

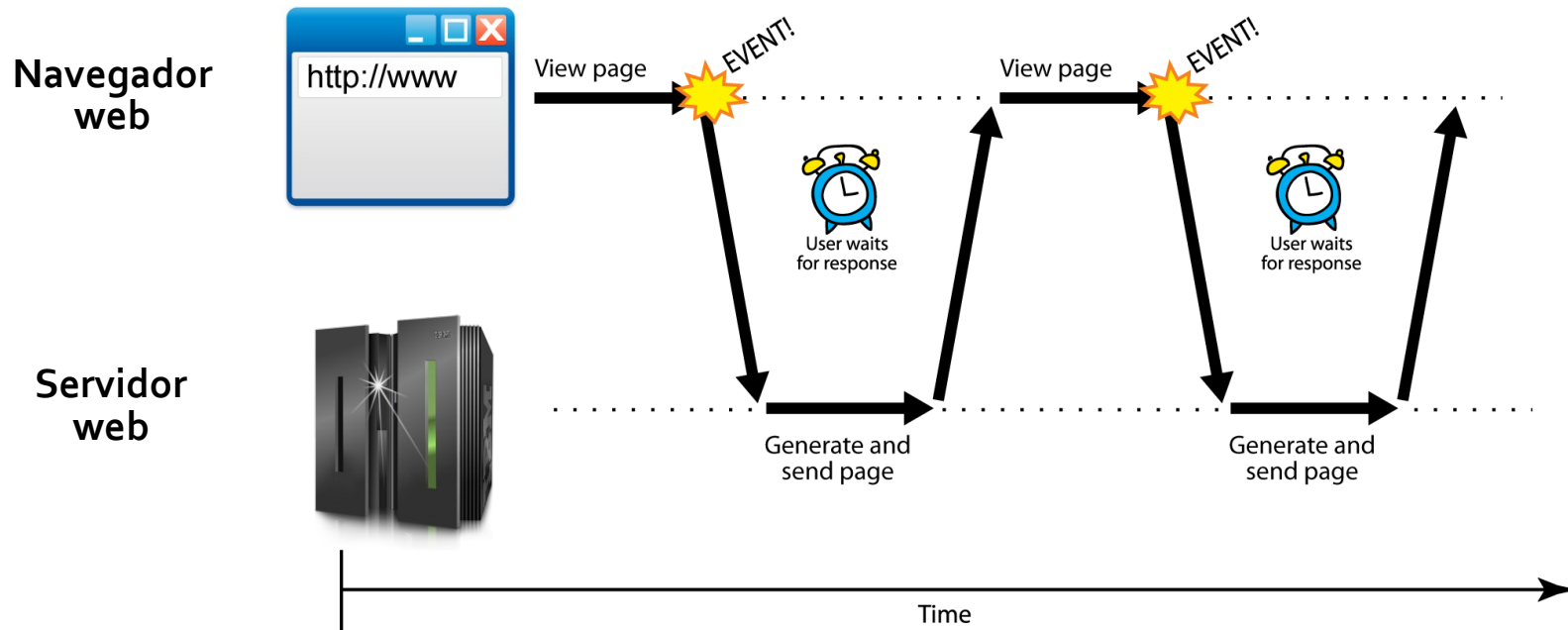
Fundamentos de la Web

Bloque II: Tecnologías de cliente web

Tema 3.10 – AJAX

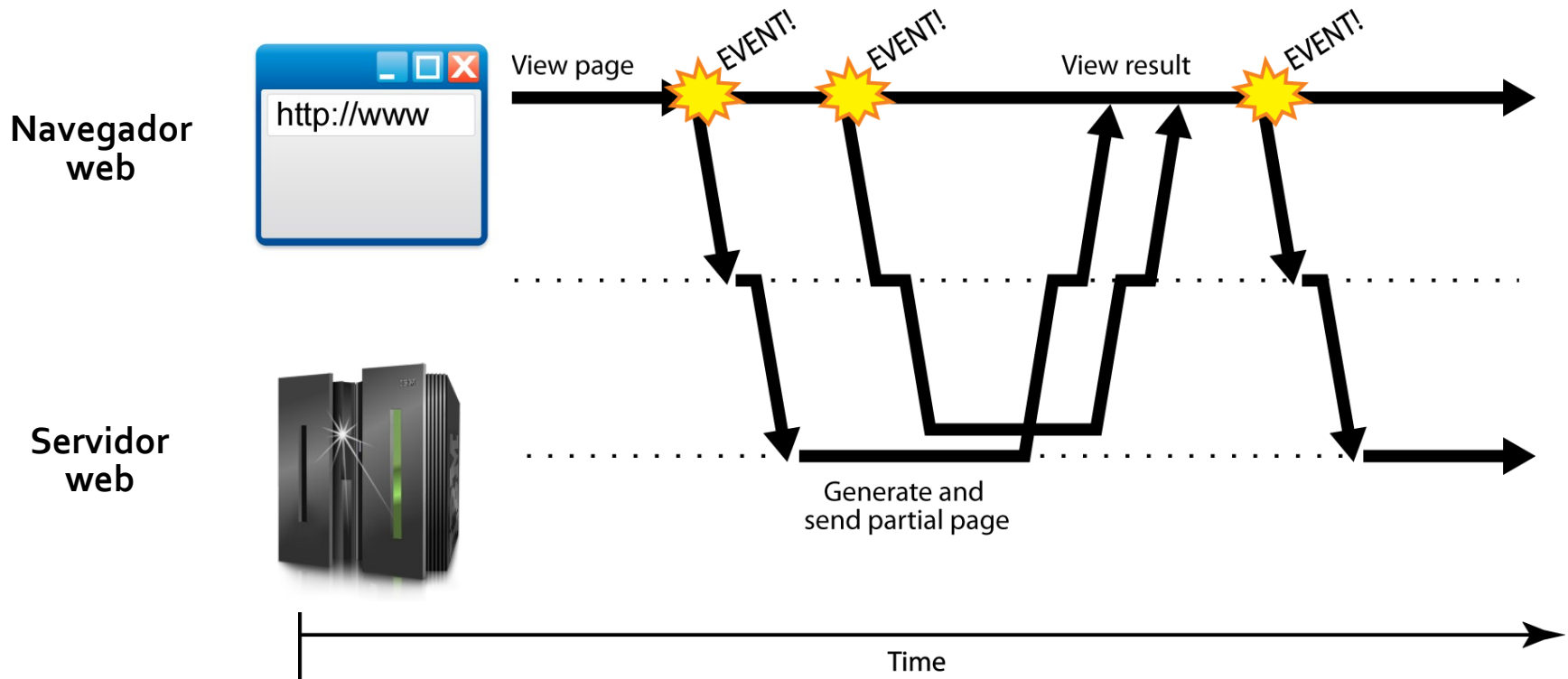
Introducción

- En una página web básica, cada vez que el usuario **pulsa un enlace**, en navegador **recarga completamente la página**

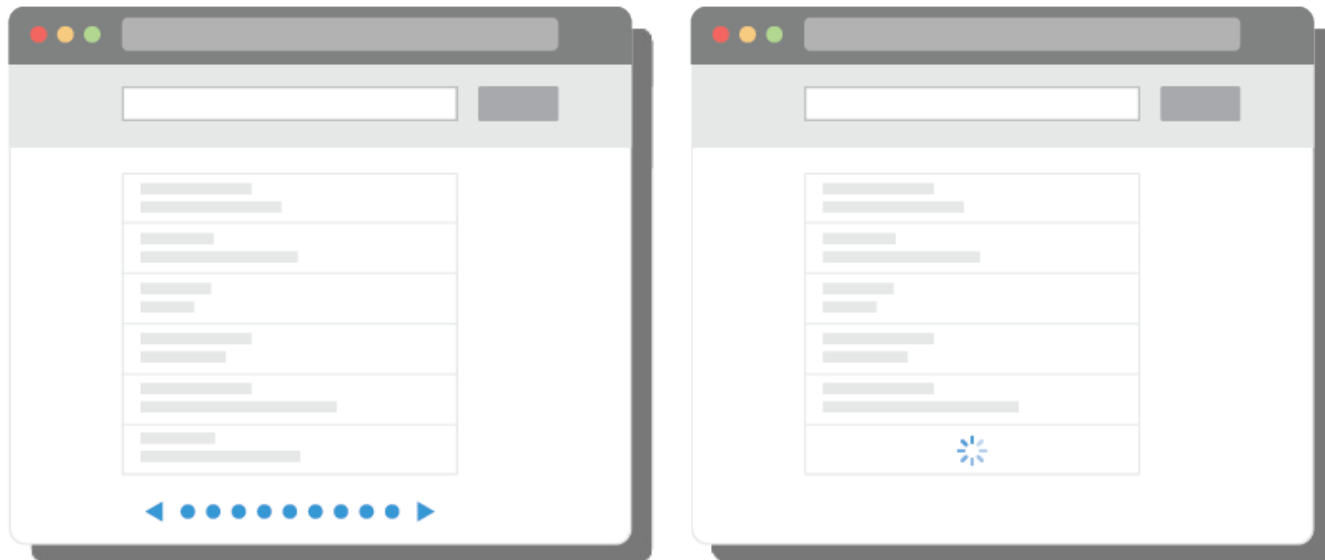


- A veces sólo cambia **parte de la página**, pero se tiene que recargar completa (más **transferencia de datos**, mayor **tiempo de espera...**)
- **AJAX** (*Asynchronous JavaScript And XML*) es una técnica que permite actualizar únicamente parte de la página
- Desde código JavaScript que **solicita al servidor la parte nueva y actualiza** la página con el contenido obtenido del servidor

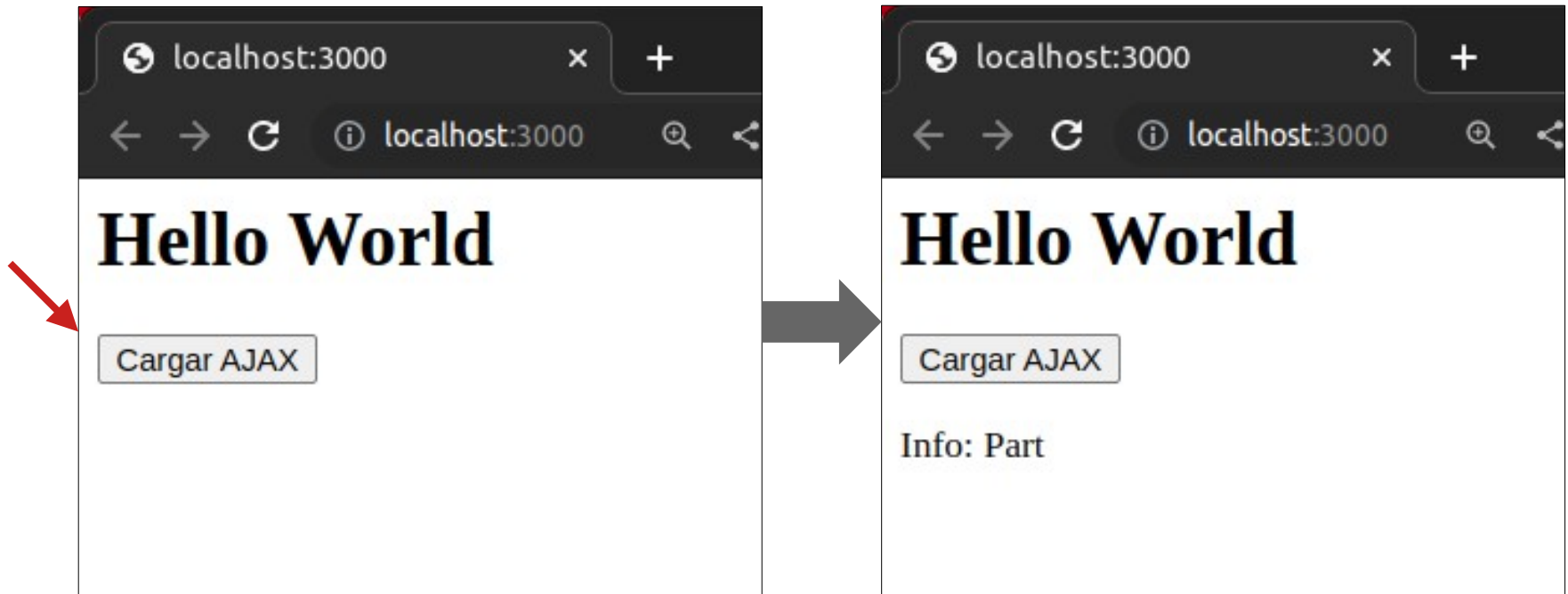
- *AJAX (Asynchronous JavaScript And XML)*



- Es la técnica que se usa para **cargar más resultados** en las búsquedas sin recargar la página (botón “**Cargar más**” o de forma **automática**)



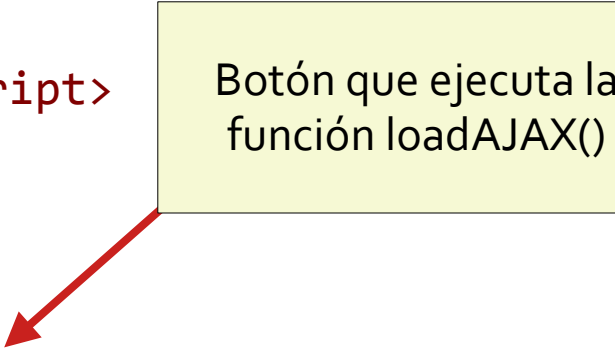
Implementación de AJAX



Implementación en cliente

- Petición AJAX desde el navegador web

```
<html>
<head>
  <script src="app.js"></script>
</head>
<body>
  <h1>Hello {{name}}</h1>
  <button onclick="loadAJAX()">Cargar AJAX</button>
  <div id="content"></div>
</body>
</html>
```



Botón que ejecuta la función loadAJAX()

Implementación en cliente

- Petición AJAX desde el navegador web

```
async function loadAJAX(){  
  
    const response = await fetch('/pagePart');  
    const pagePart = await response.text();  
  
    const content = document.getElementById("content");  
    content.innerHTML = pagePart;  
  
}
```


Implementación en cliente

ejem1

- Petición AJAX desde el navegador web

Hace la petición al servidor de la ruta **/pagePart** en segundo plano y devuelve una respuesta (response)

```
async function loadAJAX(){  
  
    const response = await fetch('/pagePart');  
    const pagePart = await response.text();  
  
    const content = document.getElementById("content");  
    content.innerHTML = pagePart;  
  
}
```

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Implementación en cliente

ejem1

- Petición AJAX desde el navegador web

```
async function loadAJAX(){
```


```
    const response = await fetch('/pagePart');
```

```
    const pagePart = await response.text();
```

```
    const content = document.getElementById("content");  
    content.innerHTML = pagePart;
```

```
}
```

Procesa la respuesta y
obtiene el **texto** que
contiene



<https://developer.mozilla.org/en-US/docs/Web/API/Response/text>


Implementación en cliente

ejem1

- Petición AJAX desde el navegador web

```
async function loadAJAX(){  
  
    const response = await fetch('/pagePart');  
    const pagePart = await response.text();  
  
    const content = document.getElementById("content");  
    content.innerHTML = pagePart;  
  
}
```

Se inserta el
contenido cargado
en la página



Implementación en cliente

- Petición AJAX desde el navegador web

```
async function loadAJAX(){  
  const response = await fetch('/pagePart');  
  const pagePart = await response.text();  
  
  const content = document.getElementById("content");  
  content.innerHTML = pagePart;  
  
}
```

Más adelante veremos qué significa **async** y **await**

Implementación en servidor

ejem1

- Generación de contenido AJAX en el servidor

router.js

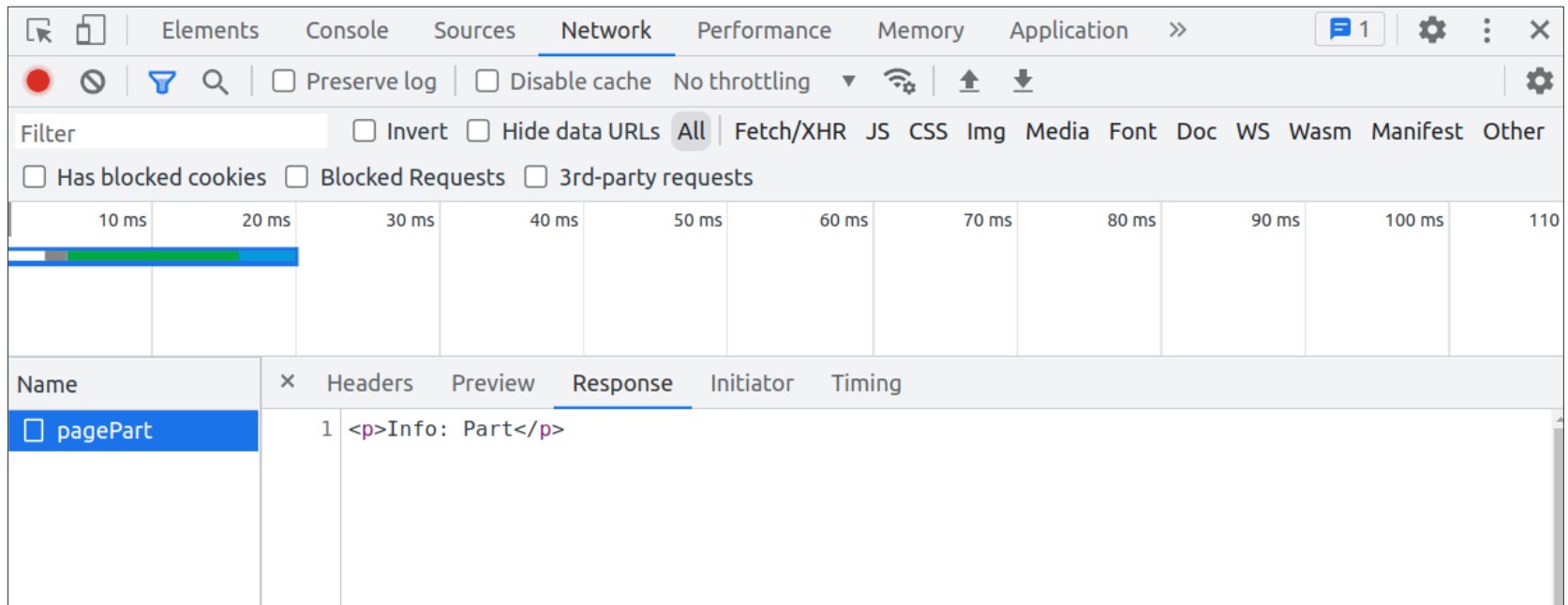
```
import express from 'express';
const router = express.Router();
router.get('/', (req, res) => {
  res.render('index', {
    name: "World"
  });
});
router.get('/pagePart', (req, res) => {
  res.render('pagePart', {
    info: "Part"
  });
});
export default router;
```

pagePart.mustache

<p>Info: {{info}}</p>

Se genera el
contenido igual
que las páginas
completas

- La pestaña “Network/Red” del navegador permite **ver las peticiones** realizadas por AJAX



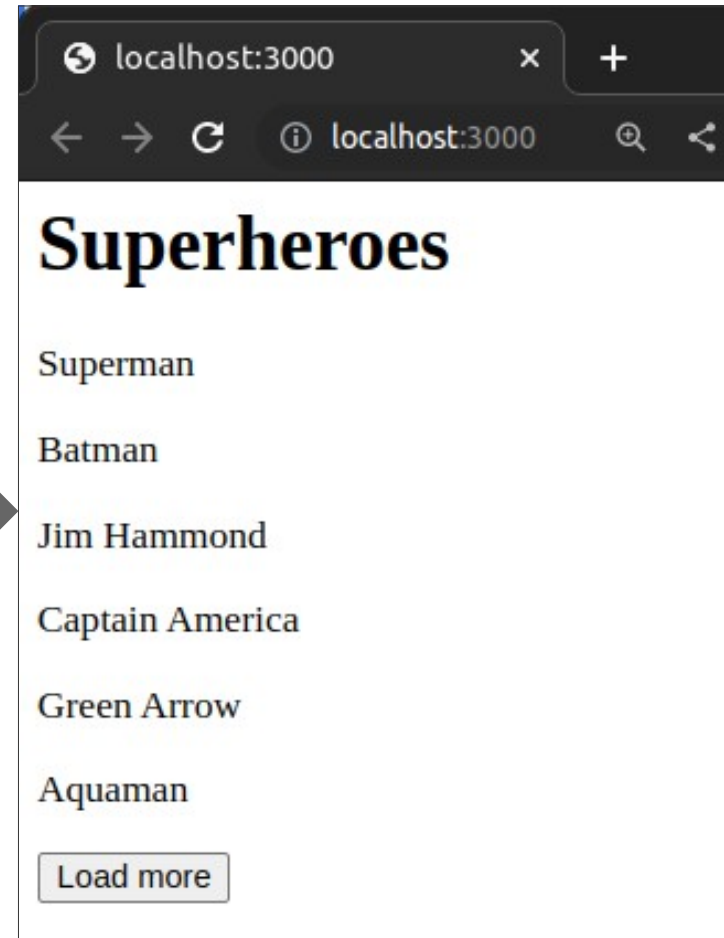
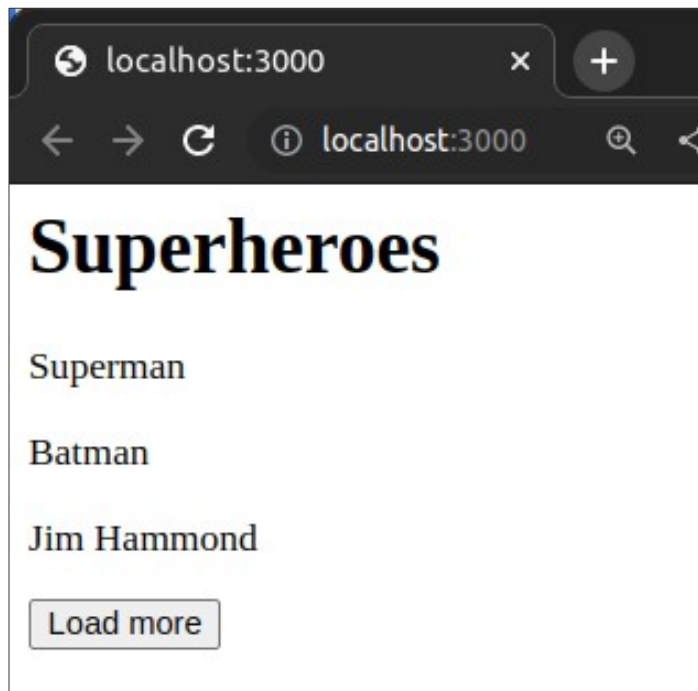
Ejercicio 1

ejer1

- Se tiene una web que muestra los libros guardados en un mapa
- Se quiere implementar un botón de “Cargar más” con AJAX
- Los libros se mostrarán de 3 en 3

Ejercicio 1

ejer1



Ejercicio 1

ejer1

- Al cargar la página se mostrarán los libros 0,1 y 2.
- Cuando se pulse el botón la primera vez se cargarán los libros 3, 4 y 5.
- Cuando se pulse el botón la segunda vez se cargarán los libros 6, 7 y 8.
- La URL que se usa para cargar más resultados deberá indicar qué resultados cargar

Ejercicio 1

ejer1

- Se proporciona una web en la que se muestran inicialmente todos los libros

```
import express from 'express';
import { getSuperheroes } from './superheroes.js';

const router = express.Router();

router.get('/', (req, res) => {
  const superheroes = getSuperheroes();

  res.render('index', {
    superheroes: superheroes
  });
});

export default router;
```

Ejercicio 1

ejer1

- Se proporciona una web en la que se muestran inicialmente todos los libros

```
const superheroes = new Map();
let id = 0;

export function addSuperhero(superhero) {
  superheroes.set(id, superhero);
  superhero.id = id;
  id++;
}

export function getSuperhero(id) {
  return superheroes.get(id);
}
```

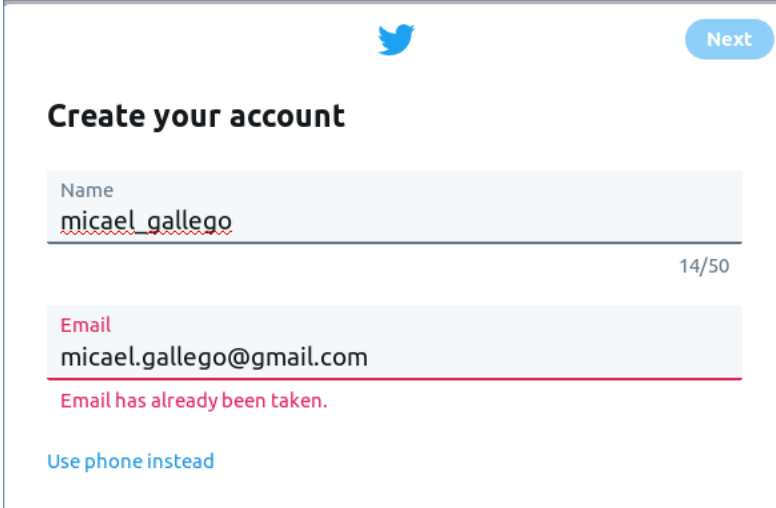
```
export function getSuperheroes(from, to) {
  let values = [...superheroes.values()];
  if (from !== undefined) {
    return values.slice(from, to);
  } else {
    return values;
  }
}

export function loadSampleData() {
  addSuperhero({ name: 'Superman' });
  addSuperhero({ name: 'Batman' });
  ...
}

loadSampleData();
```

Información estructurada con AJAX

- AJAX también puede ser para que el código JavaScript **consulte información al servidor**
- Con esa información puede **manipular la página** como quiera (por ejemplo mostrando una alerta)



The screenshot shows a Twitter account creation interface. At the top right is a blue Twitter bird icon and a blue 'Next' button. The main heading is 'Create your account'. Below it is a 'Name' input field containing 'michael gallego' with a red dashed underline. To the right of the field is a character count '14/50'. Below the name field is an 'Email' input field containing 'michael.gallego@gmail.com' with a red underline. Below the email field is a red error message: 'Email has already been taken.' At the bottom left is a blue link that says 'Use phone instead'.

- Cuando el código **JavaScript** hace peticiones, el servidor puede devolver:

Fragmentos de HTML

Se incrusta directamente en la página

Ej: Cargar más

Información estructurada

Se interpreta por JavaScript para modificar la página

Ej: Error de validación

Información estructurada con AJAX

- Cuando se solicita información estructurada al servidor la suele generar en **formato JSON**
- También se puede devolver en **formato XML**

```
{  
  validation:[  
    {id:"name",status:"ok"},  
    {id:"email",status:"error",message:"Invalid format"}  
  ]  
}
```

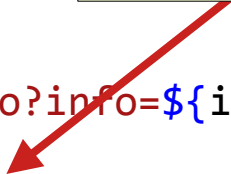
- Cliente

```
async function checkInfo(){  
  
  const info = 'someInfo';  
  
  const response = await fetch(`/checkInfo?info=${info}`);  
  
  const infoCheck = await response.json();  
  const content = document.getElementById("content");  
  
  content.innerHTML = `  
    <p>Valid: ${infoCheck.valid}</p>  
    <p>Message: ${infoCheck.message}</p>`;  
  
}
```

- Cliente

```
async function checkInfo(){  
  const info = 'someInfo';  
  
  const response = await fetch(`/checkInfo?info=${info}`);  
  
  const infoCheck = await response.json();  
  const content = document.getElementById("content");  
  
  content.innerHTML = `  
    <p>Valid: ${infoCheck.valid}</p>  
    <p>Message: ${infoCheck.message}</p>`;  
}
```

La información se
carga en un objeto con
response.json()



- Servidor


```
import express from 'express';
const router = express.Router();
...
router.get('/checkInfo', (req, res) => {
  let info = req.query.info;
  let response = {
    valid: false,
    message: `Info '${info}' not valid`
  }
  res.json(response);
});
export default router;
```

Información estructurada con AJAX

- Servidor

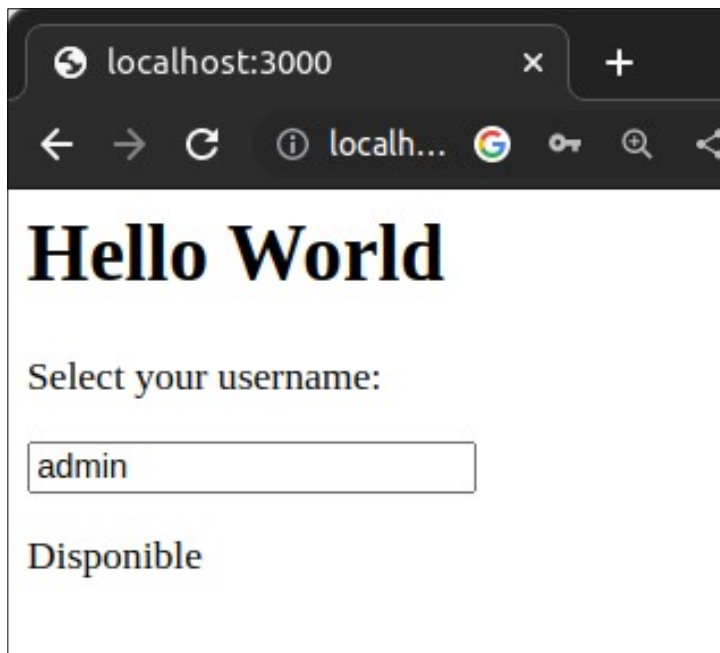
```
import express from 'express';
const router = express.Router();
...
router.get('/checkInfo', (req, res) => {
  let info = req.query.info;
  let response = {
    valid: false,
    message: `Info '${info}' not valid`
  }
  res.json(response);
});
export default router;
```

La información se envía
con `res.json()`

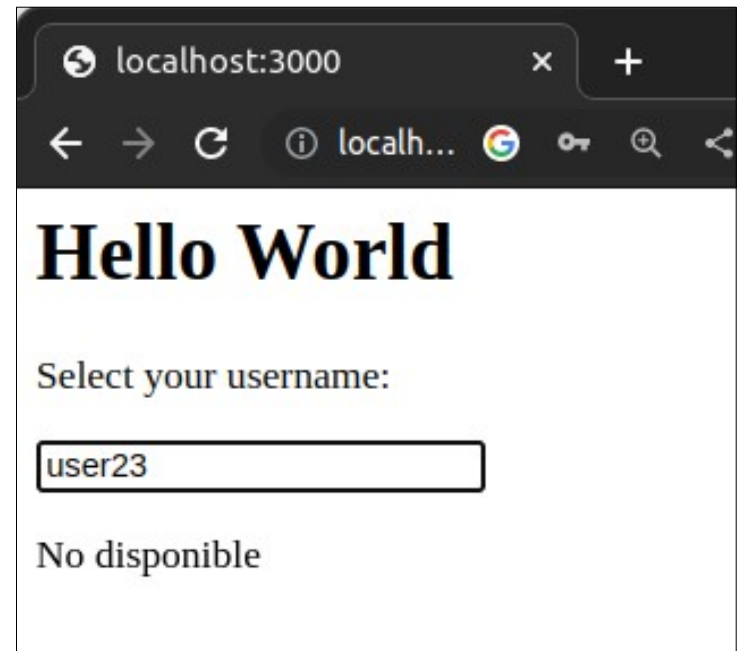


Ejercicio 2

- Web con input de texto que consulta disponibilidad al servidor según va escribiendo el usuario



A screenshot of a web browser window with the address bar showing 'localhost:3000'. The page content includes the heading 'Hello World', the text 'Select your username:', a text input field containing 'admin', and the status 'Disponible' below it.



A screenshot of a web browser window with the address bar showing 'localhost:3000'. The page content includes the heading 'Hello World', the text 'Select your username:', a text input field containing 'user23', and the status 'No disponible' below it.

Ejercicio 2

- El evento “input” permite ejecutar una función cada vez que el texto cambia

```
<input id='username' oninput='checkUsernameAvailability()' type='text'>
```

- Nota: En las web reales sólo se hace la petición cuando el usuario deja de escribir durante un tiempo para no saturar al servidor

Async / Await

- En programación, una función puede bloquearse o no:
 - Si sólo ejecuta **cálculos** en el procesador, **no se bloquea**.
 - Si solicita datos por red o al disco (**Entrada/Salida, IO**) y **espera** a que lleguen o simplemente se espera durante un tiempo sin hacer nada, **se bloquea**.

Async / Await

- En **Java** (y en la mayoría de los lenguajes de programación), cuando se hace una **llamada** a una función **no se sabe si** esta función es **bloqueante o no**.
- En **JavaScript**, a las funciones **bloquantes** (por IO o por esperas), se las tiene que ejecutar de una forma **especial**.

Async / Await

- En JavaScript las funciones bloqueantes habitualmente se ejecutan con **await**
- Si una función llama a otra con await, la función debe declararse con **async** (y se convierte en bloqueante)

```

async function loadAJAX(){

    const response = await fetch('/pagePart');
    const pagePart = await response.text();

    ...

}

```

Async / Await

- En versiones previas de JavaScript **no se podía usar `await`**
- Cuando las funciones bloqueantes **se ejecutan sin `await`** devuelven un objeto **Promesa (*Promise*)**
- Este objeto tiene un método **`then(...)`** para definir qué código ejecutará cuando **termine el bloqueo** y esté **disponible el valor** de red o disco

Async / Await

Con async / await

```
async function loadAJAX(){
    const response = await fetch('/pagePart');
    const pagePart = await response.text();
    ...
}
```

Sin async / await

```
function loadAJAX(){
    fetch('/pagePart')
        .then(response => response.text())
        .then(pagePart => { ... });
}
```

Async / Await

- Si hay código después de haber configurado la promesa se ejecutará antes de que haya llegado el valor de la petición de red

```
function loadAJAX(){
    fetch('/pagePart')
        .then(response => response.text())
        .then(pagePart => console.log('Valor recibido'));
    console.log('Petición enviada');
}
```



Petición enviada
Valor recibido

Async / Await

- El código **sin await** es un poco más **complejo** de entender y está más **limitado**.
- Pero es importante que lo conozcamos por varios motivos:
 - Se olvida poner el `await`
 - Documentación con promesas
 - Ejecución de funciones bloqueantes en paralelo

Async / Await

- Se olvida poner await
- Tenemos que saber que en vez del valor esperado tenemos una promesa

```
function loadAJAX(){
    const response = fetch('/pagePart');
    const pagePart = response.text();
    ...
}
```

ERROR: **response** es una promesa y no tiene el método **text()**

Async / Await

- **Documentación con promesas**
 - Como en versiones anteriores de JavaScript no se podía usar async/wait hay mucha **documentación con ejemplos con promesas**
 - Pero se puede usar async/await

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

Async / Await

- Ejecución de funciones bloqueantes en paralelo
- Se pueden procesar los resultados según están disponibles

```

async function loadAJAX1() { ... }
async function loadAJAX2() { ... }

loadAJAX1().then(v => console.log(v));
loadAJAX2().then(v => console.log(v));

```

Async / Await

- Ejecución de funciones bloqueantes en paralelo
- Se pueden procesar los resultados cuando han llegado todos

```

async function loadAJAX1() { ... }
async function loadAJAX2() { ... }

let [v1, v2] = Promise.all([loadAJAX1(), loadAJAX2()]);

console.log(v1);
console.log(v2);

```

Conclusiones

- Código JavaScript en el **navegador** puede ejecutar código JavaScript en el **servidor** (haciendo una **petición http**)
- El servidor puede **devolver HTML** (que se incrusta en la página) o **JSON** (que se procesa)
- Las funciones bloqueantes se pueden ejecutar con **await**.
- Si se ejecutan **sin await** se obtiene una **promesa**