires=Thu, 18 Dec 2021 12:00:00 UTC; path=/";
// Esborrar és fer que estiga expirada
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
```

- Quan hi ha moltes cookies, les funcions bàsiques es queden curtes.
- Es recomana utilitzar les funcions de la W3C: https://www.w3schools.com/js/js_cookies.asp

LocalStorage

- En els navegadors moderns, donen fins a 5MB per a guardar informació.
- Diferenciem entre LocalStorage (Permanent) i SessionStorage (Mentre el navegador estiga obert)

```
// Guardar
localStorage.setItem("lastname", "Smith");
// Obtenir
var lastname = localStorage.getItem("lastname");
// Esborrar
localStorage.removeItem("lastname");
```

indexedDB

- Fins a 50MB
- API de baix nivell **asíncrona**
- Permet guardar arxius i té un indexat més avançat.
- Internament és una base de dades transaccional.
- API més complicada.