Materiales del curso Q

Node JS, Express JS y SQL

4 Introducción

Node JS

En el módulo 4 de Adalab vamos a aprender a desarrollar aplicaciones de servidor con **Node JS**. Con Node JS podemos ejecutar un programa de JavaScript en la terminal.

Para ello vamos a aprender a usar el **framework Express JS**, que nos ayuda a montar un servidor; es decir, **nos ayuda a escuchar, procesar y responder las peticiones que se hacen desde una web**, ya sean peticiones a una API a través de un `fetch` o peticiones a ficheros (HTML, CSS, imágenes...) que realiza directamente el navegador.

Por último vamos a aprender a trabajar con la **base de datos SQL**, para poder almacenar en el servidor los datos de las usuarias de nuestras aplicaciones.

Contenidos de este módulo

Hasta ahora has aprendido las bases de la programación en JavaScript. Sabes que un dato sirve para almacenar información y una función sirve para ejecutar acciones. Sabes que utilizamos un `if` para elegir entre dos caminos y un `for` para repetir varias veces una parte del camino. Y conoces las buenas prácticas de dividir el código en ficheros pequeños para que cada uno tenga una responsabilidad concreta.

En este módulo, y durante el resto de tu vida profesional, aprenderás conceptos. Por ejemplo, que al programar un servidor los datos se reciben en los parámetros de una función y se devuelven a través de su retorno.

Tu preocupación debe centrarse en entender cada uno de estos conceptos, para qué sirven, por qué están pensados así y por qué no, cuál es su finalidad, etc.

En la primera parte del módulo aprenderemos:

- Qué es Node JS.
- Cómo ejecutar código JavaScript en un servidor.
- Qué es Express JS.
- Cómo crear una API como las que hemos estado utilizando durante el curso.
- Cómo crear un servidor de ficheros estáticos.
- Cómo crear un servidor de ficheros dinámicos.

En la segunda parte del módulo aprenderemos:

- Cómo y dónde se almacenan los datos de forma permanente en el servidor.
- Qué tipos de bases de datos existen.
- Cómo estructurar esta información en bases de datos.
- Cómo relacionar unos datos con otros.
- Cómo desplegar aplicaciones de back end en un servidor de producción real.

Ejemplos de código

En este módulo encontrarás vídeos con ejercicios. Sigue estos pasos para acceder al código fuente de los ejercicios:

1. Clona el repo https://github.com/adalab/ejercicios-de-los-materiales.
2. Abre, usa, trastea, rompe y juega con el código de cada mini lección. Ten en cuenta que el código subido a este repo es el código que queda al final de cada vídeo.
3. Recuerda hacer de vez en cuando un `git pull` para bajarte nuevos ejercicios que hayamos añadido.

Ejercicios

En este módulo hemos preparado 2 tipos de ejercicios:

- **Ejercicios normales:** estos son ejercicios pequeños y aislados. Los encontrarás en cada lección y los deberás hacer tú.
- **Ejercicio de Netflix para el pair programming:** os proponemos un ejercicio para que realicéis entre tu compañera y tú en la hora de pair programming. Lo bueno de este ejercicio es que también es un ejercicio de tamaño medio, como el que harás en las entrevistas de trabajo. Es obligatorio que lo hagas todos los días porque aprenderás cosas que no se explican en el resto de materiales.

¿Qué esperamos de ti?

Te pedimos que durante el módulo:

- **Aprendas cada uno de los conceptos aislados que te vamos a enseñar.** El objetivo es que en tu cerebro esté muy claro qué acciones hay que realizar en el servidor y cuáles en el front, qué es una petición y qué es una respuesta, si una petición se hace para obtener un fichero o para obtener datos...
- **Aprendas a relacionar entre sí los conceptos aislados.** Nosotras te vamos a seguir ayudando a unir estos conceptos, pero queremos que tú seas capaz de entender, por ejemplo, que para recibir datos en un servidor antes hay que especificar por dónde los quiero recibir.
- Leas, además de estos materiales, la documentación oficial de Node, ya que está muy bien escrita. **Queremos que te acostumbres a leer documentación técnica por Internet.**
- Te sigas acostumbrando a **buscar en Internet** con Google y StackOverflow, donde todas dudas y problemas que tengas ya los ha tenido alguien antes y ha subido la solución.
- Disfrutes aprendiendo y programando servidores.

Cuando finalicemos este módulo sabrás **hacer aplicaciones de servidor pequeñas** y serás capaz de **seguir aprendiendo por tu cuenta**. Además podrás **comunicarte sin problemas con el equipo de back end** de la empresa donde trabajes.

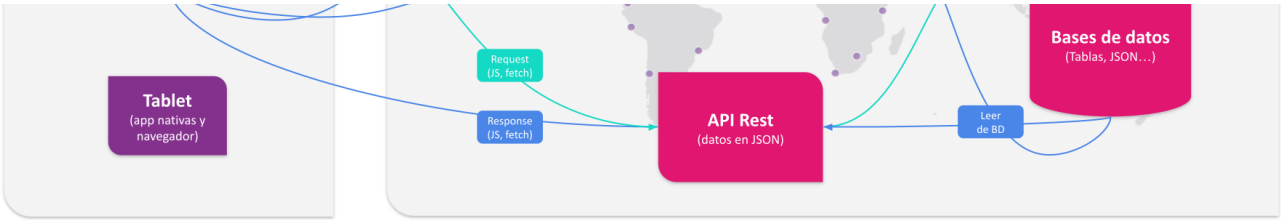
Sin duda podrás optar a posiciones de desarrolladora junior de back end con Node JS.

4.1 Node JS

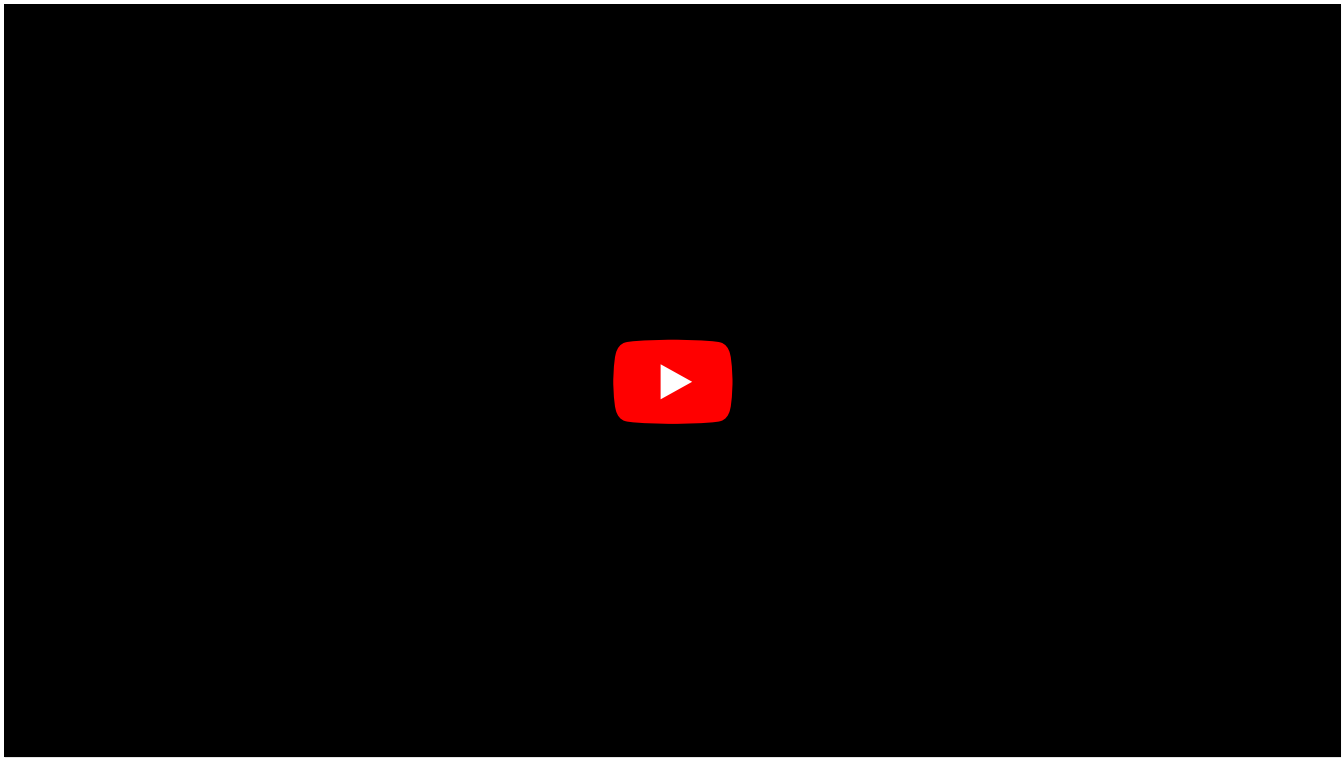
1.1 Comunicación entre front y back

Ahora mismo eres experta en programación front. Ya que vas a aprender cómo funciona la parte de back end, lo primero que tenemos que hacer es entender cómo se comunican ambos mundos.

> **Nota:** esta mini lección es básica.



Cómo se comunican el front y el back



Qué significa API Rest

Queremos hacer una pequeña aclaración: hasta ahora a la API la hemos llamado API. Así la llama todo el mundo. API significa literalmente **Application Protocol Interface** o **Application Programming Interface**. En otras palabras, un protocolo para comunicar dos ordenadores o dispositivos. Por ejemplo front con back.

En este módulo hablaremos de un tipo concreto de API que se llama **API Rest** o **API Restful**. Durante el módulo explicaremos más acerca de este protocolo, pero una mínima definición sería:

La API Rest es una convención, una especie de estándar, para organizar las comunicaciones entre navegador y servidor, como:

- Cuándo debemos usar un GET, POST...
- Por dónde enviar y recibir datos al servidor
- Qué estructuras de datos debemos enviar y recibir, etc.

1.2 Introducción a Node JS

Node JS

> **Nota:** esta mini lección es importante para entender qué cosas podemos hacer con Node.

Node JS es un lenguaje programación que nos sirve para **ejecutar JavaScript en la terminal de un ordenador**.

Existen muchísimos lenguajes de programación de back end (como PHP, Java, Python, .NET...). Nosotras vamos a aprender Node JS porque ya sabemos JavaScript y tenemos el 80% de los conocimientos necesarios. Cuando en el futuro quieras aprender otro lenguaje de back end te costará mucho menos.

Hasta ahora estamos acostumbradas a ejecutar nuestro código JavaScript en un navegador como Chrome. Gracias a Node JS podemos ejecutar JavaScript en una terminal y que este haga tareas como leer y escribir en los ficheros del ordenador, o podemos crear una aplicación de servidor.

Diferencias entre JavaScript y Node JS

Las diferencias entre un código de JavaScript ejecutado en un navegador y en una terminal son las características propias de cada uno de estos dos entornos. Por ejemplo:

- En un navegador las usuarias producen eventos (click, keyUp, scroll...) pero esto no tiene sentido en una terminal. Es decir, **una terminal no tiene una interfaz gráfica que puedan usar las usuarias**.
- **Una terminal tiene muchos más permisos para acceder a servicios del ordenador**, como acceder a los ficheros del ordenador, abrir puertos de Internet, instalar otros programas... Todas estas cosas no se pueden hacer desde un navegador por temas de permisos y seguridad.
- Node JS no va a ejecutar o interpretar HTML, CSS, imágenes... Pero sí los va a gestionar, crear, modificar o servir a las páginas web que los pidan. **En Node JS solo se ejecutan ficheros de JavaScript.**

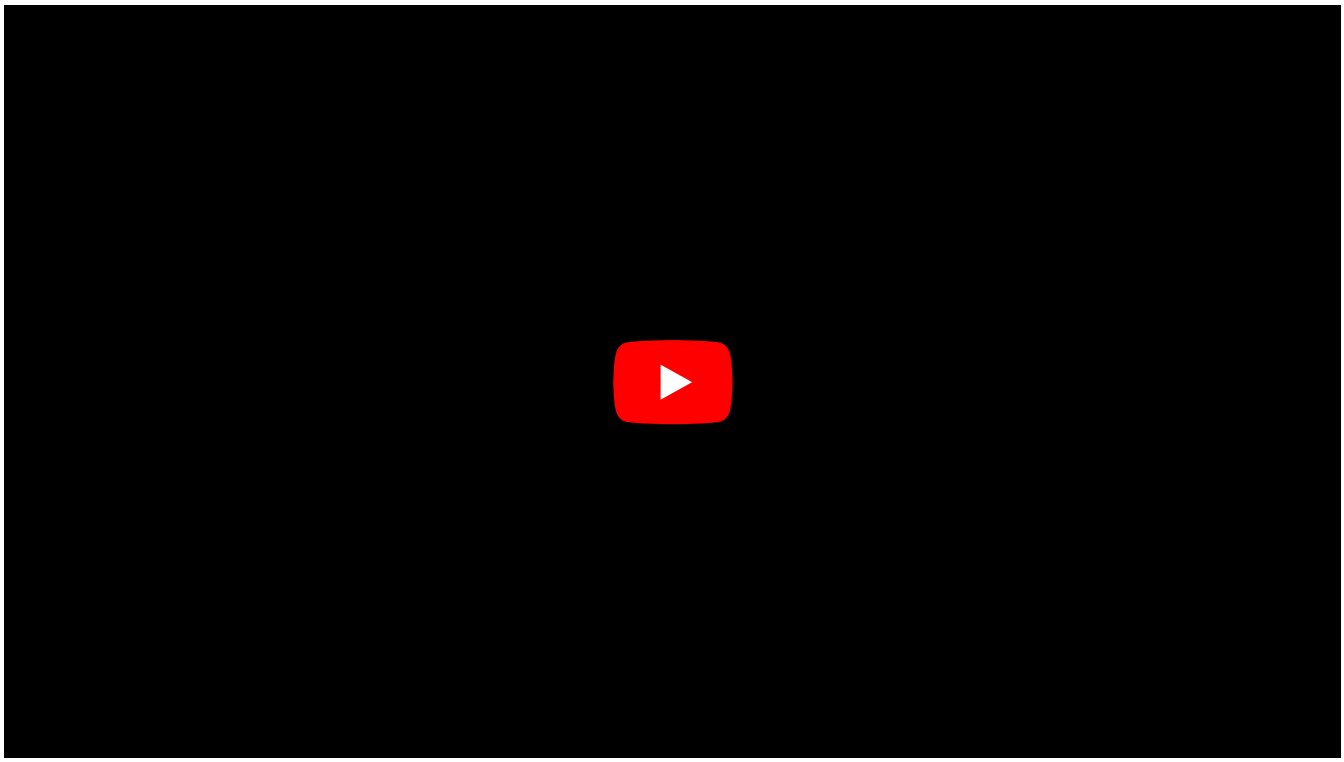
Similitudes entre JavaScript y Node JS

Todo lo demás es común entre JavaScript y Node JS. En Node JS también vamos a trabajar con nuestras variables, constantes, `if`, nuestros **amados** arrays y bucles, dividiremos el código en diferentes ficheros para exportar e importar, vamos a poder **depurar** nuestras aplicaciones... También trabajaremos con **asincronía**, que es especialmente útil en un servidor.

En resumen, nos interesa aprender de Node JS los conocimientos que son únicos de Node JS. También nos interesa aprender a pensar en cómo funciona un servidor.

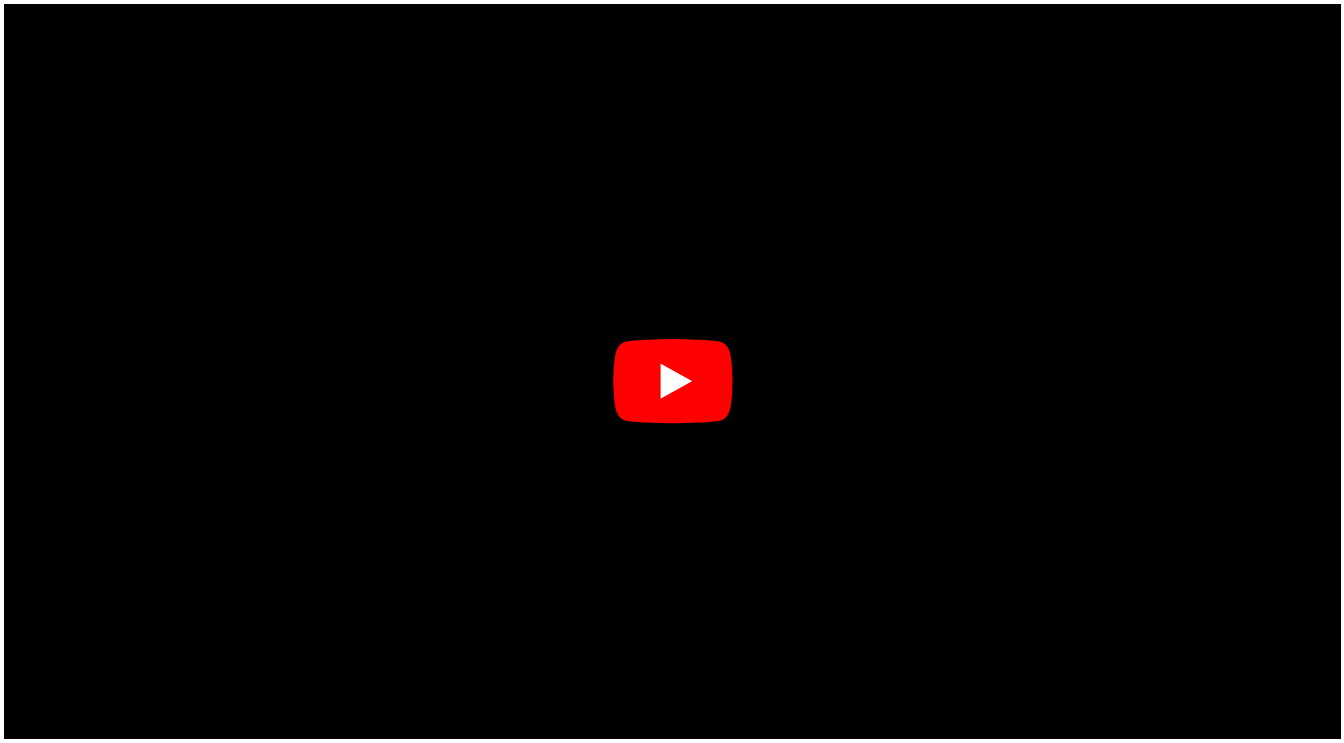
Qué es Node JS

En este vídeo explicamos qué es Node JS y ejecutamos nuestro primer programa:



> Ejercicio del vídeo

Y tú, ¿has utilizado alguna vez Node JS?



Recuerda lo que hemos comentado en este vídeo: pulsando **Ctrl+C** en la terminal, puedes forzar que un programa de Node JS finalice.

Acceso al sistema de ficheros

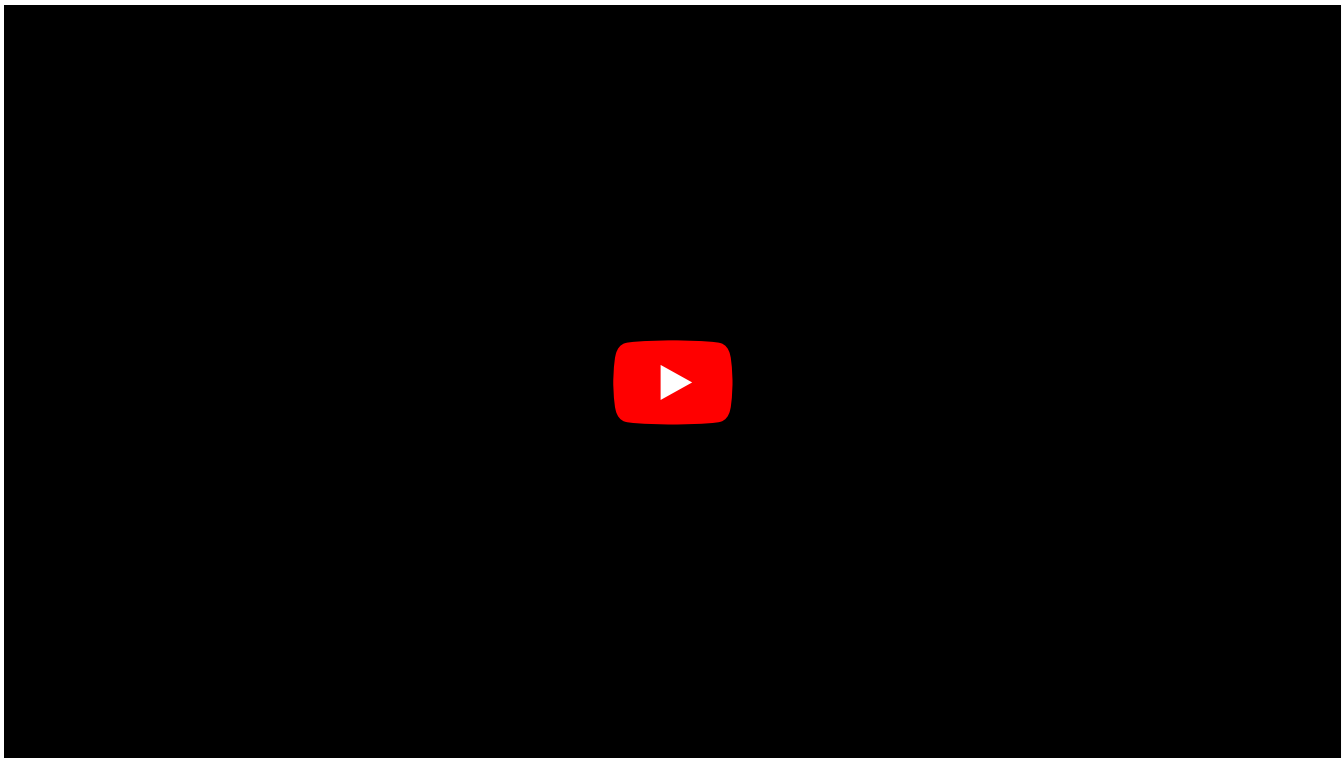
Otra funcionalidad muy útil y usada es el **File system (también conocido como fs)**, que no es otra cosa que una librería interna de Node para acceder a los archivos y carpetas del ordenador donde se está ejecutando.

Vamos a ver varios ejemplos de cómo se usa:

Leer ficheros

En NodeJS todas las operaciones de acceso al sistema de archivos están englobadas dentro del módulo File System (fs).

Si queremos leer un archivo de texto que tenemos en local usaremos ese módulo para extraer el contenido del fichero, indicando su ruta y otra serie de parámetros. Os contamos en el siguiente video como leer un fichero a través de un ejemplo.



> Ejercicio del vídeo (Recuerda que el código de los ejercicios está como se muestran al final del vídeo)

Por cierto: no se si te has fijado, pero los datos que consoleamos se muestran en la terminal de VS Code.

En resumen los pasos a ejecutar son:

1. Importar el módulo fs, donde el nombre del objeto para operar con el sistema de archivos lo hemos guardado en una variable llamada 'fs'. Podrías usar el nombre de variable que tú desees:

```
let fs = require('fs');
```

2. El método para acceder a un fichero es `readFile()`, que recibe tres parámetros, siendo el segundo de ellos opcional.
   - En el primer parámetro le indicamos el nombre del archivo que deseamos leer, cadena de texto que contiene el nombre del archivo y la ruta que estamos accediendo.
   - El segundo parámetro, el opcional, puede ser tanto un objeto como una cadena, e indica una cadena que contendrá el juego de caracteres en el que el archivo está codificado, por ejemplo 'utf-8'.
   - El tercer parámetro es la función callback, que se ejecutará en el momento que el archivo está leído y se encuentra disponible para hacer alguna cosa.

   ```
   fs.readFile('./input.json', 'utf8', handleFile);
   ```

3. Crear la función manejadora `handleFile()`, que recibe dos parámetros, el archivo ya leído o bien, si ocurre algún error, entonces recibiremos el correspondiente objeto de error de Node.

   ```
   const handleFile = (err, fileContent) => {
     if (err) {
       console.log('Error:', err);
     } else {
       console.log('El contenido del fichero es:', fileContent);
       console.log('La longitud del contenido es:', fileContent.length);
     }
   };
   ```

   Si ese archivo existe, veremos su contenido como salida del programa.
   Si el archivo no existe, nos mostrará el error y podremos ver los diferentes campos del objeto error de node, desde los que podremos saber qué es lo que ha ocurrido. La salida sería parecida a esta:

   ```
   error:  { Error: ENOENT: no such file or directory, open 'archivo-inexistente.txt'
    at Error (native)
   errno: -2,
   code: 'ENOENT',
   syscall: 'open',
   path: 'archivo-inexistente.txt' }
   ```

Leer ficheros JSON

Una de las formas más extendidas para intercambiar información entre aplicaciones escritas en diferentes idiomas es utilizar el JSON. Para leer ficheros JSON usamos las mismas líneas de código vistas anteriormente. Os contamos en el siguiente video como leer y trabajar un fichero JSON a través de un ejemplo.

> Ejercicio del vídeo

En resumen, es ejecutar los mismos pasos que para leer un fichero cualquiera, lo que solo debemos modificar los datos a través del `JSON. parse()`, la función manejadora entonces quedaría asi:

```
const handleFile = (err, fileContent) => {
  if (err) {
    console.log('Error:', err);
  } else {
    // console.log('Contenido del fichero en formato texto:', fileContent);
    const jsonData = JSON.parse(fileContent);
    console.log('Contenido del fichero en formato objeto:', jsonData);
    console.log(`Título: ${jsonData.title}`);
    console.log(`Contenido: ${jsonData.content}`);
  }
};
```

Escribir en ficheros

El método `fs.writeFile()` escribe datos de forma asincrónica en un archivo, reemplazando el archivo si ya existe. Os contamos en el siguiente video como crear y escribir un fichero a través de un ejemplo.

> Ejercicio del vídeo

En resumen los pasos a ejecutar son:

1. Importar el módulo fs:

```
let fs = require('fs');
```

2. El método para acceder a un fichero es `writeFile()`, que recibe tres parámetros.
   - En el primer parámetro especifica la ruta al archivo donde se realizará la operación de escribir.
   - El segundo parámetro especifica los datos que se escribirán en el archivo.
   - El tercer parámetro es la función callback.

```
s.writeFile('./output.txt', fileContent, handleWriteFile);
```

3. Crear la función manejadora `handleWriteFile()` que solo recibe si ocurre algún error durante la escritura del fichero.

```
const handleWriteFile = (err) => {
  if (err) {
    console.log('Error:', err);
  } else {
    console.log('The file has been saved!');
  }
};
```

Leer y escribir en ficheros

¿Y si combinamos leer y escribir ficheros en el mismo programa? Os contamos en el siguiente video como leer y escribir ficheros a través de un ejemplo.

> Ejercicio del vídeo

Conclusiones

Para **leer un fichero** desde Node JS usamos:

```
fs.readFile(
  './ruta-relativa-del-fichero/nombre-del-fichero.extension',
  'utf8',
  funcionManejadora
);
```

Para **leer un fichero JSON** desde Node JS usamos la misma función que para cualquier otro fichero pero luego tenemos que pasar el contenido del fichero de string a objeto con:

```
const jsonData = JSON.parse(fileContent);
```

Para **escribir en un fichero** desde Node JS usamos:

```
fs.writeFile(
  './ruta-relativa-del-fichero/nombre-del-fichero.extension',
  'utf8',
  funcionManejadora
);
```

Ejercicios

1: Suma

1. Crea un `index.js`.
2. Crea una función `add`:
   - La función debe recibir dos números como parámetros.
   - La función debe retornar el resultado de la suma.
3. Ejecuta y consolea el resultado.

2: Escribe en un fichero

1. Crea un `index.js`.

2. Crea un objeto con 3 propiedades, por ejemplo:

```
const myObject = {
  user: 'Mari Carmen',
  email: 'mari@gmail.com',
  age: 28,
};
```

3. Consolea el objeto en modo objeto.
4. Guarda el objeto tal cual en el fichero de destino, por ejemplo en `output.txt`:
   - La terminal te muestra un error; ¿sabrías decir por qué?
5. Convierte el objeto a texto y consoléalo.
   - Pista: ¿Te acuerdas de qué herramienta tenemos para pasar un objeto a texto?
6. Guarda el objeto en modo texto en el fichero de destino y verás que sí funciona.
7. Abre el fichero de destino y mira cómo se ha guardado la información.

3: Lee un fichero, modifica los datos y escríbelos en otro fichero

1. En un ejercicio nuevo crea un fichero `input-file.json` con lo que tiene el fichero de destino del ejercicio anterior, por ejemplo:

```
{"user": "Mari Carmen", "email": "mari@gmail.com", "age": 28}
```

> Cuidado con la sintaxis: el JSON tiene que estar bien escrito, por ello, los nombres de las propiedades como `user` tienen que estar entre comillas dobles.

2. Crea un `index.js`.
   1. Lee el contenido del fichero `input-file.json` y guárdalo en una constante.
   2. El contenido de dicha constante será un texto. Conviértelo a objeto:
      - Pista: ¿Te acuerdas de qué herramienta tenemos para pasar un texto a objeto?
   3. Modifica alguna propiedad del objeto, como por ejemplo el nombre o la edad.
   4. Guarda el objeto en modo texto en un fichero de destino llamado `output-file.json`.

Pista: la asincronía es importante; debes guardar en el fichero de destino después de leer el fichero de origen.

Con esto hemos aprendido que podemos guardar datos en nuestro servidor en ficheros JSON.

1.3 Módulos de Node JS

> **Nota:** esta mini lección es importante.

Qué son los módulos

Un módulo (también llamado paquete) **es una librería de código con una funcionalidad concreta**, que han creado otras personas, para que nosotras lo utilicemos dentro de nuestro código. Esas personas pueden ser los creadores de Node JS o cualquier otra programadora.
La funcionalidad que puede tener un módulo es muy diversa, por ejemplo:

- Funcionalidades que nos ayudan a automatizar tareas.
- Funcionalidades que nos ayudan a trabajar más fácilmente con datos complejos.
- Funcionalidades que procesan imágenes y reducen su tamaño para que pesen menos, y así cuando el navegador las carga la usuaria gaste menos datos de su conexión y las imágenes se carguen más rápidamente.
- Funcionalidades que nos ayudan a subir un fichero desde el navegador de la usuaria al servidor.
- Y muchas más.

No tiene sentido que tengamos que volver a programar en nuestras aplicaciones código que ya hemos creado en aplicaciones anteriores, o que ya ha creado otra persona en otro proyecto. **¡¡¡Compartir es vivir!!!**

Para qué son útiles los módulos

Los módulos son útiles para:

- Separar el código en diferentes partes y ficheros y así organizar la aplicación mejor. Es mejor tener muchos ficheros pequeños que pocos grandes.
- Separar la responsabilidad del código. Cada módulo o fichero se encarga de hacer solo una cosa. Así no mezclamos en un único fichero muchas cosas.
- Tener código privado. Desde fuera de un código solo podemos acceder al código que el módulo publica a través del `export`. El código no publicado es privado.
- Reutilizar código llevándonos el módulo a otro proyecto.

Módulos antiguos vs. módulos modernos

Módulos modernos

En React ya hemos utilizado los módulos. Los utilizamos para crear componentes de React en diferentes ficheros. Si recuerdas, la sintaxis para importar un componente es:

```
import Header from './Header.js';
```

Y en el fichero `Header.js` la sintaxis para exportar el componente es:

```
export default Header;
```

Esta forma con la sintaxis `import` y `export` es la moderna. El problema es que Node JS todavía no

ha terminado de implementar esta sintaxis en su versión actual. Hasta que Node JS no implemente los módulos modernos no podemos utilizarlos

Módulos antiguos

La forma antigua de hacerlo es cambiando la sintaxis por:

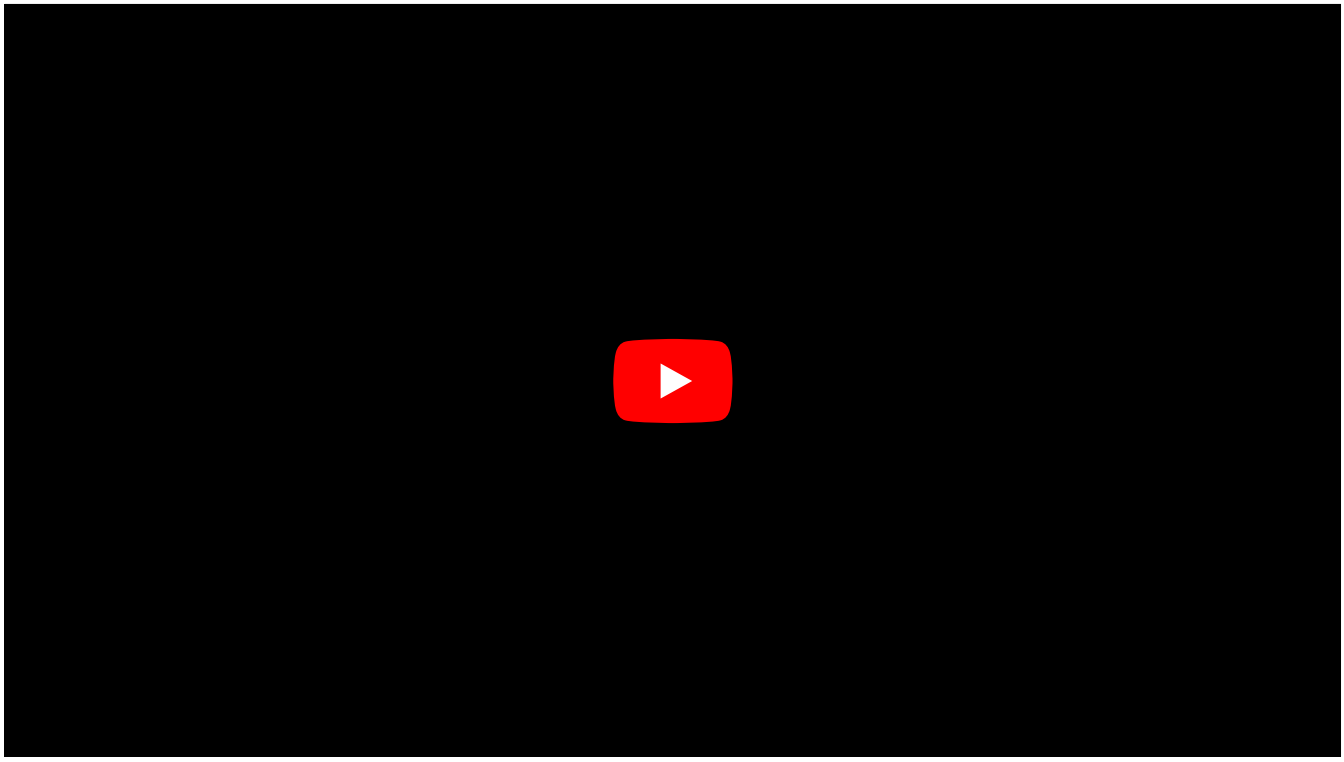
```
const Header = require('./Header.js'); // que equivale a: import Header from './Header.js';
```

y

```
module.exports = Header; // que equivale a: export default Header;
```

Como veréis es simplemente un cambio de sintaxis, nada más. **Esta sintaxis antigua es la que debemos usar en Node JS.**

Qué son y para qué sirven los módulos



> Ejercicio del vídeo

En el video anterior hemos visto que son los módulos y un ejemplo de como usarlos. Podemos crear nuestros propios módulos y luego exportarlos para que sean utilizados en otra parte de nuestro fichero.

1. Para exportar módulos utilizamos `module.exports = {}` y entre las llaves colocamos todo aquello que queramos exportar.

   ```
   function suma(a, b) {
     return a + b;
   }
   ```

```
function multiplicar(a, b) {
  return a * b;
}

module.exports = {
  suma: suma,
  multiplicar: multiplicar,
};
```

2. Asumiendo que el archivo anterior se llame `operaciones.js`, se puede utilizar el mismo en otro archivo.js de la siguiente manera:

```
let operaciones = require('./operaciones');
operaciones.suma(2, 3);
```

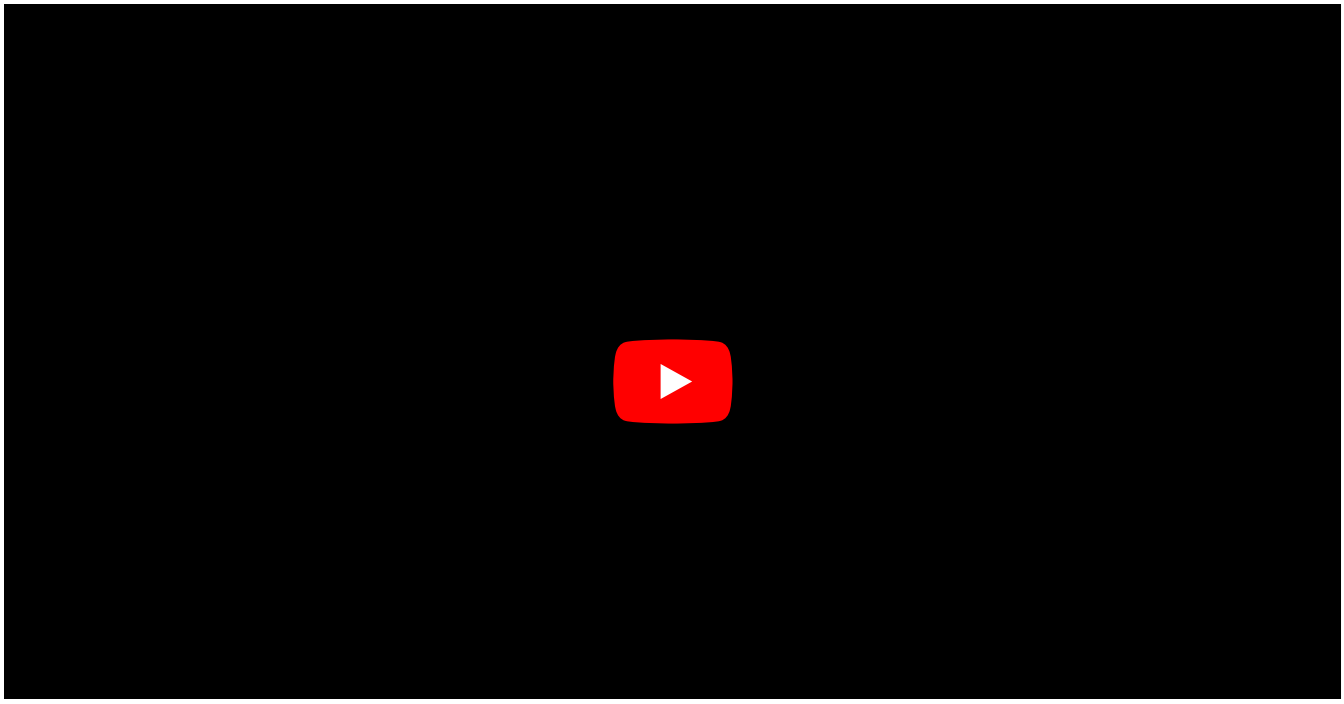
Tipos de módulos

Hay 4 tipos de módulos y los vamos a explicar a continuación:

- Mis propios módulos o custom modules
- Módulos nativos de Node JS
- Módulos instalados con NPM
- Módulos especiales de tipo JSON, para datos

Mis propios módulos

Los módulos propios son módulos que creamos las desarrolladoras , son propios de nuestra aplicación y responden a una funcionalidad de la misma. Y como hemos visto se exportan las funciones con la línea `module.exports = {}` y se utiliza importando en otros ficheros a través de `let nombre_modulo = require('./ruta_al_modulo');`

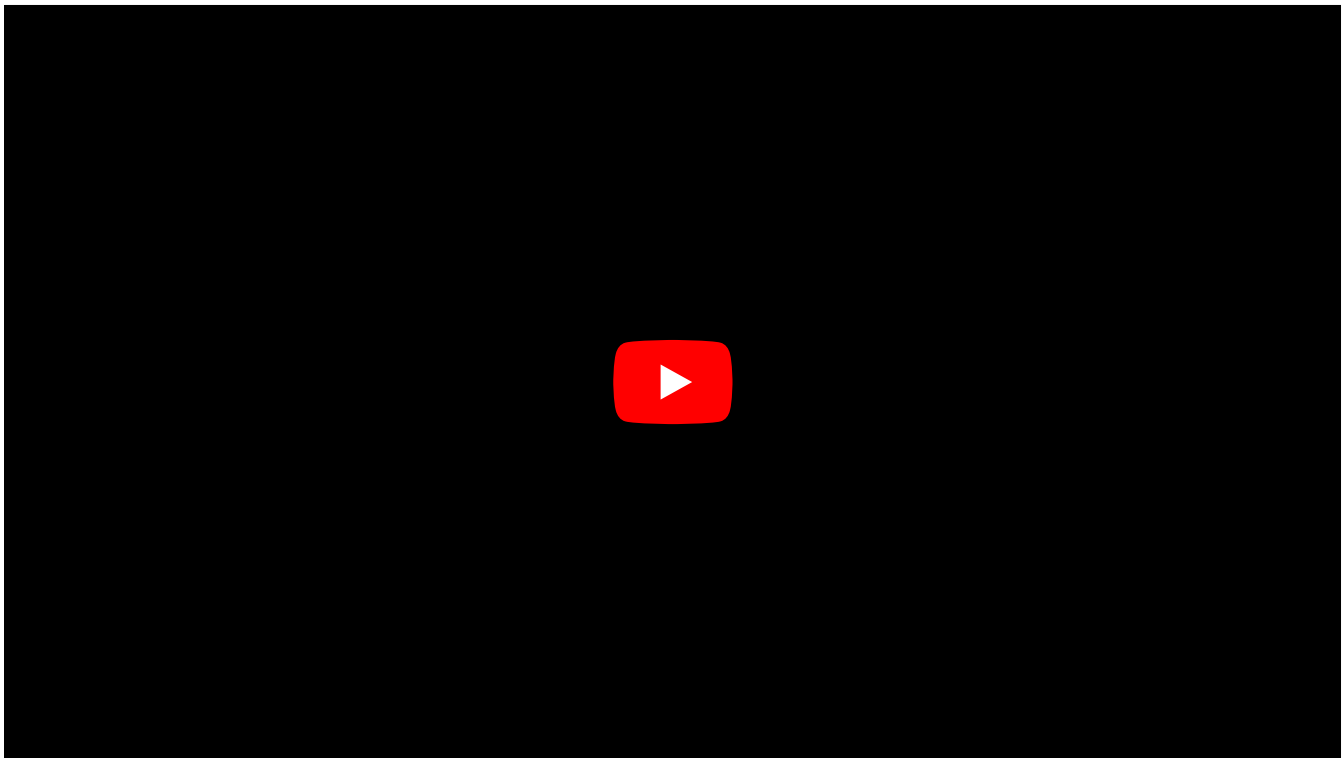


> Ejercicio del vídeo

Módulos nativos de Node JS

Los módulos nativos son los que vienen instalados por defecto en NodeJS, podemos ver los que existen a través de la documentación de NodeJS. Por ejemplo:

- HTTP: para configurar la comunicación entre servidores y clientes.
- FileSystem: El módulo fs permite interactuar con el sistema de archivos
- DNS: El módulo dns permite la resolución de nombres. Por ejemplo, para buscar direcciones IP de nombres de host.



> Ejercicio del vídeo

Módulos instalados con NPM

Node Package Module(NPM) forma parte de Node.js es una herramienta que permite instalar/desinstalar dependencias, gestión de versiones, gestión de dependencias necesarias para ejecutar un proyecto.

Fichero package.json Para usar los módulos que instalamos con NPM, el proyecto debe contener un archivo llamado package.json. Dentro de ese archivo se encuentran los metadatos específicos para los proyectos. Los metadatos muestran algunos aspectos del proyecto en el siguiente orden:

- El nombre del proyecto

- La versión inicial
- Descripción
- El punto de entrada
- Comandos de prueba
- El repositorio git
- Palabras clave
- Licencia
- Dependencias
- Dependencias de desarrollo

Iniciar un proyecto con npm Para construir un proyecto con node, ejecuta el comando `npm init` que inicializa el proyecto, funciona como una herramienta para crear el archivo `package.json`, solo debemos reponder a las indicaciones del `npm init`

Instalación de módulos npm Para instalar los módulos utiliza el siguiente comando, donde es el nombre del módulo que se desea instalar. Este comando instalará el módulo en la carpeta `/node_modules` en el directorio del proyecto. Cada vez que instales un módulo desde el administrador de paquetes Node, este se instalará en la carpeta node_modules.

```
$ npm install <modulo>
$ npm i <modulo>
```

Os contamos en el siguiente video como inicializar un proyecto e instalar módulos a través de un ejemplo.

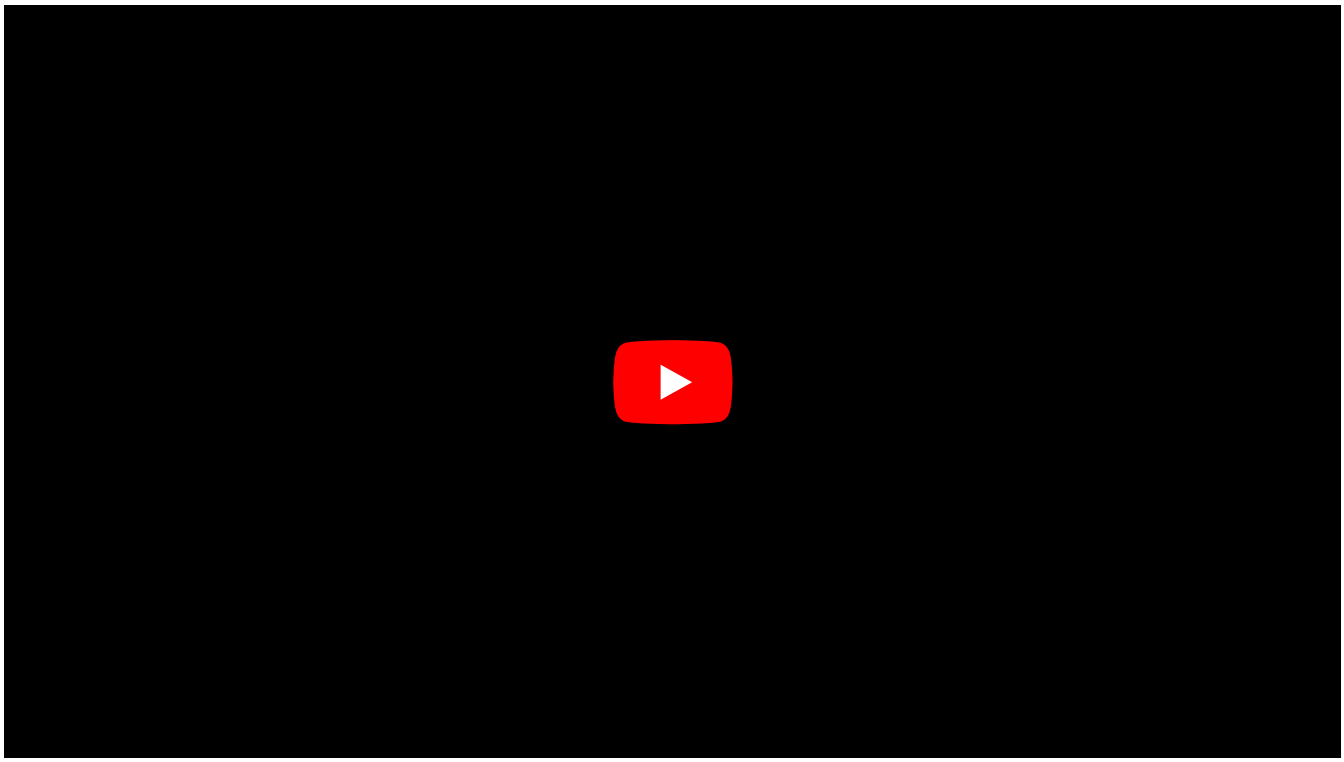
> Ejercicio del vídeo

Cuando tengas tiempo puedes echarle un ojo a toda la configuración que podemos meter en el `package.json`.

Módulos JSON (módulo de datos)

Los módulos de tipo JSON guardan información en este formato, generalmente configuraciones de mi aplicación, credenciales y datos. Se importa y exporta igual que los anteriores. De esta forma se separan en ficheros independientes la lógica de los datos de la aplicación.

Os contamos en el siguiente video a través de un ejemplo utilizando la librería `moment.js`.



> Ejercicio del vídeo

Cuando tengas tiempo puedes echarle un ojo al módulo Moment JS que se puede usar en proyectos de front y de back.

Conclusiones

La forma actual de importar y exportar un módulo es:

```
const Header = require('./Header.js'); // que equivale a: import Header from './Header.js';

module.exports = Header; // que equivale a: export default Header;
```

Para importar nuestros propios módulos y los módulos JSON usamos **la ruta relativa**, ya que son ficheros que están en nuestro proyecto.

```
const Header = require('./rutaRelativaAlModulo.js');
```

Para importar módulos nativos o de NPM utilizamos el nombre del módulo:

```
const fs = require('fs');
```

Ejercicios

1. Librería de file system

1. Mira, ejecuta y entiende los ejemplos que están en las carpetas `node-modules-read-and-write-files-with-one-module` y `node-modules-read-and-write-files-with-two-modules`.
2. Piensa y razona si te gusta más que las dos funciones estén en un solo módulo o separadas en dos módulos.

2. Mi propia librería Math

1. Crea un `math.js` que sea un módulo.
   - Exporta dos funciones, una para sumar y otra para restar.
   - Estas dos funciones deben recibir dos números como parámetros.
   - Estas dos funciones deben retornar el resultado de la operación.
2. Crea un `index.js`. e importa el módulo `math.js`.
   - Usa el módulo importado para hacer una suma y consolea el resultado.
   - Usa el módulo importado para hacer una resta y consolea el resultado.

3. Librería Math avanzada: módulos que usan otros módulos

1. Crea un `math-add.js` que sea un módulo que exporta una función de suma.
2. Crea un `math-sub.js` que sea un módulo que exporta una función de resta.
3. Crea un `math.js` que sea un módulo que importa los dos módulos anteriores y los exporta dentro de un objeto.
4. Crea un `index.js`.
   - Importa en `index.js` el módulo `math.js`.
   - Haz una suma y consolea el resultado.
   - Haz una resta y consolea el resultado.

4. Lodash: obtener la unión

En NPM hay un módulo que se llama Lodash que nos ayuda a trabajar fácilmente con grandes cantidades de datos.

1. Crea un `index.js`.

1. Importa el módulo `Lodash`. Para ello, antes tienes que instalarlo dentro del `package.json`.
2. Crea dos constantes para los arrays: `[1, 2, 3]` y `[2, 3, 4]`.
3. Usa el módulo lodash para hallar la **unión** de estos dos arrays. Para ello necesitas buscar en la documentación de Lodash.
4. Si consoleas el resultado que te devuelve la función de Lodash el resultado debe ser `[1, 2, 3, 4]`.

1.4 Nodemon

> **Nota:** esta mini lección no es importante pero te ahorrará mucho tiempo al programar.

Nodemon es un módulo de NPM que nos ayuda programar más rápido.

Recordemos que lo que hace la extensión de VS Code **Live Server** es observar cambios en los ficheros de nuestra página web; cuando algún fichero cambia, Live Server se da cuenta y nos refresca la página para que no tengamos que hacerlo nosotras manualmente. **Pues Nodemon hace lo mismo pero para proyectos de Node JS.**

Cómo instalar Nodemon

Vamos a instalar Nodemon de forma global en nuestro ordenador. De esta forma lo tendremos disponible para cualquier proyecto de Node JS. Abre una terminal y ejecuta:

```
sudo npm install -g nodemon
```

Cómo usar Nodemon

Hasta ahora hemos ejecutado cada ejercicio de Node ejecutando en la terminal `node index.js`. A partir de ahora podemos ejecutar `nodemon index.js` y nuestro proyecto se ejecutará de la misma manera.

La diferencia es que a partir de hoy, cada vez que modifiquemos el fichero `index.js` o cualquier otro fichero de nuestro proyecto, Nodemon parará el proyecto y volverá a ejecutarlo otra vez, evitándonos hacerlo nosotras mismas de forma manual.

A partir de ahora te recomendamos usar siempre Nodemon para ahorrar tiempo.

Conclusiones

Usa nodemon para trabajar más deprisa. Instálalo con:

```
sudo npm install -g nodemon
```

Y úsalo con:

```
nodemon index.js
```

Ejercicios

1. Prueba Nodemon

Abre un ejercicio de los vídeos anteriores o uno que hayas creado tú y arráncalo con Nodemon. Haz algún cambio en tus ficheros y verás que, al guardarlo, en la terminal se reinicia tu proyecto.

Netflix

Ejercicio para el pair programming

Hemos creado este ejercicio para que lo hagáis durante la hora de pair programming entre tu compañera y tú. Este ejercicio es incremental, es decir, cada día vamos a ir añadiendo las nuevas funcionalidades que hemos aprendido.

Es obligatorio que lo hagáis en la hora de pair programming porque:

- Aquí os enseñamos trucos y buenas prácticas.
- El último día del módulo haréis **una evaluación sobre este ejercicio** con vuestra profesora. Esta evaluación consistirá en una entrevista técnica similar a la de los módulos anteriores, pero en esta ocasión por parejas.

Enunciado

Vamos a crear una plataforma de series y películas que replique la aplicación de Netflix. Para entender lo que vamos a hacer durante los próximos días debes empezar por estos pasos:

Ejercicios

1. Crea el proyecto

1. Crea un repositorio nuevo en GitHub llamado `promo-X-module-4-pair-Y-netflix`. Hazlo en la organización de Adalab.
2. Descarga este código.
3. Descárgalo dentro del nuevo repo que has creado.
4. Súbelo a tu repo.
5. No hagas nada con GitHub Pages, ya que no lo vamos a utilizar.

2. Arranca el backend

1. En la raíz del proyecto ejecuta `npm install` para instalar las dependencias.
2. Ejecuta `npm run dev` para arrancar el servidor.
    - Verás que se arranca en http://localhost:4000.

3. Arranca el front end

1. En una nueva terminal ejecuta `cd web` y `npm install` para instalar las dependencias.
2. Ejecuta `npm start` para arrancar el proyecto de React.
    - Verás que se arranca en http://localhost:3000.

La idea es que **mientras estemos desarrollando este proyecto siempre tengamos arrancados el servidor y React a la vez**, para que ambos se puedan comunicar.

Cuando subamos nuestro código a un servidor real, cambiaremos la forma de trabajar, pero esto lo explicaremos más adelante.

4. Prueba el proyecto

1. Abre la página http://localhost:3000.
2. Entra en el **Inicio**: verás que se muestra un listado de películas y unos filtros.
    - Los filtros ahora mismo no funcionan porque los tenéis que programar vosotras.
3. Entra en el **Registro**: verás que si envías el formulario, este siempre responde **Usuario ya existente**.
4. Entra en el **Login**: introduce los datos **mari@adalab.es** / **12345678** y verás que te muestra el error **Usuario no encontrado**.
5. Entra en el **Login** otra vez: introduce los datos **mari@gmail.com** / **12345678** y verás que sí te permite entrar en la parte privada de la web.
6. Entra en **Mi perfil**: en esta página recuperamos los datos del perfil de la usuaria con un formulario por si los quiere modificar.
7. Entra en **Mis películas**: verás que hay un listado de películas, que son las favoritas de la usuaria.
8. Pulsa en **Cerrar sesión**: verás que te saca de la sesión, te manda al inicio y se refresca la página.
    - Es una buena práctica que cuando la usuaria cierre la sesión refresquemos la página, para limpiar todos los datos del estado de React. Esto lo podemos hacer manualmente a través del código (en vez de refrescar) pero es más propenso a errores.

Como ves el proyecto está funcionando, pero solo es una simulación. Funciona porque la parte de React está terminada, y la parte de los servicios de `web/src/services/` está gestionando datos falsos.

5. Entiende el código del proyecto

1. Abre `web/src/components/App.js` para entender lo que hace el proyecto de React. Este es el fichero principal y el que tiene toda la lógica de la aplicación front. **Debes entender bien este fichero para poder hacer los siguientes ejercicios.**

2. Abre el resto de componentes de `web/src/components/` para ver cómo funcionan. Verás que no tienen mucho misterio. No pierdas mucho tiempo en ellos.

6. Parte del código que vas a editar

1. Abre los servicios `web/src/services/api-movies.js` y `web/src/services/api-user.js` y verás que ya están programados, pero con datos falsos.
   - En estos dos ficheros tenemos varios `fetch`.
   - Estos `fetch` son todos de tipo `GET` y llaman al endpoint `//beta.adalab.es/curso-intensivo-fullstack-recursos/apis/netflix-v1/empty.json` que siempre devuelve un objeto vacío.
   - Los `then` de los `fetch` están devolviendo datos falsos puestos a pincho en el código.
   - Estos ficheros son los únicos que debéis modificar del proyecto de React.
2. Abre `src/index.js` y verás que está prácticamente vacío. Este es el fichero principal del servidor y lo que tenéis que programar.

7. Entiende el objetivo de los ejercicios

En los próximos ejercicios te vamos a guiar paso a paso para que los consigas terminar. Lo importante no es que sigas los pasos, sino que entiendas lo que estamos haciendo en cada uno de ellos y por qué.

Conclusión

Vuestro objetivo es realizar cada uno de los ejercicios de los próximos días. Para ello tendréis que:

1. Analizar lo que os estamos pidiendo en el enunciado.
2. Analizar los datos que se están recibiendo en cada una de las funciones de los `services`.
3. Enviar esos datos a vuestro propio servidor modificando el `fetch`.
4. Programar el endpoint en vuestro servidor.
5. Comprobar que todo funciona como antes.

El otro objetivo es que debéis saber en todo momento cuándo debemos trabajar en el front (la aplicación de React) y cuándo en el back (el servidor) y a quién le corresponde hacer cada cosa.

Indicadnos qué ejercicios habéis hecho

Para llevar un mejor seguimiento de cuál es vuestra evolución durante este módulo os pedimos que en el README de vuestro repo marquéis con una X los ejercicios hayáis conseguido terminar.

4.2 Express JS I

2.1 Introducción a Express JS

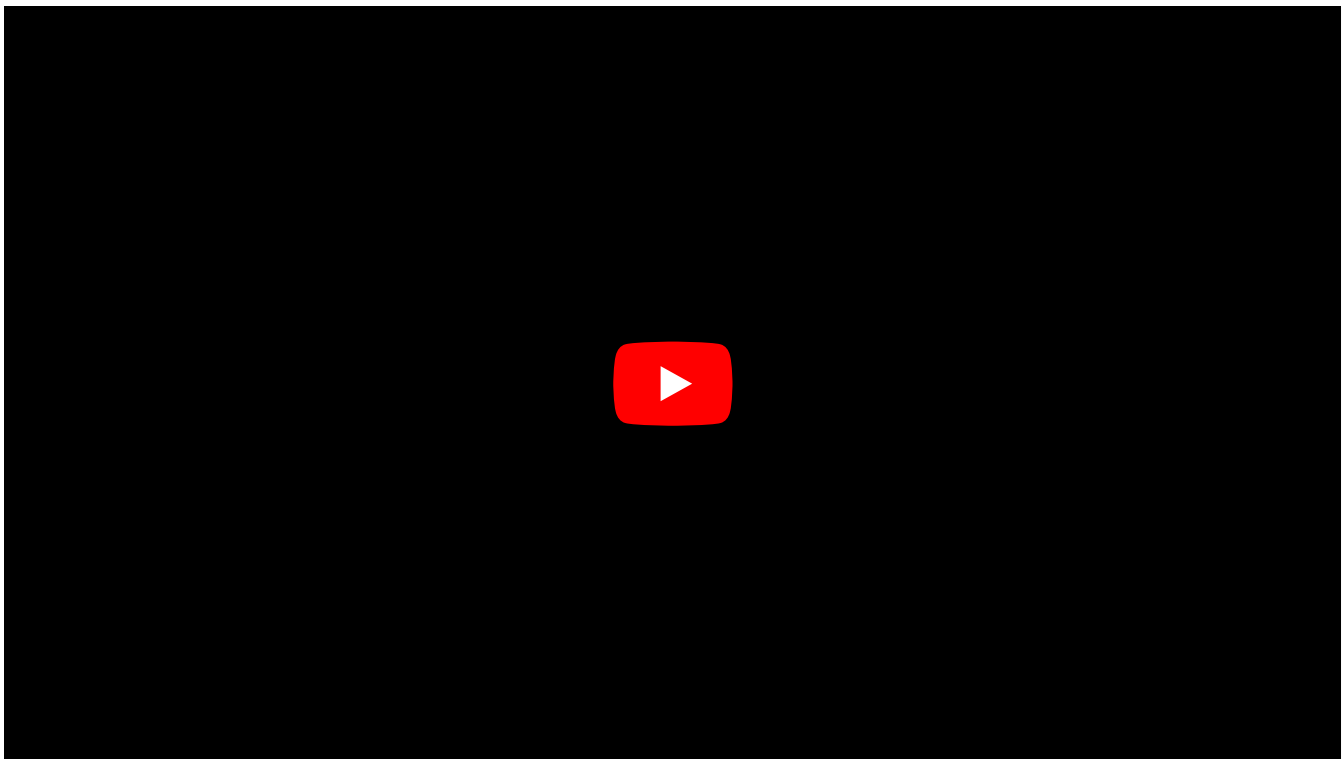
> **Nota:** esta mini lección es importante.

Ya sabemos crear aplicaciones con Node JS, pero todavía no sabemos cómo crear un servidor. Podríamos ponernos a programar un servidor, aunque es algo bastante complejo y seguramente tendríamos que escribir varios miles de líneas de código. Uff, qué pereza.

Por suerte ya hay alguien que ha creado esa programación: los creadores de Express JS. Así que vamos a aprender a utilizar el módulo de NPM de Express JS.

Qué es Express JS

Express JS es un módulo de NPM que nos facilita la vida a la hora de programar un servidor, ya que nos ayuda a escuchar las peticiones que se hacen desde el navegador al servidor. También nos ayuda a obtener los datos de dichas peticiones, procesarlos y crear una respuesta que le devolvemos al navegador.



> Ejercicio del vídeo

> **Nota:** cuando programamos una API, a los métodos que utilizamos (GET, POST, etc.) también los llamamos **verbos**. Y a las rutas (`/users` , `/new-user` , etc.) las llamamos endpoints.

Pasos para crear un servidor con Express JS

1. Crea un proyecto o repo.
2. Crea un `package.json` usando `npm init` o creándolo a mano.

3. Instala el módulo `express` ejecutando en la terminal `npm install express`.
   - Para saber que lo has hecho bien, dentro del fichero `package.json` debe aparecer `express` dentro de `dependencies`.
4. Instala el módulo `cors` ejecutando en la terminal `npm install cors`.
   - Para saber que lo has hecho bien, dentro del fichero `package.json` debe aparecer `cors` dentro de `dependencies`.
5. Crea un fichero `index.js` dentro de la carpeta `src` donde programarás el servidor.
6. Para arrancar, haz una de las siguientes cosas:
   - Ejecuta el comando `node src/index.js` o
   - Añade al `package.json` el script:

     ```
     "scripts": {
       "start": "node src/index.js"
     }
     ```

     y ejecuta el comando `npm start` para que en consola se ejecute `node src/index.js`. También puedes añadir al `package.json` los scripts:

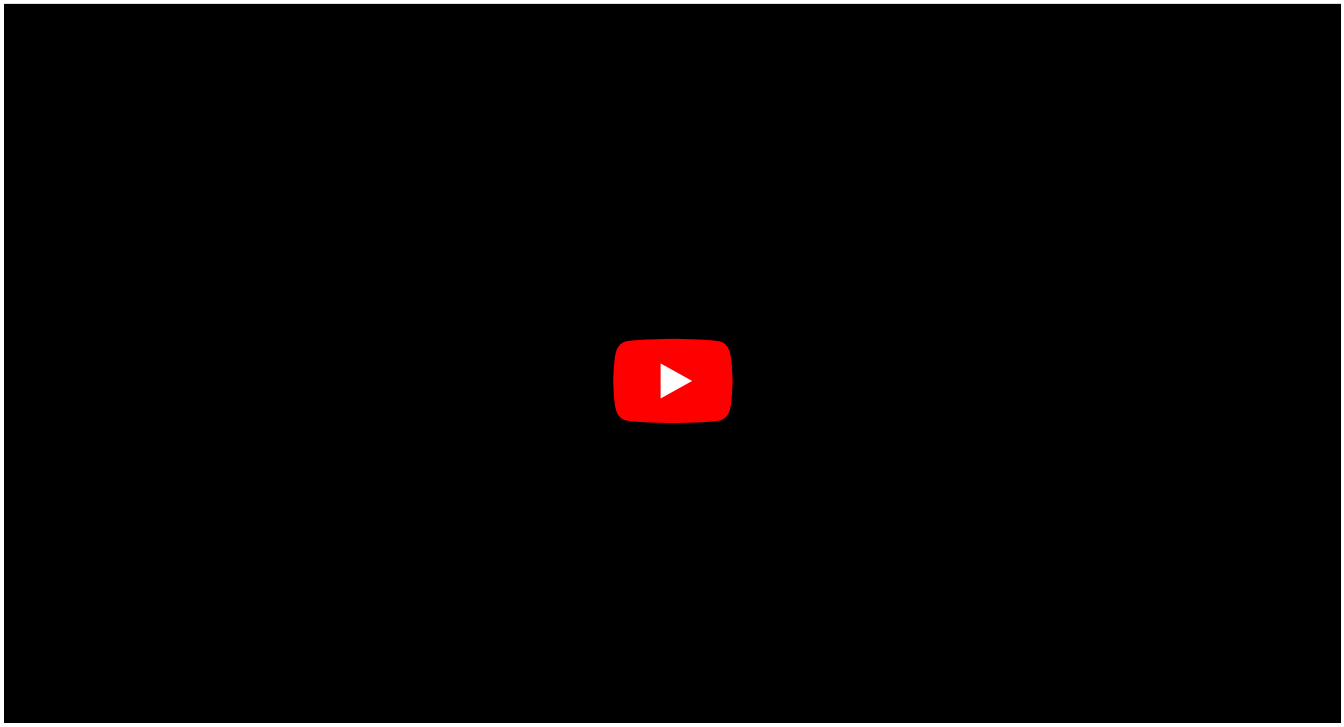
     ```
     "scripts": {
       "start": "node src/index.js",
       "dev": "nodemon src/index.js"
     }
     ```

     y ejecutar el comando `npm run dev` para que en consola se ejecute `nodemon src/index.js` y que se reinicie el servidor con cada cambio.

Errores comunes

Servidor no responde

¿Qué pasa cuando hacemos una petición al servidor y este no responde?



> Ejercicio del vídeo

Peticiones grandes

En express se utiliza el middleware body-parser para analizar los datos de las solicitudes entrantes, y también establece el tamaño predeterminado del cuerpo de la solicitud en 100 kb, de modo que cuando el tamaño del cuerpo de una solicitud exceda los 100 kb, veremos este error `Error: request entity too large`.

Para corregir este error, necesitamos aumentar el tamaño límite para procesar peticiones de esta manera:

```
server.use(express.json({limit: '25mb'}));
```

Conclusiones

Para crear un servidor con Express JS usaremos el siguiente código:

```
// Fichero src/index.js

// Importamos los dos módulos de NPM necesarios para trabajar
const express = require('express');
const cors = require('cors');

// Creamos el servidor
const server = express();

// Configuramos el servidor
server.use(cors());
server.use(express.json({limit: '25mb'}));


// Arrancamos el servidor en el puerto 3000
const serverPort = 3000;
server.listen(serverPort, () => {
  console.log(`Server listening at http://localhost:${serverPort}`);
});

// Escribimos los endpoints que queramos
server.get('/users', (req, res) => {
  const response = {
    users: [{name: 'Sofía'}, {name: 'María'}],
  };
  res.json(response);
});
```

Ejercicios

1. Crea un servidor desde cero

1. Sigue los pasos que hemos visto antes en el apartado: **Pasos para crear un servidor con Express JS**
2. Añade al fichero `index.js` las líneas de código:

```
const express = require('express');
const cors = require('cors');

const server = express();

server.use(cors());
server.use(express.json());

const serverPort = 3000;
server.listen(serverPort, () => {
  console.log(`Server listening at http://localhost:${serverPort}`);
});

server.get('/users', (req, res) => {
  const response = {
    users: [{name: 'Sofía'}, {name: 'María'}],
  };
  res.json(response);
});
```

3. Entra en `http://localhost:3000/users` desde Chrome
4. Verás que el navegador muestra el JSON con la respuesta del servidor `{ users: [...] }`.

> **Nota:** recuerda que cada vez que hagas un cambio en los ficheros de tu servidor, debes parar el servidor con `Ctrl+C` y volver a arrancarlo.

2. Cambio de endpoints y verbos

1. Descarga el ejercicio del vídeo.
2. Haz `npm install` para que NPM instale Express JS dentro de `node_modules/`.
3. Haz `npm start` para arrancar (o levantar) el servidor.
4. En el fichero `src/index.js`:
   1. Cambia la ruta `app.post('/new-user', (req, res) => {` por `app.post('/users/add', (req, res) => {`
      - ¿Qué debes cambiar en `public/main.js` para que la web siga funcionando?
   2. Cambia la ruta `app.get('/users', (req, res) => {` por `app.post('/users', (req, res) => {`
      - ¿Qué debes cambiar en `public/main.js` para que la web siga funcionando?
   3. Cambia el puerto `const serverPort = 3000;` por `const serverPort = 3500;`
      -

¿Qué debes cambiar en `public/main.js` para que la web siga funcionando?

2.2 Postman

> **Nota:** esta mini lección es la menos importante de hoy. Puedes simplemente leerla por encima.

Es muy frecuente que cuando programamos un servidor no tengamos programada la correspondiente web que va a comunicarse con este servidor.

Se nos plantea el problema de no tener una web para probar que lo que estamos haciendo funciona bien.

Para solucionar esto existe una herramienta muy útil que nos permite hacer cualquier petición a un servidor. Es como si fuera una web universal que se puede comunicar con cualquier servidor del mundo.

Esa herramienta se llama **Postman** y es muy muy útil mientras estamos programando.

Instalación de Postman

Para instalar Postman en tu ordenador:

1. Descarga e instala Postman en tu ordenador:
   - Desde Windows o Mac entra en la página de descargas de Postman.
   - Desde Ubuntu accede al instalador de aplicaciones desde el menú, busca **Postman** e instálalo.
2. Ábrelo:
   - Si al abrirlo te pide que te registres, debes saber que no hace falta. Sáltate este paso pulsando en **Skip signing in and take me straight to the app**.
3. Abre una nueva pestaña pulsando en el icono +.
4. Empieza a hacer peticiones a un servidor como no si hubiera un mañana. Recuerda que el servidor debe estar arrancado para que conteste a las peticiones.

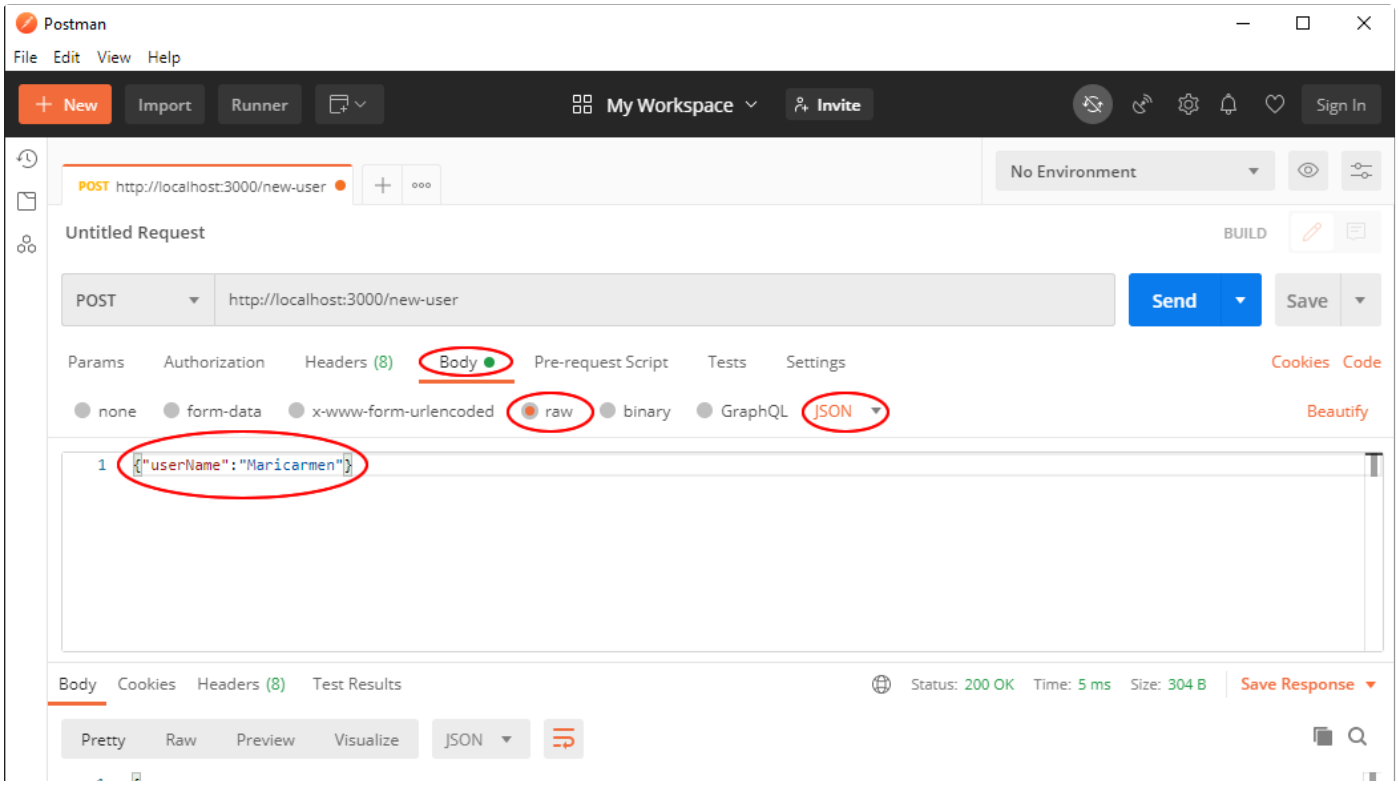
¿Cómo usar Postman?

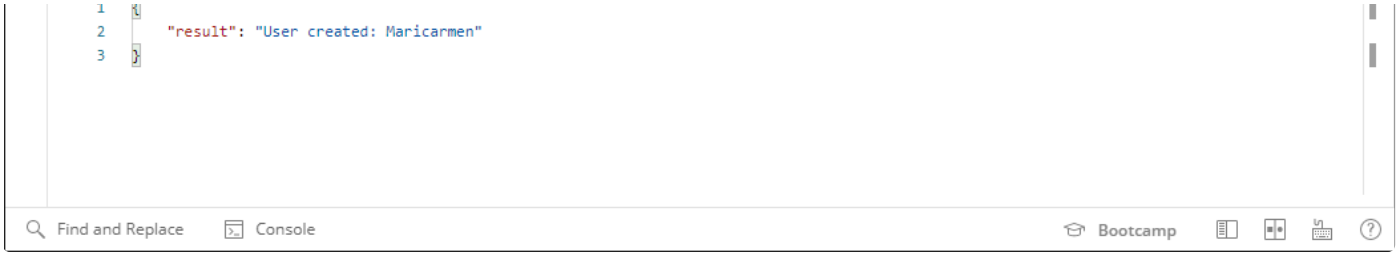
> Ejercicio del vídeo

Cómo enviar datos en una petición con Postman

Este es un pequeño recordatorio de lo que hemos visto en el vídeo anterior. Cuando hacemos una petición POST y queremos enviar datos tenemos que:

- Seleccionar el verbo, casi siempre va a ser **POST** (el verbo GET no permite enviar datos en el body).
- Poner la URL, en la imagen es http://localhost:3000/new-user.
- Entrar en la pestaña **Body**.
- Seleccionar la opción **raw**.
- Seleccionar el formato **JSON**.
- Escribir correctamente en formato JSON los datos que queremos enviar. El formato JSON requiere los nombres de las propiedades estén entre comillas dobles `"` (en la imagen `"userName"`).





Ejercicios

1. Usa Postman

1. Descarga, instala y arranca este ejercicio.
2. Abre la página http://localhost:3000.
    1. Abre Devtools > Network y mira qué peticiones se hacen cuando pulsamos en los botones de la página.
3. Realiza las mismas peticiones desde Postman y comprueba que la respuesta que muestra Postman es la misma que muestra la pestaña de Devtools > Network.

2.3 Peticiones con query params

> **Nota:** esta mini lección es importante.

Ya sabemos que nuestro trabajo cuando programamos un servidor es básicamente enviar datos desde el navegador, recibirlos en el servidor y responder con otros datos.

En esta mini lección nos vamos a centrar en enviar datos desde el navegador al servidor. Lo que responda el servidor es un problema que abordaremos en otra lección.

Partes de una petición

Toda petición a un servidor (y toda respuesta de este al navegador) está compuesta de dos partes:

- Cabecera o *header*: contiene información relativa a la petición en sí misma, como la URL, información sobre el navegador y usuaria que hace la petición, tipo de petición (GET, POST, etc.), tipos de datos que se pueden enviar y recibir (texto, HTML, JSON, etc.).
- Cuerpo o *body*: contiene información relativa al contenido o datos que se quiere enviar y recibir. Suelen ser los datos que enviamos al servidor y los datos y ficheros que recibe el navegador.

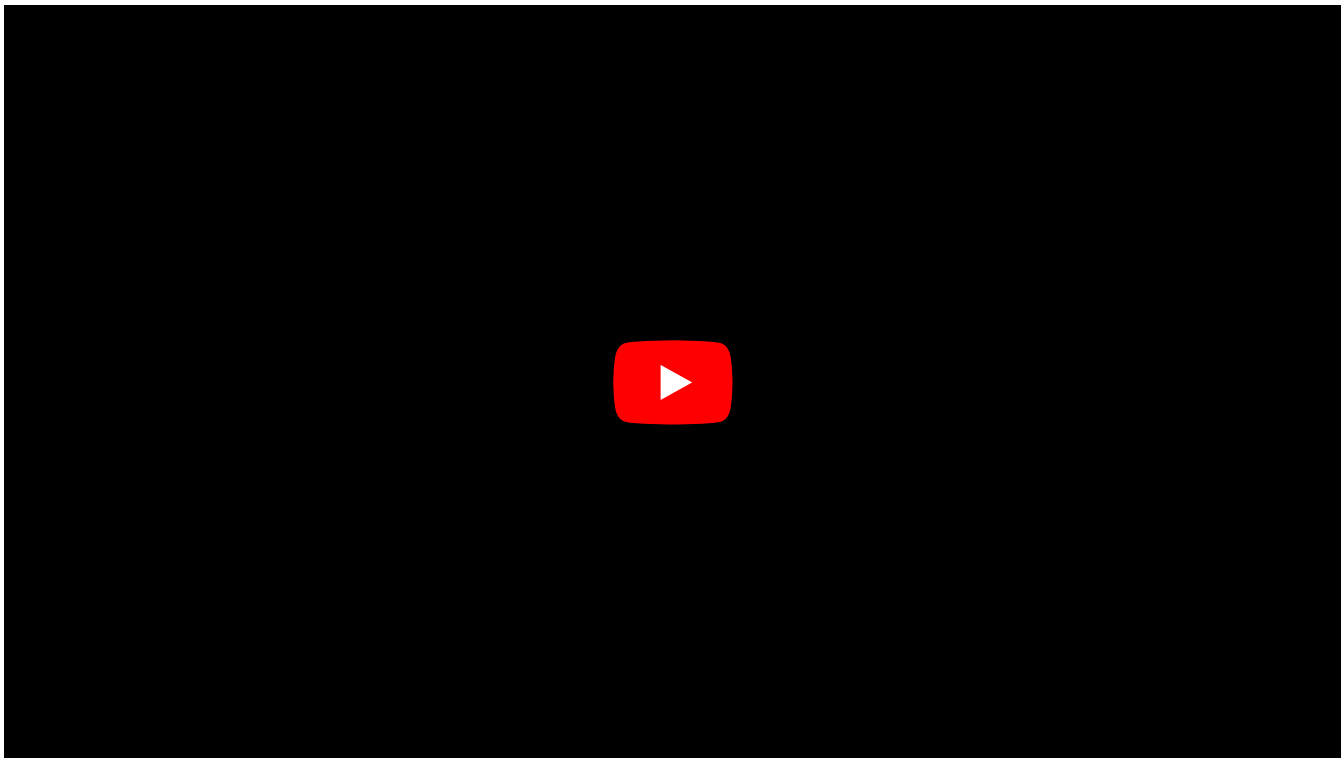
Formas de enviar datos al servidor

Hay 4 formas de enviar datos desde el navegador al servidor:

- Query params
-

- Body params
- URL params
- Header params

Con este vídeo vamos a aprender cómo se envían datos a través de **query params**. En las próximas lecciones veremos cómo se envía el resto de tipos.



> Ejercicio del vídeo

URL compartible

Puesto que los query params van al final de la URL, podemos crear una URL con query params y compartirla con quien queramos. Por ejemplo, si envías por Slack o Whatsapp la URL https://twitter.com/search?q=oidoEnAdalab&f=live, la persona que la reciba puede acceder directamente a Twitter con la búsqueda ya hecha.

Conclusiones

Para trabajar con query params debemos saber que:

- Desde el navegador:
  - Se añaden al final de la URL.
  - Empiezan con el símbolo `?`: http://localhost:3000/user?userName=maricarmen
  - Si queremos enviar varios query params los separamos con `&`: http://localhost:3000/user?userName=maricarmen&userEmail=mari@gmail.com
  - Se pueden utilizar con cualquier verbo (GET, POST, PUT, PATCH...) ya que todas las peticiones tienen URL.

- En el servidor:
  - Los datos que recibimos están en `req` porque son los datos de la petición ("request").
  - Recibimos los datos en `req.query`.
  - Todos los datos enviados por query param se reciben en el servidor como **string**.

Ejercicios

1. Filtrando usuarias por nombre

Vamos a partir del ejercicio del vídeo y a añadir una nueva funcionalidad. Ya sabemos que cuando pulsamos en el segundo botón del ejercicio, se llama al endpoint http://localhost:3000/users y el servidor nos devuelve el listado completo de usuarias.

Pues bien, queremos añadir un filtro a la web y al servidor para que este nos devuelva las usuarias filtradas por nombre. Para ello:

1. Añade un campo de filtro a la web:
   1. Edita `public/index.html` para añadir un input de texto en el segundo rectángulo.
   2. Edita `public/js/main.js` para que cuando ejecutamos `fetch('http://localhost:3000/users')` se envíe por query params un nuevo dato con el nombre `filterByName` añadiéndolo al final de la URL.
   3. Comprueba desde DevTools > Network que la llamada que estás haciendo tiene el formato correcto, es decir, la URL concatenada con el query param. Si este formato es correcto ya puedes empezar a editar el servidor.
2. Añade el filtro al servidor:
   1. Edita `src/index.js`.
   2. En el endpoint `server.get('/users', (req, res) => {...})` debes recoger el query param `filterByName` y guardarlo en una constante.
   3. En el ejercicio del vídeo estamos devolviendo todo el array de usuarias, que lo hacemos con el código:

      ```
      res.json({
        result: users,
      });
      ```

   4. Filtra el array `users` con el query param `filterByName` y guarda el array filtrado en una constante.
   5. Responde a la petición con el nuevo array filtrado.

> Recuerda que para que el array `users` contenga usuarias, hay que añadirlas a través del primer formulario de la web.

2. Filtrando usuarias por nombre e email

Partiendo del ejercicio anterior:

1. Añade un segundo campo de texto a la web para filtrar por email y envíalo también por query params al servidor.
2. Añade un segundo filtro en el servidor en la función `server.get('/users', (req, res) => {...})` para que el servidor solo devuelva las usuarias cuyo nombre contenga el texto del filtro de nombre y cuyo email contenga el texto del filtro de email.

Netflix

> **Nota:** veréis que os hemos propuesto bastantes ejercicios para cada sesión de pair programming.
>
> No hace falta hacerlos todos. Estos ejercicios están pensados para que cada día podáis hacer los primeros ejercicios aunque no hayáis terminado los últimos del día anterior. **Cada día debéis hacer los ejercicios de este día, no los que no terminasteis el día anterior.**

Ejercicios

1. Pide todas las películas

El objetivo de este ejercicio es recuperar todas las películas del catálogo de Netflix y pintarlas en la web. Para ello vamos a crear un endopoint `GET:/movies` en la API y lo vamos a llamar desde el front.

Vamos a modificar el front para que pida las películas a la API:

1. Edita el fichero `web/src/services/api-movies.js`.
2. Cambia el endpoint del fetch para que llame a:
    1. `//localhost:4000`.
    2. Usa el verbo GET. Usamos este verbo porque solo vamos a pedir datos, no vamos a modificar nada en el servidor.
    3. Usa la ruta `/movies`.
3. Para comprobar si lo has hecho bien, refresca la página http://localhost:3000 y verás en DevTools > Network que se está haciendo una petición pero que esta devuelve un error 404. Esto es porque todavía no la hemos programado en la API. Por ahora lo importante es que la petición se esté haciendo a la dirección web http://localhost:4000/movies.
4. Tenemos que hacer un último cambio. En el último `then` de este `fetch` estamos gestionando datos falsos. Borra todo el contenido de este `then` y retorna lo que retorne el servidor, para pasárselo a React; para ello pon el código:

```
.then(response => response.json())
.then(data => {
   return data;
});
```

Ahora vamos a modificar el back para que responda a la petición que acabamos de crear en el front, pero lo vamos a hacer en varias fases:

1. Edita el fichero `src/index.js`.
2. Crea un endpoint para escuchar las peticiones que acabas de programar en el front.
   - Tendrás que averiguar cuál es la ruta o endpoint y cuál es el verbo que debes poner.
3. A continuación responde a la petición con los datos:

```
{
  success: true,
  movies: [
    {
      id: '1',
      title: 'Gambita de dama',
      gender: 'Drama',
      image: 'https://via.placeholder.com/150'
    },
    {
      id: '2',
      title: 'Friends',
      gender: 'Comedia',
      image: 'https://via.placeholder.com/150'
    }
  ]
}
```

4. Para comprobar si lo has hecho bien, refresca la página http://localhost:3000 y verás en DevTools > Network que se está haciendo una petición pero que ya no devuelve el error 404. Ahora debería devolver los datos que hemos puesto en la respuesta. Además ya se deberían pintar otra vez las películas en pantalla.

Enhorabuena, has creado tu primer endpoint. Ahora vamos a limpiar un poco. No nos gusta que las películas estén puestas a pincho dentro del endpoint. Así que:

1. Crea un fichero en `src/data/movies.json` y añade el array de movies `[ { "id": 1, "title": "Gambita de dama"... }]`.

   > Ten en cuenta que para que el JSON tenga una sintaxis válida los nombres de las propiedades y los valores deben estar entre comillas dobles.

2. Importa con `require` este JSON en `src/index.js`. Hazlo en la línea 3 de este fichero. Los `require` siempre los ponemos al principio del código.
3. Dentro del endpoint que has creado retorna las películas que has importado.
   - Recuerda que para que todo funcione también debes responder con `success: true` porque es lo que está esperando React.
4. Para comprobar si lo has hecho bien, añade una nueva película a `src/data/movies.json` y refresca la página http://localhost:3000. Si se está mostrando la película que acabas de crear todo está funcionando fetén.
5. Otra forma de comprobar que todo funciona es utilizar Postman para ver si el endpoint está devolviendo

los datos que tú esperas.

2. Filtra por género

Ahora vamos a enviar el filtro de género desde el front a la API para que éste devuelva todas las películas, o solo las de drama o solo las de comedia.

1. Edita el fichero `web/src/services/api-movies.js`.
2. La función `getMoviesFromApi` ya está recibiendo por parámetro un objeto con el género seleccionado por la usuaria.
   - En la primera línea de `getMoviesFromApi` consolea lo que recibes por parámetro para saber con qué datos estás trabajando.
3. Amplía la URL del `fetch` para enviar por query params un parámetro llamado `gender` que sea igual que el valor seleccionado por la usuaria.
4. Para comprobar que lo has hecho bien, cambia el select del género y verás en DevTools > Network que se está enviando el valor seleccionado por la usuaria en la petición a través de query params.

Y ahora vamos a modificar el endpoint de la API para que filtre las películas por género:

1. Edita el fichero `src/index.js`.
2. Dentro del endpoint que has creado en el ejercicio anterior, consolea los query params que estás recibiendo.

   - Todo el código que añadas debe estar dentro del endpoint, ya que queremos que se ejecute solo cuando desde el front se llama a este endpoint.
3. Guarda el valor del query param de género en una constante, por ejemplo `const genderFilterParam = ...`.
4. Hasta ahora estamos devolviendo todas las películas que tenemos en `src/data/movies.json`. Ahora tienes que filtrar dichas películas y responder con el listado filtrado.
5. Para comprobar que lo has hecho bien, juega con el select de la página y comprueba que las películas devueltas tienen sentido. También puedes editar `src/data/movies.json` cambiando las películas y los géneros para ver que las películas mostradas en el front son las adecuadas.

> **Nota:** si no te da tiempo a terminar este ejercicio no te preocupes. No lo necesitas para seguir trabajando los próximos días.

3. Ordena por nombre

Queremos que la API devuelva las películas ordenadas por nombre de forma ascendente o descendente.

La función `getMoviesFromApi` ya está recibiendo por parámetro un objeto con el orden seleccionado por la usuaria. A partir de ahí deberás repetir los mismos pasos que hemos seguido para el filtro de género, pero esta vez para ordenar las películas por su nombre.

> **Nota:** si no te da tiempo a terminar este ejercicio no te preocupes. No lo necesitas para seguir

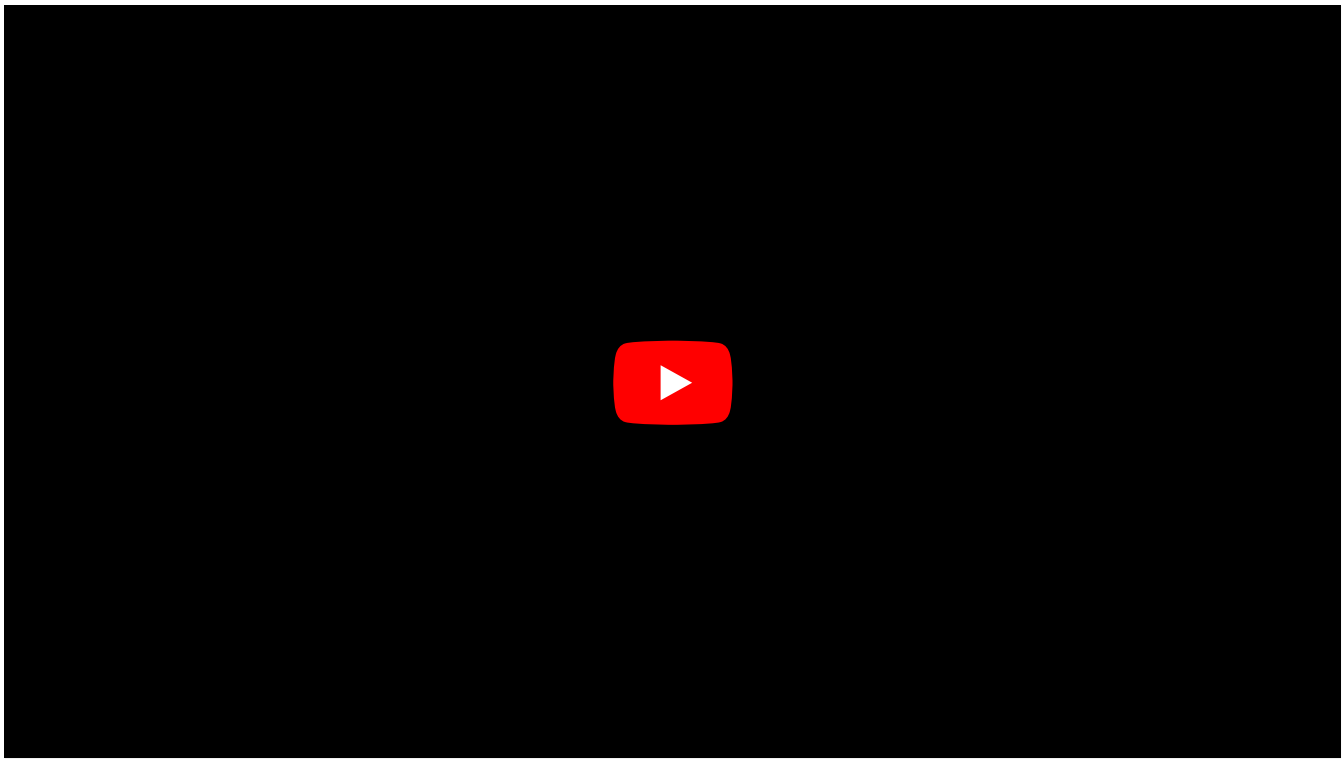
> trabajando los próximos días.

4.3 Express JS II

3.1 Peticiones con body params

> **Nota:** esta mini lección es importante.

En esta lección vamos a explicar cómo enviar datos a través de body params. Es exactamente lo mismo que enviar datos con query params, pero por otro sitio.



> Ejercicio del vídeo

Conclusiones

Para trabajar con body params debemos saber que:

- Desde el navegador:
  - Se añaden en el cuerpo del fetch.
  - Se pueden utilizar con cualquier verbo (POST, PUT, PATCH...) excepto GET.
  - Se envían en un string que contiene un JSON, por eso usamos `JSON.stringify(...)`.
  - Se envía una cabecera indicando que vamos a enviar los datos en formato JSON

`'application/json'` es decir con:

```
const bodyParams = {
  userName: 'Maricarmen',
  userEmail: 'mari@gmail.com',
};
fetch('http://localhost:3000/user', {
  method: 'POST',
  body: JSON.stringify(bodyParams),
  headers: {
    'Content-Type': 'application/json',
  },
});
```

- Como lo que estamos enviando es un objeto de JS, este puede ser todo lo complejo que queramos. Puede ser un objeto que dentro tenga un array, y dentro otro objeto y dentro strings, numbers, booleans, etc.

- En el servidor:
  - Los datos que recibimos están en `req` porque son los datos de la petición ("request").
  - Recibimos los datos en `req.body`.
  - Recuerda poner `server.use(express.json());`. Si no, el objeto `req.body` estará vacío.

Ejercicios

1. Filtrando usuarias por nombre

Vamos a partir del ejercicio del vídeo y añadir una nueva funcionalidad. Ya sabemos que cuando pulsamos en el segundo botón del ejercicio, estamos llamando al endpoint http://localhost:3000/users y el servidor nos devuelve el listado completo de usuarias.

Pues bien, queremos añadir un filtro a la web y al servidor para que el servidor nos devuelva las usuarias filtradas por nombre. Para ello:

1. Añade un campo de filtro a la web:
   1. Edita `public/index.html` para añadir un input de texto en el segundo rectángulo.
   2. Edita `public/js/main.js` para que cuando ejecutamos `fetch('http://localhost:3000/users')` se envíen con el verbo `POST`.
   3. Edita `public/js/main.js` para que se envíe por body params un nuevo dato con el nombre `filterByName`.
   4. Comprueba desde DevTools > Network > Selecciona tu petición > Subpestaña Headers (abajo del todo) que la llamada que estás haciendo tiene el formato correcto, es decir, la URL concatenada con el body param. Si este formato es correcto ya puedes empezar a editar el servidor.
2. Añade el filtro al servidor:
   1. Edita `src/index.js`.
   2. Puesto que en `main.js` has cambiado el verbo del endpoint de `GET` a `POST`, debes cambiar el verbo de `server.get('/users', (req, res) => {...})`.
   3.

3. En el endpoint `server.post('/users', (req, res) => {...})` debes recoger el body param `filterByName` y guardarlo en una constante.
4. En el ejercicio del vídeo estamos devolviendo todo el array de usuarias, lo que hacemos con el código:

   ```
   res.json({
     result: users,
   });
   ```

5. Filtra el array `users` con el query param `filterByName` y guarda el array filtrado en una constante.
6. Responde a la petición con el nuevo array filtrado.

Recuerda que para que el array `users` contenga usuarias, hay que añadirlas a través del primer formulario de la web.

2. Filtrando usuarias por nombre e email

Partiendo del ejercicio anterior:

1. Añade un segundo campo de texto a la web para filtrar por email y envíalo también por body params al servidor.
2. Añade un segundo filtro en el servidor en la función `server.post('/users', (req, res) => {...})` para que el servidor solo devuelva las usuarias cuyo nombre contenga el texto del filtro de nombre y cuyo email contenga el texto del filtro de email.

3.2 Tipos de respuestas

> **Nota:** esta mini lección es importante.

Hasta ahora hemos visto cómo puede un servidor de Express JS recoger los datos de una petición. **En esta lección vamos a ver cómo puede Express JS responder a una petición.**

En la documentación de Express JS podemos ver las diferentes formas de responder a una petición. En el siguiente vídeo vamos a ver las formas más importantes y más usadas.

> **Nota:** hasta ahora hemos utilizado `const server = express();` para que sepáis que lo que devuelve la función `express();` es un servidor. Pero normalmente todo el mundo llama a esta constante **app**. Para que os acostumbréis a verlo así, a partir de este vídeo vamos a utilizar `const app = express();`, `app.listen(...)`, `app.get()`, etc.

> Ejercicio del vídeo

Los tipos de respuestas que hemos visto en este vídeo son:

- res.json
- res.redirect
- res.sendFile
- res.download
- res.send

Y para cambiar el código de estado de la respuesta usamos:

- res.status

También te recomendamos leer más información sobre los códigos HTTP de respuesta.

Errores comunes

Responder dos veces a la misma petición:

> Ejercicio del vídeo

Más información sobre res.headerSent.

Conclusiones

Desde el servidor nos debemos preocupar de:

1. **Responder con los datos en el formato correcto:**
   - res.json: para responder con un JSON.
   - res.send: para responder con un string.
   - Visita la documentación de Express JS para aprender más métodos de respuesta.
2. **Responder con el código de respuesta correcto:**
   - Con la función res.status indicamos el código de respuesta que queremos.
   - Si no usamos `res.status`, el código por defecto con el que contestamos es el **200**.
3. **Asegurarnos de que en ningún caso estamos respondiendo más de una vez a la misma petición.**

Ejercicios

1. Respondiendo muchas cosas

Vamos a crear un servidor para diferentes tipos de respuesta:

- Cuando la usuaria haga un GET a `/response-a` debemos responder `{ result: 'ok' }`.
- Cuando la usuaria haga un GET a `/response-b` debemos responder con una página HTML en la que el `h1` ponga **Esta es una página de prueba**.
- Cuando la usuaria haga un GET a `/response-c` debemos calcular un número aleatorio entre 0 y 10 y redireccionar a:
  - Youtube si el número es par.
  - Instagram si el número es impar.
- Cuando la usuaria haga un GET a `/response-d`
  - con un query param `user=1` o `user=2` debe responder con un JSON con status 200 y respuesta `{ result: 'ok' }`.
  - si se llama a este endpoint sin query param o con un query param diferente de `user=1` o `user=2` debe responder un JSON `{ result: 'error: invalid query param' }` con el status **404**.

2. Todo el mundo usa códigos HTTP para informar sobre errores

Entra en esta página que no existe y:

- Mira qué información nos muestra la página
- Abre devtools > network, refresca la página y mira qué código de respuesta está devolviendo el servidor de Google.

Entra en esta otra página que tampoco existe y repite los pasos anteriores.

3.3 Servidor de estáticos

> **Nota:** esta mini lección es importante.

Qué es un servidor de estáticos

Un servidor de ficheros estáticos es un servidor que devuelve al navegador que los solicita **ficheros sin modificar, independientemente de quién, cuándo o desde dónde los pida**.

Por ejemplo, si una usuaria entra en la página https://adalab.es/contacto, el servidor devolverá el fichero **contacto.html**, independientemente de si la usuaria está logada o no, si se encuentra en Sevilla o en Pekín, o si accede desde un ordenador o un móvil.

Las páginas web que desarrollamos como programadoras front end las subimos a los servidores de estáticos para que las usuarias puedan entrar a ellas. Por ello, normalmente, en un servidor tenemos la programación de back end dentro de la carpeta `src/` y la programación de front end dentro de la carpeta `public/`. Por cierto, un **GitHub Pages es un ejemplo de servidor de ficheros estáticos.**

La diferencia con un servidor de ficheros dinámicos, es que el servidor de dinámicos devolverá los **ficheros modificados al vuelo**, es decir, el servidor cogerá el fichero, lo modificará añadiendo información y después lo devolverá al navegador.

Por ejemplo, si yo entro en https://www.youtube.com/feed/library, el servidor de Youtube cogerá el fichero **library.html**, le añadirá mis listas de reproducción en tiempo real y lo devolverá a mi navegador. Si entras tú, el servidor hará lo mismo pero añadiendo tus listas de reproducción. Si entra una usuaria que no esté logada, el servidor añadirá un mensaje del tipo "Identifícate para ver tus listas de reproducción".

Vamos a crear un servidor de ficheros estáticos usando express.static().

> Ejercicio del vídeo

En el vídeo anterior explicamos **`__dirname` y el módulo `path`**. Vamos a explicarlo un poco más en detalle:

- `__dirname`: es una variable global especial que **contiene en todo momento la ruta del directorio donde está el fichero que se está ejecutando actualmente**. Dicho de otra forma, si en el fichero `src/api/users.js` está escrito el código `__dirname`, esta variable valdrá `src/api/`. Pero si a la vez está escrito en `src/data/info.js`, en el fichero `info.js` valdrá `src/data`.
- Módulo `path`: este es un módulo nativo de Node JS que contiene un montón de funcionalidades para trabajar con las rutas de tu ordenador, por ejemplo:
  - Si tenemos un string con la ruta de un fichero (incluyendo carpetas, nombre de fichero y extensión) podemos obtener con `path` **la ruta de la carpeta, el nombre del fichero, la extensión del fichero...** y otras cosas.
  - En concreto, la función `path.join` nos permite unir dos o más rutas para obtener una ruta completa. Además nos permite hacerlo sin tener que preocuparnos de las barras que separan cada carpeta. Las desarrolladoras de Node JS han creado esta funcionalidad porque gestionar rutas de forma manual no es demasiado fácil. Por ejemplo:

    ```
    path.join('/foo', 'bar'); // returns '/foo/bar'
    path.join('/foo/', '/bar'); // returns '/foo/bar'
    path.join('/abuela', '/madre', '/hija', '..'); // returns '/abuela/madre'
    ```

En el vídeo hemos visto cómo obtenemos la ruta del servidor de estáticos con estas dos funcionalidades.

Ficheros por defecto: index.html

Si desde el navegador se hace una petición **sin indicar el fichero**, el servidor de estáticos buscará el fichero `index.html`, **que es el fichero por defecto**, y si existe lo devolverá. En el vídeo hemos visto que:

- Cuando se entra en la página http://localhost:3000, no estamos indicando ningún fichero en la URL. Por ello el servidor de estáticos busca en `public/` si existe el fichero `index.html`, lo encuentra y lo devuelve.
-

Cuando se entra en la página http://localhost:3000/blog, tampoco estamos indicando ningún fichero. Por ello el servidor de estáticos busca en la carpeta `public/blog/` si existe el fichero `index.html`, lo encuentra y lo devuelve.

- Cuando se entra en la página http://localhost:3000/contact.html **sí estamos indicando el fichero que queremos**. Por ello el servidor busca exactamente el fichero `contact.html` dentro de la carpeta `public/`. Lo encuentra y lo devuelve.

Esto es útil para poder hacer rutas más sencillas (o rutas amigables, llamadas así por influencia del inglés), pues para la usuaria es más fácil entrar en http://localhost/blog/ que en http://localhost:3000/blog/index.html o en http://localhost:3000/blog.html.

En el fichero `public/index.html` hemos puesto el enlace `<a href="./blog/">Ir al blog</a>` para que cuando la usuaria acceda a la página del blog vea en la barra de direcciones del navegador la ruta amigable http://localhost:3000/blog/.

Barra al final del las rutas `/`

A los servidores les da igual si en el navegador ponemos la dirección https://localhost:3000/blog o https://localhost:3000/blog/.

Los servidores interpretan en los dos casos que `blog` es una carpeta y que queremos obtener el fichero por defecto `index.html` que está dentro de esta carpeta.

Gestión de ficheros que no existen: error 404

El servidor de estáticos intenta gestionar la petición hecha desde el navegador. Si no encuentra el fichero buscado, Express JS intenta gestionarlo con los endpoints que estén declarados más abajo en el código.

Puesto que más abajo hemos puesto `app.get('*', (req, res) => { ... })`, el servidor intenta gestionarlo con este endpoint. Este endpoint tiene la ruta `'*'`. Como el asterisco significa **cualquier ruta**, el servidor gestionará todas las rutas que no hayan sido gestionadas por un endpoint anterior.

Si la petición es gestionada por `app.get('*', (req, res) => { ... })` significa que el servidor de estáticos no ha encontrado el fichero, es decir, el fichero no existe. Por ello este endpoint devuelve el fichero `404-not-found.html` con un status 404.

Siempre es conveniente poner el código `app.get('*', (req, res) => { ... })` en el `index.js` después del resto de endpoints, para que el servidor intente gestionar cada petición con los `app.get(...)` que hayamos puesto más arriba. En caso de que el servidor no pueda gestionarlo con ningún endpoint específico, lo gestionará con el endpoint genérico `app.get('*', ...)`.

> **Nota:** más info sobre el status 404.

Uso de dos servidores de estáticos a la vez

A menudo nos interesa usar dos servidores de estáticos a la vez. ¿Cuándo? Imaginemos que en la web de Adalab hay un servidor que devuelve dos tipos de páginas:

- Páginas públicas para la web normal de Adalab con `index.html`, `contact.html`, etc.

- Páginas con el temario del curso con `intro-a-html.css`, `intro-a-css.html`, `flexbox.html`, etc.

Como son páginas muy diferentes y gestionadas por equipos diferentes, hemos decidido hacer dos proyectos separados.

Para configurar nuestro servidor de forma que sirviera a ambos proyectos haríamos lo siguiente:

1. Crear una carpeta en el servidor con el nombre que queramos, por ejemplo `web/`, donde ponemos los ficheros (HTML, CSS, JS, etc.) de la parte pública.
2. Crear una carpeta en el servidor con el nombre que queramos, por ejemplo `lessons/`, donde ponemos los ficheros (HTML, CSS, JS, etc.) de las lecciones de las alumnas.
3. Añadir a nuestro `index.js`:

```
// Parte del fichero src/index.js


// Configuración del primer servidor de estáticos
const staticServerPathWeb = './web';
app.use(express.static(staticServerPathWeb));
// Configuración del segundo servidor de estáticos
const staticServerPathLessons = './lessons';
app.use(express.static(staticServerPathLessons));

// Endpoint para gestionar los errores 404
app.get('*', (req, res) => {
  // Relativo a este directorio
  const notFoundFileRelativePath = '../web/404-not-found.html';
  const notFoundFileAbsolutePath = path.join(
    __dirname,
    notFoundFileRelativePath
  );
  res.status(404).sendFile(notFoundFileAbsolutePath);
});
```

De esta manera cuando una usuaria entre en http://adalab.es/contact.html, el servidor buscará en la carpeta `./web` el fichero `contact.html`. Al encontrarlo, lo devolverá al navegador de la usuaria.

Sin embargo, cuando una usuaria entre en http://adalab.es/flexbox.html, el servidor buscará en la carpeta `./web` el fichero `flexbox.html`. Al **no** encontrarlo lo buscará en la carpeta `./lessons`. Ahora **sí** lo encontrará y lo devolverá.

Si la usuaria entra en http://adalab.es/no-existo.html, el servidor buscará el fichero `no-existo.html` en las carpetas `./web` y `./lessons`, pero no lo encontrará. En ese momento lo gestionará con el endpoint `app.get('*', ...)` y devolverá la página `web/404-not-found.html`.

Dos servidores estáticos con dos index.html

Ahora que ya sabemos crear un servidor con dos servidores estáticos, vamos a pensar qué pasaría si tenemos los siguientes ficheros:

- `./web/index.html` con un texto que pone "Bienvenida a la web de Adalab".
- `./lessons/index.html` con un texto que pone "Bienvenida al temario de Adalab".

¿Qué crees que vería la usuaria al entrar en https://adalab.es/index.html ?

¿Qué crees que vería la usuaria al entrar en https://adalab.es/index.html si en el código del servidor cambiamos el orden de las carpetas? Como en el siguiente código:

```
// Parte del fichero src/index.js

// Configuración del primer servidor de estáticos
const staticServerPathLessons = './lessons';
app.use(express.static(staticServerPathLessons));
// Configuración del segundo servidor de estáticos
const staticServerPathWeb = './web';

app.use(express.static(staticServerPathWeb));
```

En ambos casos el servidor siempre responde con el fichero `index.html` de la carpeta que esté declarada antes en el `index.js`.

Conclusiones

Para crear un servidor de estáticos debemos escribir el siguiente código:

```
const staticServerPathWeb = './public'; // En esta carpeta ponemos los ficheros estáticos
app.use(express.static(staticServerPathWeb));
```

Este código debe ir después del resto de los endpoints.

Para gestionar errores 404 debemos escribir:

```
// Endpoint para gestionar los errores 404
app.get('*', (req, res) => {
  // Relativo a este directorio
  const notFoundFileRelativePath = '../public/404-not-found.html';
  const notFoundFileAbsolutePath = path.join(
    __dirname,
    notFoundFileRelativePath
  );
  res.status(404).sendFile(notFoundFileAbsolutePath);
});
```

Este endpoint siempre, siempre, siempre debe ser el último, por detrás del resto de endpoints y de los servidores de estáticos.

Ejercicios

1. Añade una de tus páginas al servidor de estáticos

Ahora que ya sabemos montar un servidor de estáticos, qué mejor forma de celebrarlo que creando con tus propias páginas:

1. Crea un servidor de estáticos.
    1. Crea un proyecto nuevo con un servidor de Express.
    2. Añade el código de tu servidor de estáticos.
        - Configura el servidor de estáticos para que utilice la carpeta `public/`.
2. Elige una de las páginas que hayas hecho durante el curso:
    - Si eliges una página hecha con el starter kit de Adalab tienes que:
        1. En el proyecto del starter kit crea la carpeta `docs/` con `npm run docs`.
        2. Copia los ficheros de dicha carpeta `docs/`.
        3. Pega los ficheros copias en la carpeta `public/` del servidor.
    - Si eliges una página hecha con React:
        1. Abre el proyecto y ejecuta `npm run build`.
        2. Copia los ficheros de la carpeta `build/`.
        3. Pega los ficheros copiados en la carpeta `public/` del servidor.

2. Rutas amigables

Partiendo del ejercicio del vídeo cambia el código que necesites en `src/index.js` y `public/index.html` para que las usuarias puedan entrar en la dirección http://localhost:3000/contact y así hacerles la vida más cómoda.

3. Dos servidores de estáticos en un solo servidor

Nos interesa tener dos servidores de estáticos en el mismo servidor. Por ejemplo uno para servir las páginas públicas y otro para servir las páginas del área de administración.

> **Nota:** este ejercicio es interesante porque en el proyecto tendrás que hacer algo parecido.

Partiendo del ejercicio del vídeo:

1. Añade una carpeta hermana de `public/` que se llame `admin/`.
2. Añade dentro de `admin/` dos ficheros, un `index.html` y un `admin.html`. Escribe en estos ficheros un texto cualquiera.
3. Añade en `src/index.js` el siguiente código después del servidor de estáticos que ya tenemos:

```
const staticServerPathAdmin = './admin';
app.use(express.static(staticServerPathAdmin));
```

4. Arranca el servidor y:
    1. Entra en http://localhost:3000. ¿Qué fichero te está devolviendo el servidor, el

2. Entra en http://localhost:3000/admin.html y mira qué fichero está devolviendo.

Netflix

1. Servidor de estáticos para React

Hasta ahora estamos trabajando con dos proyectos: **el back de Node JS y el front de React**. A priori son independientes, pero cuando los subamos a un servidor de producción real querremos que ambos proyectos se sirvan desde el mismo sitio. Para ello vamos a meter el proyecto de React dentro del servidor Node JS para que este último sirva la página hecha en React.

Para conseguirlo vamos a crear dentro de Node JS un servidor de estáticos. Primero construiremos la versión de React para producción y lo meteremos en el servidor de Node JS:

1. Edita el fichero `package.json` de la raíz del proyecto.
    1. Añade a los `scripts` la línea:

        ```
        "publish-react": "cd web && npm run build && cd .. && rm -rf ./src/public-react && mv
        ```

    2. Analiza la línea anterior para entender lo que está haciendo:
        - En un comando de la terminal los caracteres `&&` significan "ejecuta un comando y cuando termine ejecuta el siguiente".
        - El comando `cd web && npm run build...` significa "entra en la carpeta web y dentro ejecuta el comando `npm run build` y cuando termine..."
        - Sigue analizando el comando entero para saber qué hace.
    3. Abre una terminal en la raíz del proyecto y ejecuta este `script` con `npm run publish-react`.
    4. Verás que este `script` está construyendo el proyecto de React en modo producción y lo está copiando en `src/public-react/`.

A continuación tenemos que programar un servidor de estáticos en Node JS:

1. Edita `src/index.js`.
2. En la lección de hoy hemos aprendido que con dos líneas de código podemos crear el servidor de estáticos. Cópialas en la parte inferior de `src/index.js`. Asegúrate de que la ruta del servidor de estáticos es correcta.
3. Arranca el servidor con `npm run dev`.
4. Entra en http://localhost:4000.
5. Si todo está bien deberás ver en tu navegador la página de React.

Ahora vamos a analizar lo que estamos haciendo:

- Si arrancas el proyecto de React y entras en http://localhost:3000 ves la página de React, pero en modo desarrollo. La página de React se comunica con el servidor porque esta hace los `fetch` al dominio http://localhost:4000.
- Si paras el proyecto de React, arrancas el proyecto de Node JS y entras en http://localhost:4000 también ves la página de React, pero esta vez en modo producción. En este caso no es React quien está publicando la página, sino el servidor de estáticos de Express.
- Si estás viendo la página de React en http://localhost:4000 y haces un cambio dentro de los ficheros de web, la página no se actualizará con los nuevos cambios. Esto es porque no estás en modo desarrollo sino en modo producción. Estás viendo la versión de React que creaste hace un rato con `npm run publish-react`.

> **Nota:** cada vez que hagas un cambio en el proyecto de React y quieras subirlo a producción tendrás que ejecutar otra vez la línea `npm run publish-react` para volver a generarlo en modo producción y pushearlo. Este es el equivalente a `npm run docs` que hacíamos en los proyectos que trabajan con el starter kit.

> **Nota**: aunque hemos creado un servidor de estáticos para ver la página de React hemos comentado que esa página tendrá la versión de la última vez que se hizo `npm run publish-react`. Por ello te recomendamos que mientras sigas desarrollando arranques el proyecto de Node JS y el de React a la vez y que la pruebes desde http://localhost:3000, para que ambos siempre estén levantados con la última versión de tu código.

---

2. Servidor de estáticos para las fotos

En los ejercicios anteriores hemos creado el endpoint `GET:/movies` para servir las películas a través de la API. Estos datos los tenemos en el servidor, por ello cuando estamos programando la web de React no sabemos qué películas se mostrarán en la página.

Ahora vamos a crear otro servidor de estáticos para servir más ficheros. ¿Cuáles? las fotos de las películas:

1. Crea la carpeta `src/public-movies-images/`.
2. Añade dos imágenes que se llamen `gambita-de-dama.jpg` y `friends.jpg` a la carpeta anterior.
3. Al final del fichero `src/index.js` añade otra vez las dos líneas que nos permiten crear otro servidor de estáticos, pero esta vez pon la ruta correcta para que apunte a la carpeta `src/public-movies-images/`.
4. Si todo está fetén deberías entrar en http://localhost:4000/gambita-de-dama.jpg y http://localhost:4000/friends.jpg y ver correctamente estas dos imágenes.
    - Lo importante de este ejercicio es que entiendas por qué estas son las rutas correctas.

Pues muy bien, ya tenemos otro servidor de estáticos para servir las imágenes de las películas. Ahora nos falta el último paso: **modificar los datos de las películas para que muestren las imágenes**.

1. Edita el fichero `src/data/movies.json`.
2. Edita las imágenes de las películas para que tengan la ruta correcta.
   - Las rutas correctas son //localhost:4000/gambita-de-dama.jpg y //localhost:4000/friends.jpg.
   - Si quieres saber por qué las rutas empiezan por `//` lee el apartado Mixed Content: HTTP vs HTTPS cuando tengas un rato.
3. Comprueba que todo está correcto haciendo las siguientes acciones:
   1. Arranca el servidor y entra en http://localhost:4000 y deberías ver las películas con las imágenes (modo producción).
   2. Y arranca el servidor y el proyecto de React y entra http://localhost:3000; deberías ver las películas con las imágenes (modo desarrollo).

> Si no te da tiempo a terminar este ejercicio no te preocupes. No lo necesitas para seguir trabajando los próximos días.

---

3. Peticiones POST con body params

Vamos a cambiar de tercio. Ahora vamos a crear el endpoint que nos permite hacer login en la web:

1. Edita la función `sendLoginToApi` que está en `web/src/services/api-user.js`.
2. Cambia la ruta del `fetch` para que haga una llamada al dominio http://localhost:4000 y a la ruta `/login`.
3. Cambia el `fetch` para que use el verbo POST.
4. La función `sendLoginToApi` ya está recibiendo por parámetro el email y contraseña de la usuaria. Añade al `fetch` estos datos para que se envíen como body params.
5. Tenemos que hacer un último cambio. En el último `then` de este `fetch` estamos gestionando datos fake. Borra todo el contenido de este `then` y retorna lo que retorne el servidor, para pasárselo a React; para ello pon el código:

   ```
   .then(response => response.json())
   .then(data => {
      return data;
   });
   ```

6. Si todo está bien hecho, cuando envíes el formulario de login verás que en DevTools > Network > Petición enviada > Headers se está haciendo una petición a http://locahost:4000/ con el verbo POST y en la parte de abajo se están enviando el email y la contraseña.

Seguramente la petición anterior te estará devolviendo un error 404. Esto es porque nos falta crear el endpoint en Express JS. Vamos a ello:

1. Crea el fichero `src/data/users.json` con los datos:

   ```
   ```

```
[ {
    "id": "1",
    "email": "lore@gmail.com",
    "password": "12345678",
    "name": "La Lore"
  },
  {
    "id": "2",
    "email": "macu@hotmail.com",
    "password": "12345678",
    "name": "La Macu"
  }
]
```

2. Importa este fichero al principio de `src/index.js` y guárdalo en la constante `users`.
3. En `src/index.js` después del endpoint de `GET:/movies` añade otro endpoint con la ruta y el verbo correctos.
4. Dentro del endpoint consolea los body params para ver que todo está ok. Envía el formulario de login desde React para ver qué se está consoleando.
5. Dentro de este endpoint debes buscar con un `find`, en el array de usuarias que has importado, la usuaria que tenga el mismo email y contraseña que estás recibiendo por body params:
   - Si la usuaria existe responde a la petición con:

```
{
  "success": true,
  "userId": "id_de_la_usuaria_encontrada"
}
```

   - Si la usuaria no existe responde a la petición con:

```
{
  "success": false,
  "errorMessage": "Usuaria/o no encontrada/o"
}
```

6. Para comprobar que todo ha ido como la seda envía el formulario de login desde React con los datos de una usuaria que no existe y vuelve a hacerlo con los datos de una usuaria que sí existe.

> **Nota**: responder a las peticiones añadiendo `"success": true / false` no es obligatorio. En este ejercicio hemos decidido hacerlo así. En otro proyecto tú decidirás qué quieres responder para que la comunicación entre el back y el front sea tan rica como quieras.

---

4. Mantén logada a la usuaria

Con el ejercicio anterior verás que cuando una usuaria se identifica correctamente en el back, este devuelve el id de la usuaria a React, React lo guarda en el estado y la página funciona bien. Si refrescas la

página React resetea el estado y por ello ya no guarda el `userId` y la página aparece como si la usuaria

Para conseguir que tras un refresco la usuaria siga logada lo que tenemos que hacer es muy sencillo:

1. Edita el fichero `web/components/App.js`.
2. Usa un `useEffect` para que, cada vez que cambie el valor de `userId`, se guarde en el local storage.

3. Haz que al arrancar la página se recupere el `userId` desde el local storage y se guarde en el estado de React.
4. Para comprobar que lo has hecho bien, identifícate en la web con una usuaria válida y refresca la página. Si sigue logada todo está bien.

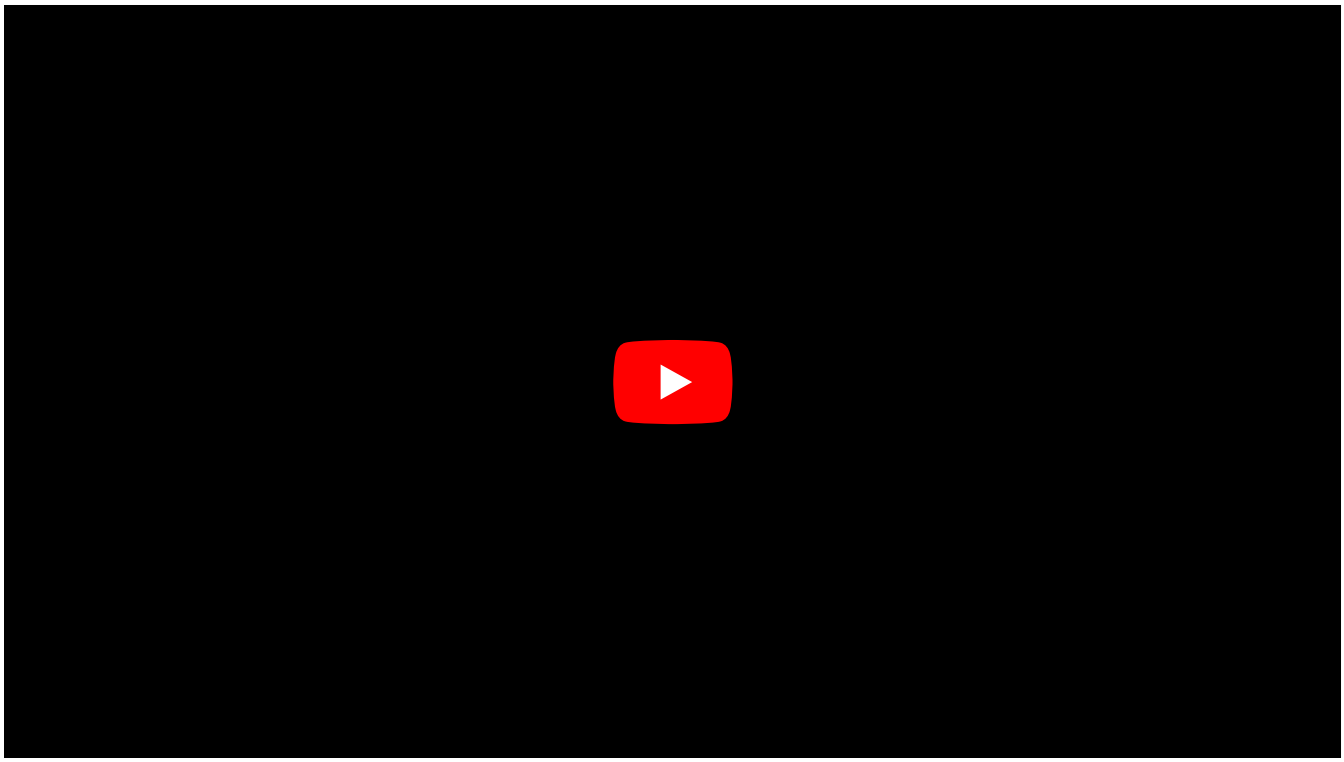
> **Nota:** dispones de un servicio ya programado en React para trabajar con el local storage.

4.4 Express JS III

4.1 Peticiones con URL params

> **Nota:** esta mini lección es importante.

En esta lección vamos a explicar cómo enviar datos a través de URL params. Es exactamente lo mismo que enviar datos con query params o body params, pero por otro sitio.



> Ejercicio del vídeo

Múltiples URL params

¿Crees que un endpoint puede tener varios URL params? Claro que sí, guapi.

Si creáramos una tienda online, lo normal sería crear los siguientes endpoints:

- http://localhost:3000/users/123: que devolvería un **objeto con la información de la usuaria** que tenga el id **123**.
- http://localhost:3000/users/123/orders: que devolvería un **array con los pedidos de la usuaria** que tenga el id **123**.
- http://localhost:3000/users/123/orders/456: que devolvería un **objeto con la información del pedido** que tenga el id **456** de la usuaria con el id **123**.
- Y lo mismo para otros endpoints que devuelvan el listado de direcciones postales, el listado de métodos de pago, etc.

Pues bien, si en el servidor creamos un endpoint que sea:

- http://localhost:3000/users/:userId/orders/:orderId

En `req.params` Express JS nos metería el objeto:

```
{
  userId: 123,
  orderId: 456
}
```

Conclusiones

Para trabajar con URL params debemos saber que:

- Desde el navegador:
  - Como van en la URL y todas las peticiones tienen URL, se pueden utilizar con cualquier verbo (GET, POST, PUT, PATCH...).
  - Se pueden usar para hacer peticiones desde la barra de direcciones del navegador o a través de un `fetch`.
- En el servidor:
  - Recibimos los datos en `req.params`.
  - Todos los datos enviados por URL param se reciben en el servidor como **string**.

Ejercicios

1. Endpoint para devolver un pedido

Vamos a hacer un servidor que, cuando se le haga una petición a la URL http://localhost:3000/users/123/orders/456, la gestione y devuelva algo. ¿Qué tiene que devolver? Da igual, porque aquí nos interesa la parte de la petición, no la de la respuesta.

1. Crea un nuevo proyecto con un servidor de Express JS.
2. En `src/index.js` crea un endpoint de tipo GET que sea capaz de atender la petición http://localhost:3000/users/123/orders/456 y consolear en la terminal los parámetros de la URL.

> **Nota:** Usa Postman y así no perderás tiempo en montar una web.

2. Troceando a Rick y Morty

Vamos a crear una API que devuelva los datos de Rick y Morty que ya conocemos, pero separados en varios endpoints diferentes.

Cuando hicimos el proyecto de Rick y Morty usamos un único endpoint que devuelve 10 personajes. Partiendo de esta información, queremos crear varios endpoints y que cada uno devuelva la información de un único personaje, como por ejemplo:

- https://rickandmortyapi.com/api/character/1 devuelve solo los datos de Rick Sánchez.
- https://rickandmortyapi.com/api/character/2 devuelve solo los datos de Morty Smith.

En los endpoints anteriores, ¿sabrías decir cuáles son los URL params?

Sigue los siguientes pasos para conseguirlo:

1. Crea un nuevo proyecto con un servidor de Express JS.
2. Crea el fichero `src/data/rick-and-morty.json` y copia dentro los datos de Rick y Morty desde aquí.
3. Importa dentro de `src/index.js` el fichero `rick-and-morty.json` para poder usar los datos.
4. Crea un endpoint para recuperar los datos de un personaje:
   - El endpoint debe ser GET, con la ruta `http://localhost:3000/users/1/` y tiene que devolver:

```
{
    "id": 1,
    "name": "Rick Sanchez",

    "status": "Alive"
    ...
}
```

   > Recuerda que los datos que vienen dentro de `req.params` siembre son string, pero el `"id": 1` es un número. ¿Sabes qué tienes que hacer para compararlos?
5. Crea otro endpoint para obtener el listado de episodios de un personaje:

- También debe ser de tipo GET, con la URL `http://localhost:3000/users/1/episodes` y tiene que devolver:

```
[
  "https://rickandmortyapi.com/api/episode/1",
  "https://rickandmortyapi.com/api/episode/2",
  "https://rickandmortyapi.com/api/episode/3",
  ...
]
```

> **Nota:** Usa Postman y así no perderás tiempo en montar una web.

Si vemos la documentación de la API de Rick y Morty nos podemos imaginar cómo están programados cada uno de los endpoints:

- Obtener un personaje
- Obtener un episodio
- Obtener una localización

4.2 Peticiones con header params

> **Nota:** esta mini lección es la menos importante de hoy, pero también es importante.

La cuarta y última forma de enviar datos al servidor desde nuestro navegador es con **header params**. Si has llegado hasta aquí ya te podrás imaginar que es muy parecido a enviar query, body o URL params.

Características de los header params

- Desde el navegador:
  - Se pueden enviar en todos los tipos de peticiones (GET, POST, PUT, PATCH...).
  - Al igual que los body params, se tienen que enviar **siempre desde un `fetch`**, no los podemos indicar en la barra de direcciones del navegador como los query o URL params.
  - Se envían añadiendo un objeto `headers` al fetch, como se puede ver en el siguiente código:

```
fetch('http://localhost:3000/ruta-del-endpoint', {
    method: 'POST', // o GET, PUT, PATCH...
    headers: {

      unParametroEnLaCabecera: 'Hola mundo',
      'otro-parametro-de-la-cabecera': 123,
      otroParametroMas: 'Soy un dato'
    }
  })
  .then(response => response.json())
  .then(data => {...});
```

- En el servidor:
  - Recibimos los datos en `req.headers`, que es un objeto.
  - También podemos acceder a un header param a través del método `req.header('nombre-de-header-param')` que nos devuelve el valor de un header param.
  - Todos los datos enviados por header param se reciben en el servidor como **string**, aunque desde el `fetch` los hayamos enviado como número u otro tipo de dato.
  - Todos los nombres de propiedades del objeto `req.headers` nos llegan al servidor en minúsculas.

Los headers params llegan en minúsculas

Una curiosidad de los headers params es que **sus nombres llegan al servidor en minúsculas**. Insistimos en esto porque es algo poco usual. Vamos, que es algo rarito.

Si desde el front enviamos:

```
fetch('http://localhost:3000/ruta-del-endpoint', {
  method: 'POST', // o GET, PUT, PATCH...
  headers: {
    PARAMETRO_EN_MAYUSCULAS: '1',
    parametro_en_minusculas: '2',
    parametroEnCamelCase: '3',
    'Parametro-Separado-Con-Guiones': '4',
  },
});
```

En el objeto `req.headers` del servidor recibiremos todos los nombres de propiedad en minúsculas:

```
{
  parametro_en_mayusculas: '1',
  parametro_en_minusculas: '2',
  parametroencamelcase: '3',
  'parametro-separado-con-guiones': '4'
}
```

Header params del navegador

Cuando enviamos una petición desde el navegador al servidor, ya sea desde un `fetch` o desde la barra de direcciones del navegador, **el navegador siempre envía sus propios headers params**.

El navegador necesita enviar su propia información al servidor para decirle cosas como:

- Qué tipo, versión, modelo... de navegador es.
- Qué idiomas (español, inglés, etc.) entiende la usuaria: los idiomas configurados en su navegador u ordenador.
- Qué tipos de formato de datos entiende (texto, JSON, datos binarios, etc.)

- Cuál es la URL actual del navegador.
- Temas de caché: durante cuánto tiempo va el navegador a almacenar la respuesta del servidor, etc.

User agent

Por todo esto, cuando nosotras enviamos header params al servidor, en el objeto `req.headers` hay muchos más datos.

Uno de los parámetros más importantes que envía el navegador es el **`user-agent`**, en el que especifica el modelo de navegador (Chrome, Safari, Firefox, Internet Explorer, Edge...). En el servidor se utiliza este parámetro para:

- Recoger estadísticas de las usuarias.
- Adaptar la respuesta que debe devolver al navegador.

Por ejemplo, si una usuaria entra a una página web desde Internet Explorer 6 (un navegador del que solo han oído hablar las más viejas del lugar), algún servidor devolverá un HTML con un mensaje del tipo "Nuestra página es muy moderna y no está preparada para navegadores tan viejunos. Por favor, actualícese".

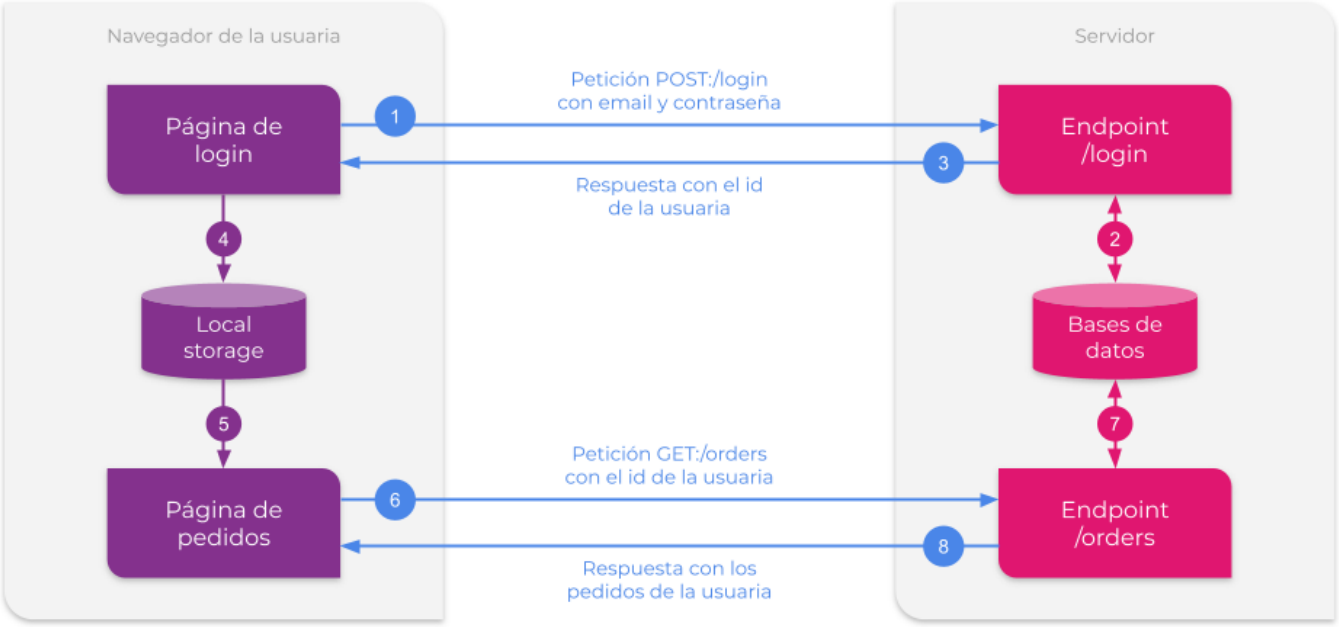
Por cierto: en algunas entrevistas técnicas se suele preguntar qué es el User Agent.

Identificación de la usuaria

Te preguntarás para qué se usan los header params. Ahora vamos a explicar un uso bastante común en proyectos reales, que seguramente te tocará programar cuando estés trabajando en una aplicación de una empresa.

Normalmente cuando una usuaria se registra o identifica en una página **las programadoras intercambiamos información entre navegador y servidor mediante los siguientes pasos**:

Envío del id de la usuaria por header params



1. La usuaria rellena el formulario de login. Desde el navegador enviamos una petición al servidor con:
    - El email y la contraseña (por body params)
    - El verbo POST
    - El endpoint (suele ser `/login`)
2. En el servidor escuchamos y atendemos esta petición, buscamos en la base de datos a la usuaria con ese email y esa contraseña.
3. Respondemos al navegador con un JSON que contiene **el id de la usuaria**.
4. Con el JS del navegador recibimos la respuesta y **almacenamos en el local storage el id de la usuaria**.
5. Un rato después, cuando la usuaria navega y entra, por ejemplo, en su página de pedidos, desde el navegador hacemos un fetch para traernos estos datos de la usuaria. Para ello lo primero que hacemos es recuperar el **id de la usuaria** desde el local storage.
6. A continuación enviamos una nueva petición al servidor para recuperar los pedidos de la usuaria con:
    - El verbo GET
    - El endpoint `/orders`
    - **El id de la usuaria en el header.**
    - Otros datos en query params o URL params, como el orden en el que queremos obtener los pedidos.
7. En el servidor atendemos esta nueva petición sabiendo qué petición es, qué datos recibe y **el id de la usuaria** que está haciendo la petición. Con este id podemos obtener sus pedidos desde la base de datos.
8. Una vez obtenidos los pedidos de la usuaria en el servidor los devolvemos al navegador para que este los pinte.

Resumen: **al enviar el id de la usuaria por header params, en el servidor sabemos qué usuaria es**.

> Si te preguntas por qué guardamos el **id de la usuaria en el local storage**, es porque queremos seguir teniendo identificada a la usuaria en el navegador, aunque ella lo cierre y lo vuelva a abrir.

Conclusiones

Para trabajar con header params debemos saber que:

- Desde el navegador:
    - Se pueden enviar en todos los tipos de peticiones (GET, POST, PUT, PATCH...).
    - Se tienen que enviar siempre desde un `fetch`.
    - Se envían añadiendo un objeto `headers` al fetch.
- En el servidor:
    - Recibimos los datos en el objeto `req.headers` o a través del método `req.header('nombre-de-header-param')`.
    - Todos los datos enviados por header param se reciben en el servidor como **string**.

- Todos los nombres de propiedades del objeto `req.headers` nos llegan al servidor en **minúsculas**.

Ejercicios

1. Investiga cómo recibir header params

1. Descarga, instala y arranca este ejercicio.
2. Entra en http://localhost:3000 y pulsa en el botón.
3. Comprueba qué datos estás recibiendo en el servidor.
4. Abre el fichero `public/js/main.js` y mira qué datos se están enviando en la cabecera.
    1. Cambia los datos que se están enviando para enviar otro tipo de datos, como números enteros, decimales, booleanos, objetos...

2. User Agent

1. Partiendo del ejercicio anterior, ¿puedes averiguar el modelo de tu navegador?
2. Haz una nueva petición al servidor utilizando Postman y mira qué valor tiene el header param `user-agent`.

4.3 Motores de plantillas

> **Nota:** esta mini lección es importante.

Qué es un servidor de ficheros dinámicos

Un servidor de dinámicos es un servidor que devuelve al navegador **el mismo fichero pero personalizado, con pequeñas modificaciones, con diferentes datos** en función de qué usuaria lo está pidiendo o en función de a qué página se está accediendo.

Si entramos en https://www.filmaffinity.com/es/film999902.html y en https://www.filmaffinity.com/es/film942734.html, vemos que la página es la misma pero cambian los datos e imágenes que se muestran.

No tiene sentido que las maquetadoras de FilmAffinity maqueten una a una todas las películas del mundo. Lo que se hace es maquetar una sola película y convertirla en una plantilla. Cuando el servidor tiene que devolver una película, coge la plantilla, le mete sus datos y la devuelve. Esto es una tarea del motor de plantillas.

Qué es un motor de plantillas

Un motor de plantillas, en inglés **template engines**, es la funcionalidad que:

1. Está **esperando** a que desde el navegador se le solicite un fichero (con un endpoint). Cuando esto sucede, el endpoint:
    1. Coge una plantilla.
    2. Obtiene los datos, generalmente desde una base de datos. En esta lección todavía no sabemos cómo trabajar con bases de datos, así que vamos a obtenerlos de un JSON.
    3. Mete los datos en la plantilla. A este proceso se le llama **renderizado**.
    4. Y devuelve la plantilla personalizada al navegador para que la usuaria la visualice.

Generalmente los ficheros que servimos desde un servidor de dinámicos son HTMLs y PDF, es decir, ficheros que tienen contenido para una usuaria. No es habitual servir ficheros CSS diferentes para cada usuaria.

Motores de plantillas en Express JS

Express JS divide la funcionalidad de los motores de plantillas en **dos partes**:

1. Escuchar la petición desde el servidor y responderla. Esta es una tarea de Express JS, que:
    - Escucha la petición con un endpoint del tipo `app.get('/ruta-del-endpoint, (res, req) => { ... })`.
    - Responde a la petición con `res.render(template, data);`.
    - Más información sobre el método res.render aquí.
2. Renderizar de la plantilla:
    - Este es el proceso que hay entre el escuchado de la petición y la respuesta.
    - Express JS **no** se encarga del renderizado.
    - Existen muchos motores de plantillas que funcionan sobre Express JS y que se encargan del renderizado.
        - Estos motores de plantillas los crean grupos de desarrolladoras y los suben a NPM para que el resto los usemos.
        - Nosotras debemos elegir uno, instalarlo en nuestro proyecto, configurarlo y usarlo.
        - En este módulo vamos a usar EJS.

Motores de plantilla

Ejemplo básico

> Ejercicio del vídeo

En el vídeo hemos visto:

- Para instalar EJS: `npm install ejs`.
- Para configurar EJS: `app.set('view engine', 'ejs');`.
- Tenemos que crear nuestras plantillas en la raíz del proyecto, en la carpeta `views/`, porque así nos lo pide EJS.
- Podemos crear las carpetas que queramos dentro de `views/`.
- Es nuestra responsabilidad poner bien la ruta de las carpetas en `res.render('pages/film', filmData)`. No hace falta poner la extensión `.ejs`.
- Es nuestra responsabilidad poner bien la ruta de los partials dentro de las plantillas, escribiendo `<%- include('../partials/header'); %>`.
- La sintaxis para añadir un partial es `<%- include('ruta-relativa-del-partial'); %>`. No hace falta poner la extensión `.ejs`.
- La sintaxis para añadir un dato a la plantilla es `<%= nombreDeMiVariableOPropiedad %>`.

Estilos CSS en los motores de plantilla

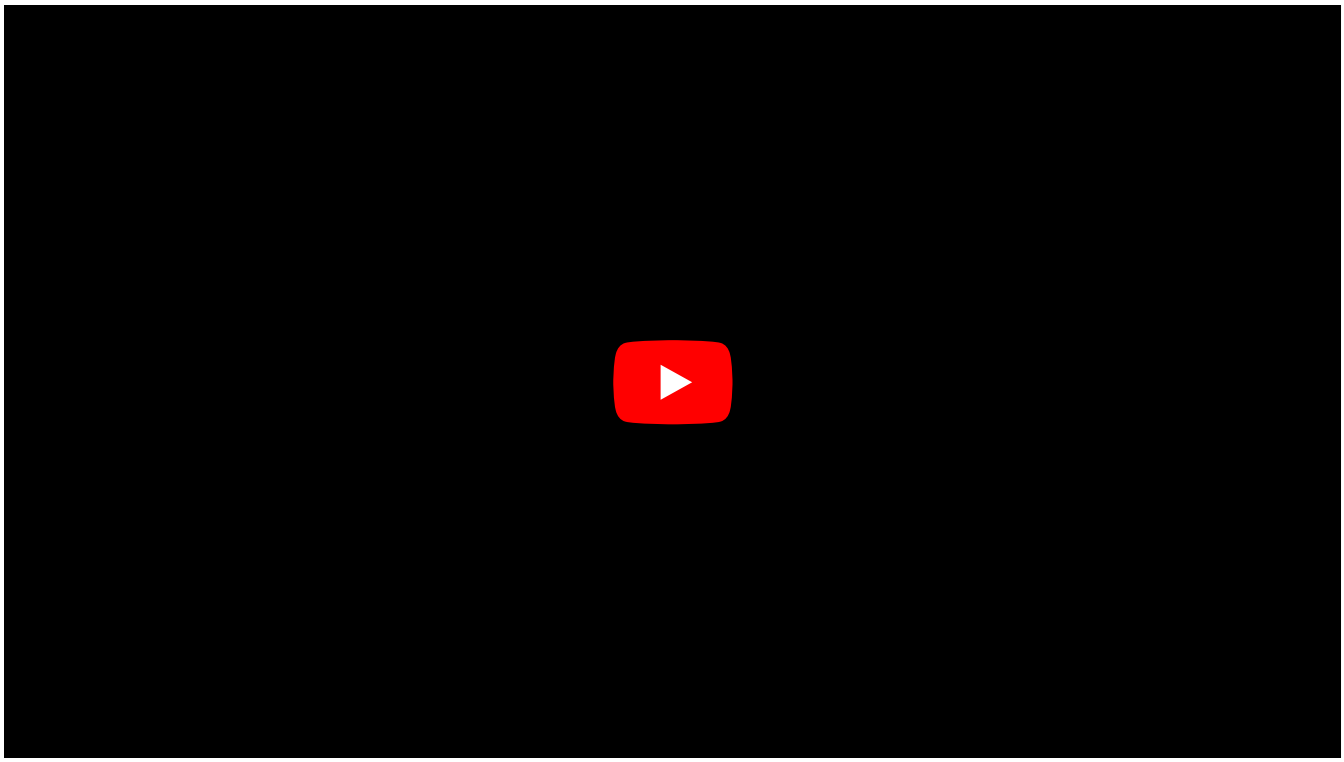
De igual manera podemos incluir estilos CSS en nuestras plantillas, para ello podemos crear un servidor de estáticos para los estilos css e incluirlos en nuestra plantilla.

1. Crea un servidor de estáticos para los estilos en tu servidor:
   - Crea el fichero `main.css` de estilos en la carpeta src del servidor `src/public-css/`.
   - Configura el servidor de estáticos en `index.js` para que esté disponible el archivo css.

     ```
     const pathServerPublicStyles = './src/public-css';
     server.use(express.static(pathServerPublicStyles));
     ```
2. Incluye el fichero `main.css` en la plantilla, presta mucha atención a la ruta del css. En la plantilla de la carpeta de views, quedaría asi:

   ```
   <link rel="stylesheet" href="/main.css" />
   ```

Ejemplo con condicionales



> Ejercicio del vídeo

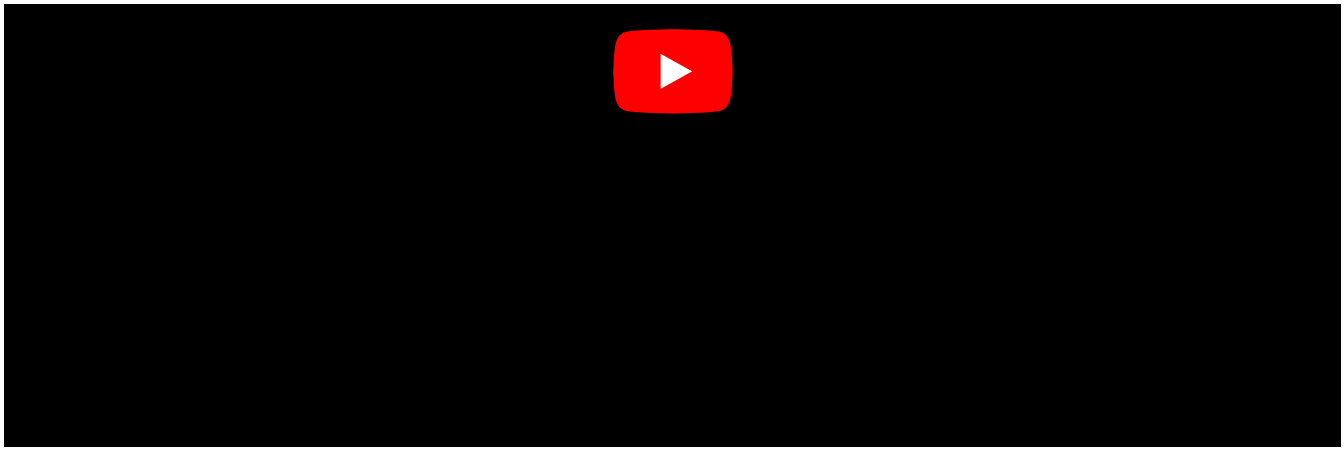
En el vídeo hemos visto que:

- La sintaxis para añadir JS que no sea un partial o para pintar en la plantilla es con `<% ... %>`. Por ejemplo:

```
<% if (country !== "") { %>
<li>País: <%= country %></li>
<% } %>
```

- A EJS no le gustan las comparaciones con `undefined`, por lo que `<% if (country !== undefined) { %>` no le gusta.
- Es mejor asegurarnos de que existen todas las variables que va a utilizar EJS antes de llamar al renderizado. Por ello en `src/index.js` nos aseguramos de que la variable existe, poniendo `filmData.country = filmData.country || '';`

Ejemplo con bucles



> Ejercicio del vídeo

En el vídeo hemos visto que:

- La sintaxis para añadir JS que no sea un partial o para pintar en la plantilla es con `<% ... %>`. Por ejemplo:

```
<% awards.forEach(function(award) { %>
<li><%= award.year %>: <%= award.info %>.</li>
<% }) %>
```

- Dentro del bucle tenemos disponibles las variables que estamos creando en el bucle. Como aquí hemos creado la variable `award` en `<% awards.forEach(function(award) { %>`, dentro podemos usarla poniendo `<%= award.info %>`.
- Es mejor pasarle los datos limpios a la plantilla para que esta se lo más sencilla posible. Por ejemplo, si queremos pintar el listado de premios, pero filtrándo por los premios Goya, tendríamos dos opciones:
  - Pasar a la plantilla el listado completo de premios y el criterio de filtrado, y que sea la plantilla la que primero filtre y luego itere para pintar, o
  - Filtrar en `src/index.js` y pasarle el array de premios ya filtrados a la plantilla. Esto es mucho más limpio y legible. Es lo que preferimos.

Múltiples plantillas

En un motor de plantillas puede haber muchas plantillas. En FilmAffinity tienen plantillas para las películas, dirección, actrices y actores, géneros...

Para crear una nueva plantilla, por ejemplo de directoras, en nuestro servidor lo único que habría que hacer es:

1. Crear una plantilla en `views/pages/director.ejs`.
2. Crear un endpoint para escuchar una petición desde el navegador, por ejemplo: `app.get('/es/director/:directorName')` y dentro de este endpoint:
   1. Obtener los datos de la directora, por ejemplo `const directorData = ...;`.
   2. Renderizar y responder con `res.render('director', directorData);`.

Conclusiones

Para crear un servidor de ficheros dinámicos necesitamos crear un endpoint con Express JS y dentro utilizar un motor de plantillas.

Para utilizar un motor de plantillas necesitamos:

1. Instalarlo:
   ```
   npm install ejs
   ```
2. Configurarlo:
   ```
   app.set('view engine', 'ejs');
   ```
3. Crear las plantillas en la carpeta `views/`.
4. Usar las plantillas dentro de un endpoint con:
   ```
   res.render(
     'rutaDeLaPlantillaDentroDeLaCarpetaViews',
     datosQueLePasamosALaPlantilla
   );
   ```
5. Usar condicionales y bucles dentro de la plantilla, si es lo que necesitamos.

Ejercicios

1. Crea una plantilla para directoras

Partiendo del ejercicio básico del vídeo:

1. Añade un fichero en `src/directors-data.json` con datos sobre dos directoras.
2. Añade a `src/index.js` un nuevo endpoint del tipo `app.get('/es/directora/:directorId', ...)` para gestionar peticiones del tipo http://localhost:3000/es/directora/iciar-bollain.
3. Obtén los datos de la directora que nos llegan por `req.params`.
4. Crea una plantilla en `views/pages/director.ejs` con el HTML que quieras.
5. Usa los datos de la directora para pintarlos en la plantilla.

2. Filtra por el año de los premios

Partiendo del ejercicio de bucles del vídeo:

1. Añade un filtro en `src/index.js` para que solo se pinten los premios del año 2004.

Una vez lo hayas conseguido, obtén el año de filtrado de los query params, es decir:

- Si entras a la página http://localhost:3000/es/film942734.html?adwarsYear=2003 se deben pintar solo los premios de 2003.

- Si entras a la página http://localhost:3000/es/film942734.html?adwarsYear=2004 se deben pintar solo los premios de 2004.

3. Plantillas dentro de plantillas

Partiendo del ejercicio básico del vídeo:

1. Crea una nueva plantilla en `views/partials/search.ejs` que sea un formulario de búsqueda. Cualquier código HTML vale, ya que no vamos a crear la funcionalidad del formulario. Como si pones un "hola mundo".
2. Añade este partial a los partials de la cabecera y el footer. Para ello elige bien la ruta relativa.

Si consigues que se vea el formulario en la página, el ejercicio estará bien.

4.4 API: buenas prácticas

> **Nota:** esta mini lección es importante pero no urgente. Puedes posponerla hasta que tengas interiorizado cómo funciona una API.

Para esta lección hemos pedido a dos grandes programadoras que nos echen una mano. Ellas son:

- **Débora Gómez**: trabaja en MyToys Group como Engineering Lead y ha colaborado en muchas ocasiones con Adalab dando master classes y como mentora de nuestras alumnas. LinkedIn / Twitter.
- **Esther Pato**: trabaja en Paradigma Digital como Front End Developer, es Adalaber de la Promo Dorcas y ha impartido master classes en Adalab. LinkedIn / Twitter.

Desde Adalab queremos agradecerles esta colaboración para ayudar a las nuevas Adalabers a estar mejor formadas.

Buenas prácticas al crear una API

Durante este módulo os estamos enseñando cómo hacer una API, con todas las opciones. Una vez que hayas entendido bien cómo funciona debes aprender cuáles son las buenas prácticas y más usadas en las empresas.

Introducción a las API

En este vídeo vemos:

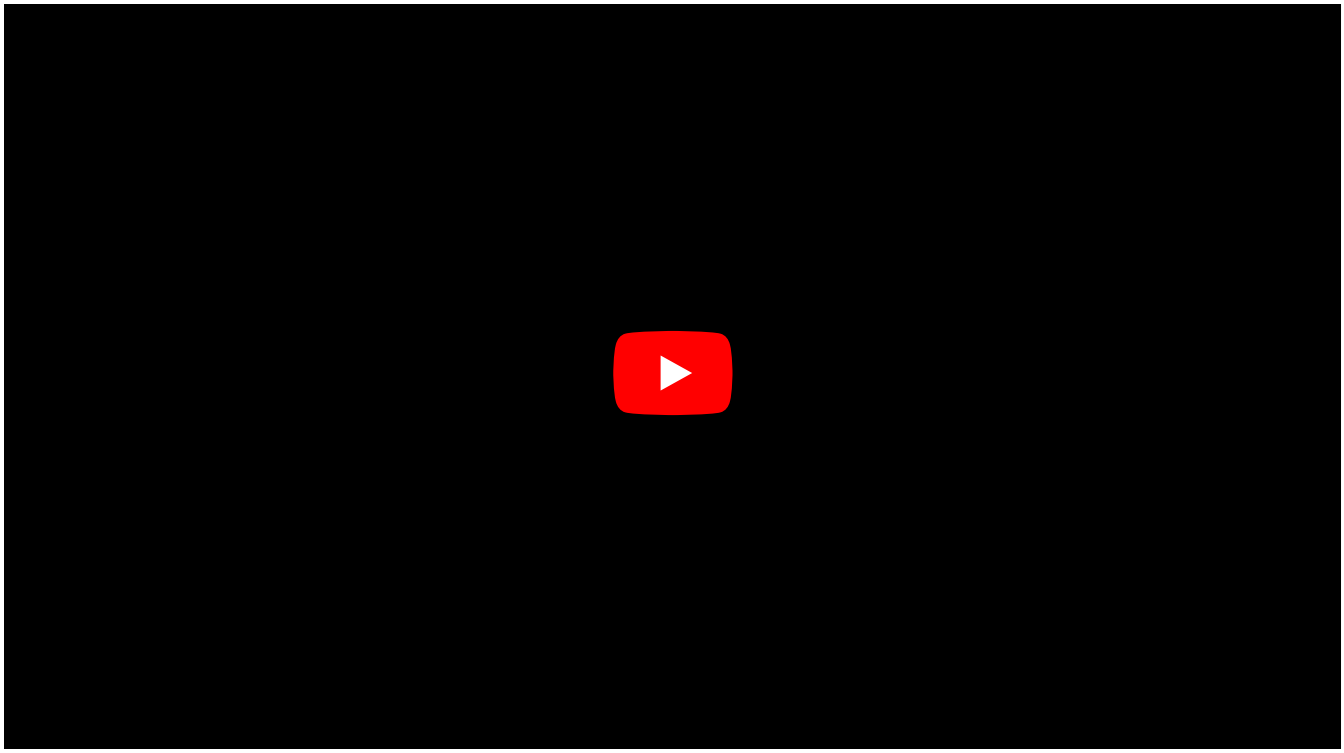
- Qué es una API y en concreto una API Rest
- Qué verbos o métodos se utilizan en las peticiones
- Cómo es la estructura de la URL de una petición
- Qué es el body de una petición
- Qué son los códigos HTTP de las respuestas

Cómo crear una buena URL en nuestros endpoints

En este vídeo vemos:

- Cómo crear una buena URL de un endpoint

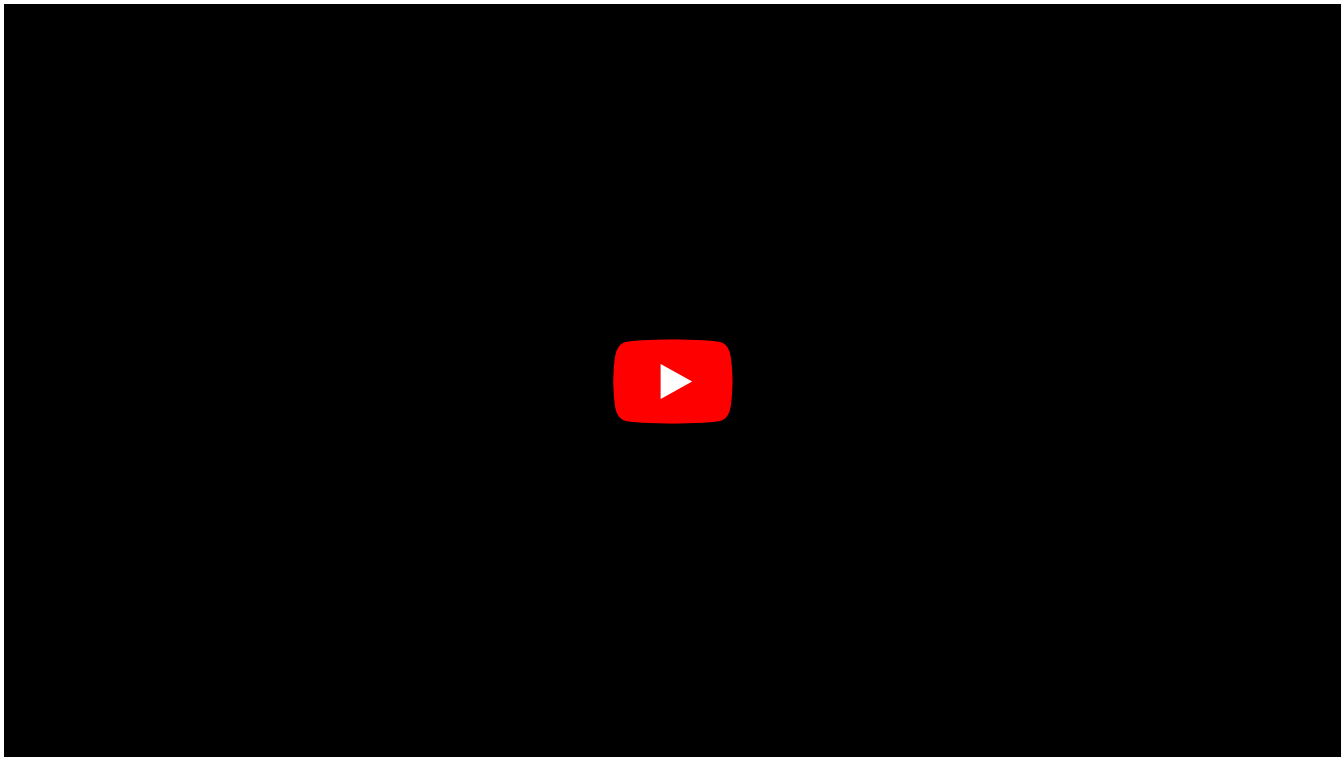
Métodos HTTP y códigos de respuesta HTTP



En este vídeo vemos:

- Cuáles son los verbos o métodos que debemos usar en cada petición (GET, POST, PUT, DELETE)
- Qué código HTTP debemos usar en cada respuesta

Query params

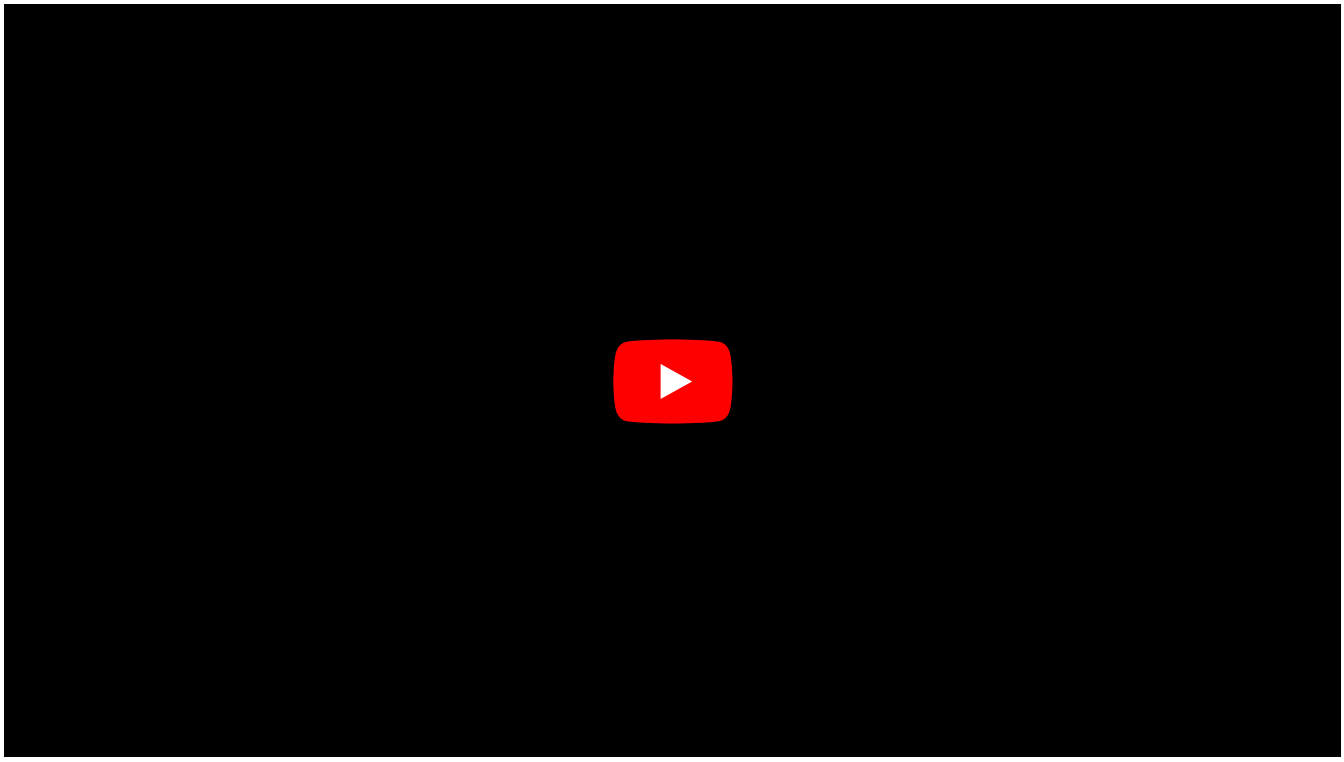


En este vídeo vemos:

- Qué datos solemos enviar en los query params

- Cómo paginamos un listado grande de resultados

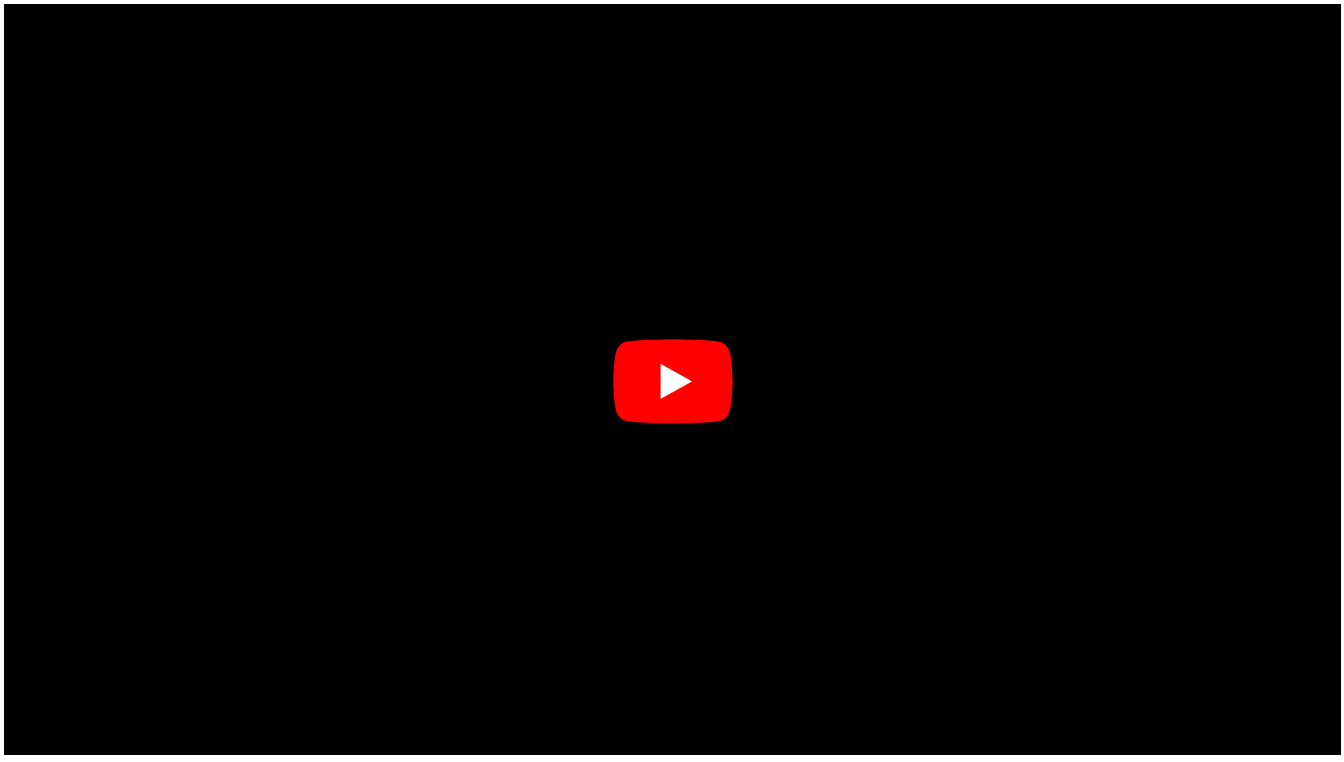
Body params



En este vídeo vemos:

- Qué datos devolvemos en el body de una respuesta

Header params



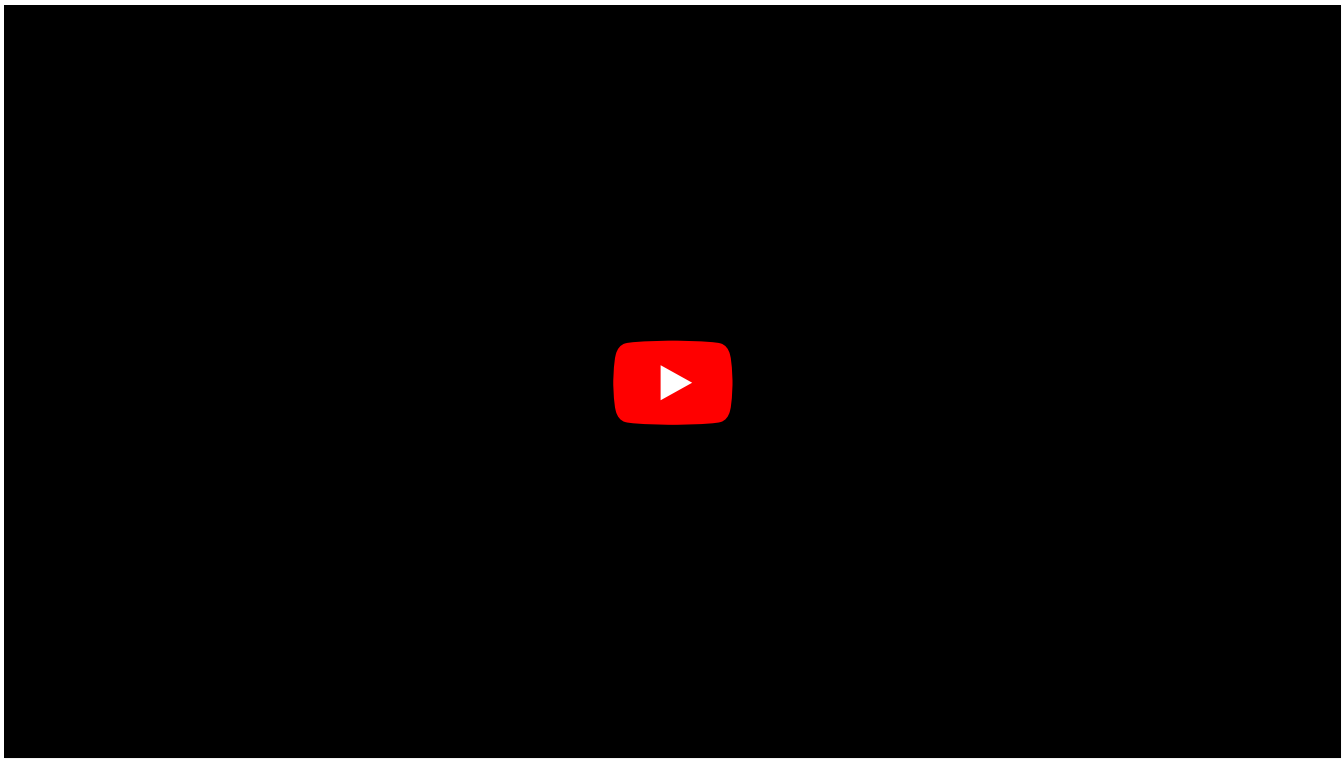
En este vídeo vemos:

- Qué datos debemos enviar en las cabeceras
- Qué nombres debemos usar para los datos que enviamos en las cabeceras

Ejercicios: ToDo List o lista de tareas

1. Define los métodos de las peticiones y sus respectivos paths

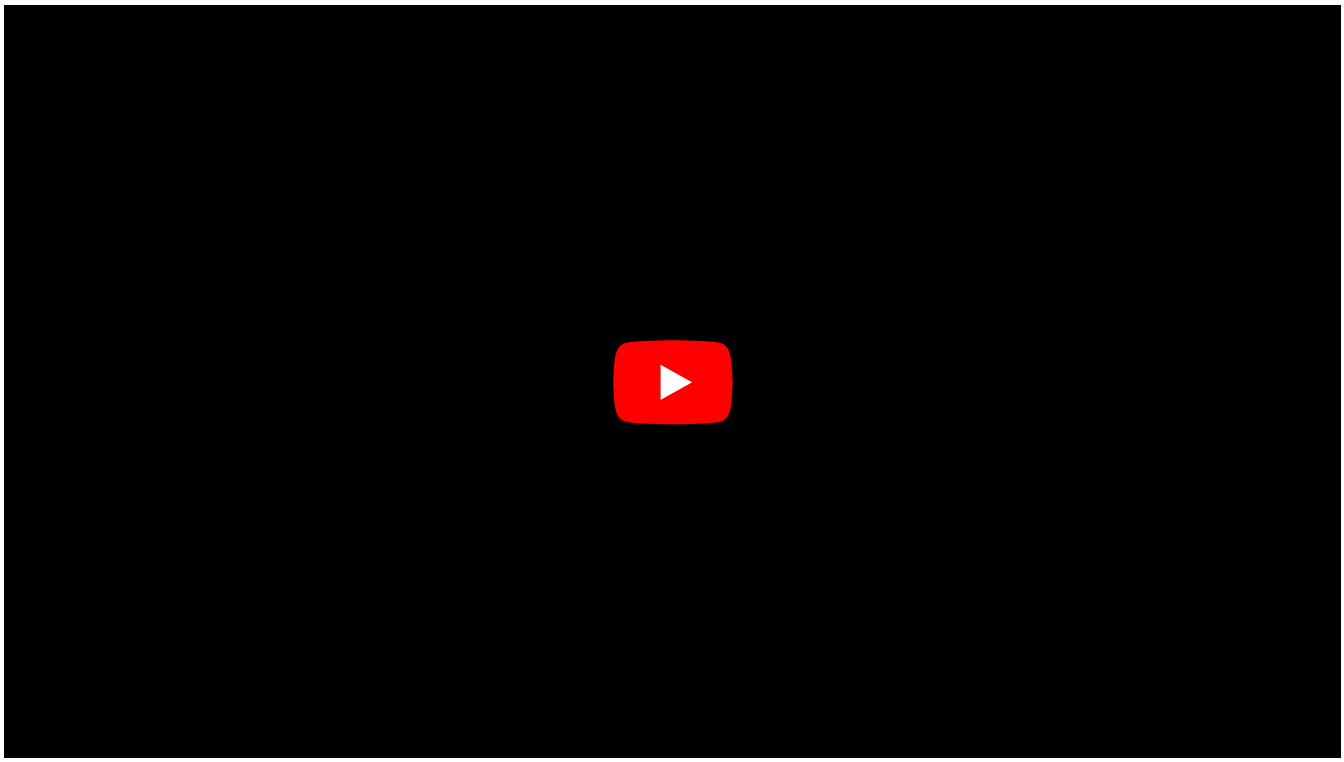
Enunciado:



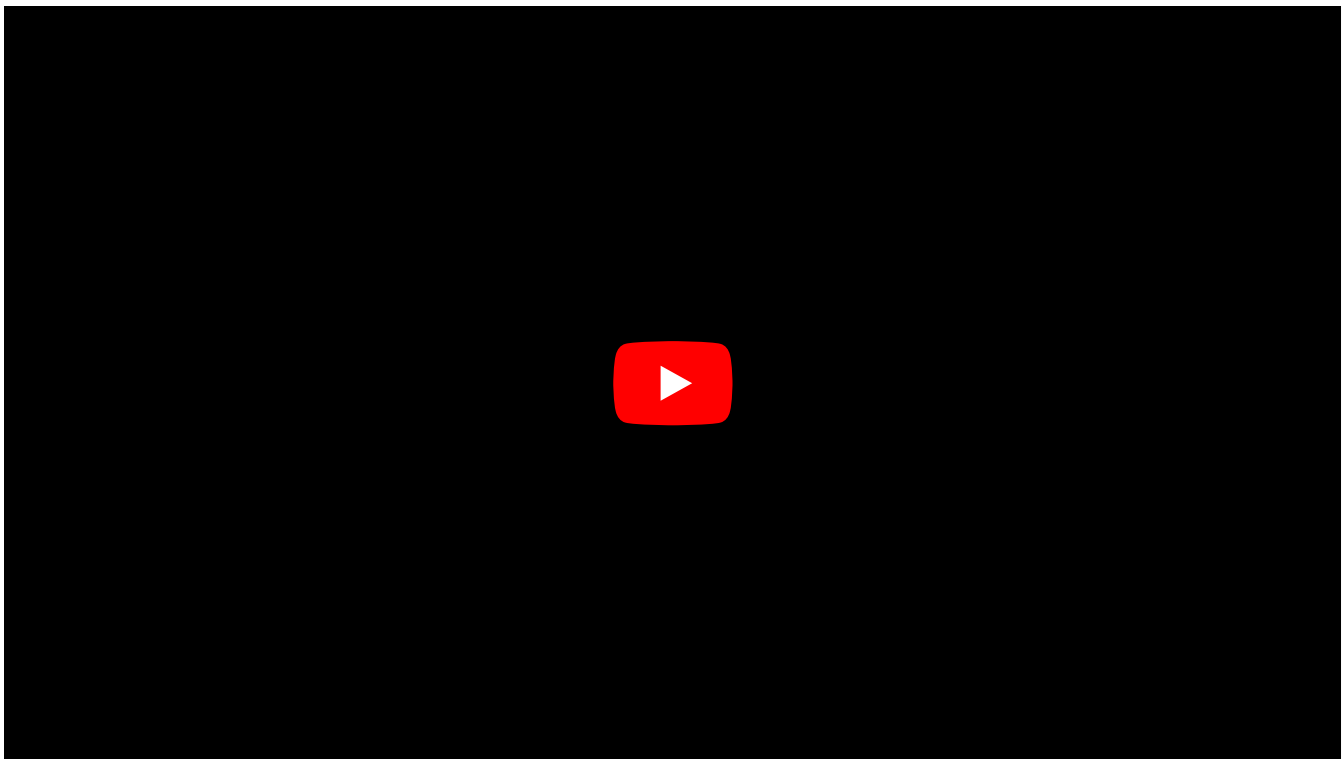
Solución:

2. Define el cuerpo y los códigos HTTP de la respuesta

Enunciado:



Solución:



Netflix

Queremos practicar lo que hemos aprendido sobre el motor de plantillas. Para ello vamos a crear unas páginas, por ejemplo http://locahost:4000/movie/1 o http://locahost:4000/movie/2, que muestren las películas del catálogo de Netflix.

Esta funcionalidad también podríamos hacerla directamente desde React y con React Router Dom como hemos hecho en el ejercicio de Rick y Morty.

Esta es una funcionalidad solo de back, por ello en este ejercicio no tienes que cambiar ningún fichero del proyecto de React.

> **Nota:** si no te da tiempo a terminar estos ejercicios no te preocupes. No los necesitas para seguir trabajando los próximos días.

---

1. Consigue el id de la película que se va a renderizar

Para crear un motor de plantillas, antes tenemos que crear un endpoint para escuchar las peticiones:

1. Añade un endpoint a `src/index.js` del tipo `server.get('/movie/:movieId', (req, res) => { ... });`.
2. Añade un console dentro del endpoint para mostrar en la terminal los URL params.
3. Entra en http://localhost:4000/movie/1.
4. Si todo ha ido bien en la terminal debería mostrar un **1** o el número que hayas puesto en la URL anterior.

Por ahora no estamos respondiendo a la petición, por ello puede que el navegador no muestre nada.

> **Nota:** este endpoint lo debes poner antes del código de los servidores de estáticos. Si lo pones después este endpoint no gestionará la petición porque lo estarán haciendo los servidores de estáticos.

---

2. Obtén la película

Ahora tenemos que buscar los datos de la película que se va a mostrar:

1. En ejercicios anteriores ya has importado las películas (movies) al principio de `src/index.js` las movies.
2. Con un `find` busca en ese array de películas que has importado la que tenga el mismo id que estás recibiendo por URL params. Guárdala en una constante, por ejemplo en `const foundMovie = ...`.

3. Una vez hayas encontrado la película consoléala.
4. Si has hecho todo bien en la terminal debería aparecer algo como `{ id: '1', title: 'Gambita de dama', ... }`.

---

3. Renderiza una página cualquiera

1. En la raíz del proyecto instala el motor de plantillas con `npm install ejs`.
2. En `src/index.js` configura el motor de plantillas añadiendo la línea `server.set('view engine', 'ejs');`
3. Crea el fichero `./views/movie.ejs` y mete código HTML dentro. Un "hola mundo" servirá.
4. Añade al endpoint el código `res.render('movie')`, donde `movie` es el nombre del fichero de la plantilla que hemos creado en el paso anterior.
5. Para saber que todo está bien entra en http://localhost:4000/movie/1 o http://localhost:4000/movie/2 y verás que en el navegador se muestra el HTML que has metido en la plantilla `./views/movie.ejs`.

---

4. Renderiza la película

1. Ahora debes sustituir la línea `res.render('movie')` por `res.render('movie', foundMovie)`. Recuerda que `foundMovie` es un objeto con los datos de la película. Aquí le estamos diciendo que renderice la plantilla `movie` con los datos `foundMovie`.
2. Por último debes añadir a `./views/movie.ejs` el código `<%= name %>`, `<%= gender %>`... donde quieras que se pinten estos datos.
3. Y por último, para comprobar que todo está fetén entra en http://localhost:4000/movie/1; deberías ver los datos de la película Gambita de dama.

Añade estilos

Para añadir estilos necesitamos crear un CSS. Pero para que una página renderizada con un motor de plantillas pueda usar este CSS debes hacer dos cosas:

1. Crea otro servidor de estáticos (sí, otro más) para servir tu fichero CSS.
2. Añade a `./views/movie.ejs` un `<link rel="stylesheet" href="aqui-la-ruta-correcta-de-los-estilos.css">`.

4.5 Bases de datos I

5.1 Intro a bases de datos

> **Nota:** esta mini lección es importante.

Una base de datos es **el programa** donde vamos a almacenar **todos los datos de nuestra aplicación y de nuestras usuarias**. Por ejemplo:

- Email, contraseña, nombre, etc. de nuestras usuarias registradas
- Tweets de Twitter
- Comentarios en los vídeos de Youtube
- Artículos de un blog
- Eventos de Google Calendar
- Configuración interna de nuestra web

En resumen, todos los datos que no tenemos a la hora de programar.

¿Dónde se ejecutan las bases de datos?

La base de datos de una aplicación **se está ejecutando en todo momento en nuestro servidor**, ya sea nuestro servidor local de desarrollo (que es un programa que instalamos en nuestro ordenador) o el servidor de producción donde subiremos nuestro código.

Cuando reiniciamos nuestro ordenador, si volvemos a arrancar una aplicación de **Node JS, esta se ejecuta como si fuera la primera vez**. Es decir, no guarda nada de una ejecución a otra.

Pero cuando trabajamos con una base de datos, los cambios que hacemos sobre ella **se guardan en ficheros especiales**. Por ello, si reiniciamos el ordenador y volvemos a abrir la base de datos, esta mantiene los datos que tenía la última vez que se ejecutó. Así, **las bases de datos nos permiten guardar datos de forma permanente.**

¿Cómo accedemos a una base de datos?

Desde nuestro código de Node JS vamos a leer y escribir datos en nuestra base de datos **usando una determinada sintaxis**. También podemos hacer lo mismo desde cualquier otro lenguaje de programación de back end usando la misma sintaxis o muy parecida.

Es decir, que lo que vamos a aprender sobre bases de datos nos vale para cuando estemos programando en cualquier lenguaje de back end.

¿Qué vamos a aprender en Adalab?

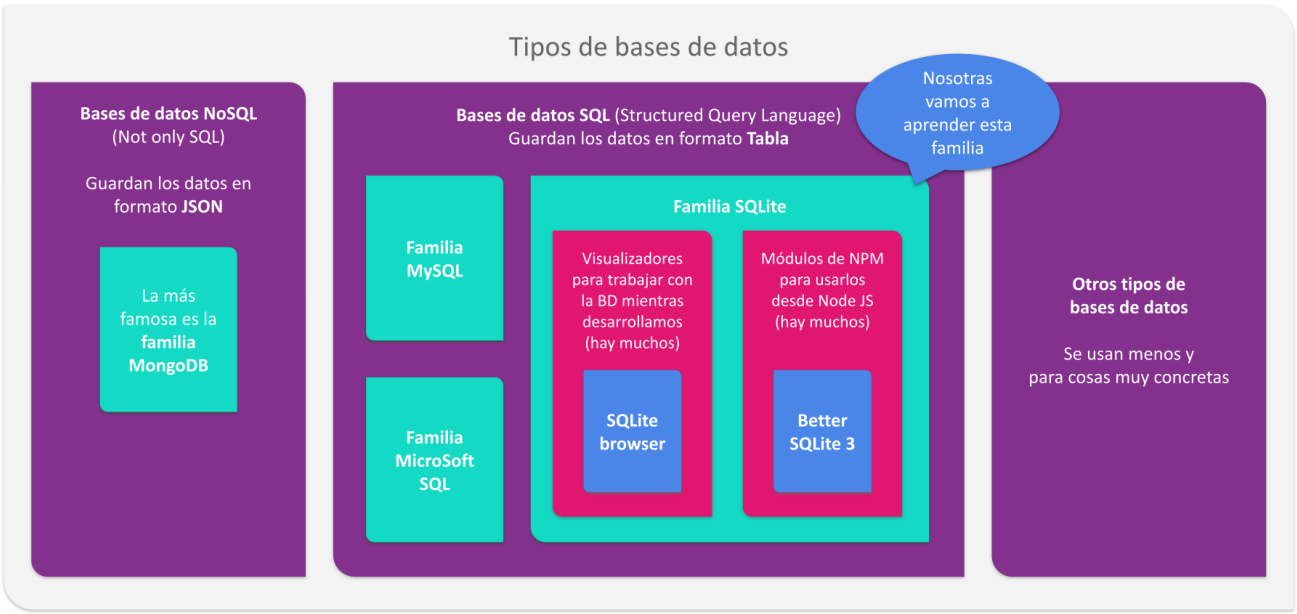


Diagrama de conceptos

EEl mundo de las bases de datos es inmenso, así que nosotras vamos a aprender solo una parte. Te recomendamos que cada vez que hablemos de un concepto nuevo, revises este diagrama para entender de qué estamos hablando.

¿Qué tipos de bases de datos hay?

Hay muchísimos tipos de bases de datos. Cada tipo está orientado a una cosa, por ejemplo:

- Bases de datos que guardan datos importantes, como las de los bancos, que almacenan todos nuestros movimientos bancarios. Están pensadas para ser muy seguras y no perder información bajo ningún concepto.
- Otras están pensadas para guardar información efímera, por lo que son más pequeñas y las búsquedas que se hacen sobre ellas muy rápidas. Un ejemplo son las que guardan las notificaciones emergentes de WhatsApp de tu móvil. Cuando una amiga te envía un WhatsApp, se añade la notificación a una base de datos. Horas después, tú enciendes el móvil, WhatsApp comprueba que en su base de datos tienes notificaciones pendientes y te las envía. Por último el servidor de WhatsApp borra la notificación de su base de datos para siempre jamás.

A nosotras nos interesa saber los tipos de bases de datos que existen **en función de cómo estructuran los datos**.

Bases de datos de tipo JSON o NoSQL

Supongamos que queremos guardar la información de las alumnas de Adalab en un JSON. Podría tener esta pinta:

```
[
  {
```

```
    "id": 1,
    "email": "maria@gmail.com",
    "password": "987widJYVxyh",
    "name": "María"
  },
  {
    "id": 2,
    "email": "lucia@hotmail.com",
    "password": "qwertyui",
    "name": "Lucía"
  },
  {
    "id": 3,
    "email": "sofia@yahoo.com",
    "password": "mnbvcdfgu",
    "name": "Sofía",
    "pc": {
      "number": 54,
      "brand": "Lenovo"
    }
  }
]
```

En esta base de datos vemos que las 3 alumnas tienen un `id`, un `email`, un `password` y un `name`. Pero solo Sofía tiene un campo más, que se llama `pc` y es un objeto. El campo `pc` indica que Adalab ha prestado a Sofía el portátil número 54, de la marca Lenovo, para que lo use durante el curso. En las alumnas María y Lucía no hemos guardado información acerca de sus portátiles.

Sabemos que los objetos de JSON pueden tener la estructura que queramos, por ello podemos guardar los datos que queramos de una alumna. **Y cada alumna puede tener diferentes datos.**

Principales características de las bases de datos de tipo JSON

- Los datos se guardan en **colecciones, es decir, arrays de objetos**.
- **Los elementos de una colección pueden ser diferentes.** Unas usuarias pueden tener unos datos y otras otros.
- Estos objetos **pueden estar compuestos de otros datos**: números, strings, arrays, objetos...
- **No tienen una estructura fija** de los datos, por lo que **somos libres** de guardar lo que queramos fácilmente.

- Al no tener una estructura fija, a veces es más difícil hacer búsquedas de datos en ellas.
- Una base de datos puede estar compuesta de muchas colecciones: una para alumnas, otra para promociones, otra para profes, otra para las empresas de la bolsa de empleo...

A estas bases de datos se les llama **NoSQL** (not only SQL) y la más famosa de este tipo es MongoDB.

Bases de datos de tipo tabla o SQL

Ahora supongamos que queremos guardar los datos de las alumnas de Adalab en una tabla. Podría tener esta pinta:

id	email	password	name	pc	pcNumber	pcBrand
1	maria@gmail.com	987widJYVxyh	María	false		
2	lucia@hotmail.com	qwertyui	Lucía	false		
3	sofia@yahoo.com	mnbvcdfgu	Sofía	true	54	Lenovo

Como veis esto es muy parecido a guardar los datos en un documento de Excel.

Principales características de las bases de datos de tipo tabla o SQL

- Los datos se guardan en **tablas, es decir, en filas y columnas**.
- **Todas las filas o registros de una tabla deben tener los mismos campos**, aunque algunos puedan estar vacíos.
- Si queremos añadir un dato más a una fila, tenemos que añadir una columna más, y por ello todas las filas tendrán ese campo, aunque esté vacío. Dicho de otra forma, **o todas las usuarias tienen una columna o no la tiene ninguna**.
- **La estructura de datos es fija.**
- Por eso **las búsquedas son más rápidas**, ya que la base de datos sabe qué datos hay y cómo buscarlos.
- Al ser una estructura fija **se pueden configurar muchas cosas**. Por ejemplo, esta tabla se podría configurar para que si la columna `pc` es `true`, las columnas `pcNumber` y `pcBrand` deban estar rellenas, pero si es `false` deban estar vacías.
- **Las columnas tienen un tipo de dato**. Por ejemplo, `id` y `pcNumber` son números, `pc` es un booleano y el resto son `strings`.
- **Una base de datos puede estar compuesta de muchas tablas**: una para alumnas, otra para promociones, otra para profes, otra para las empresas de la bolsa de empleo...

A estas bases de datos se les llama **SQL** (Structured Query Language).

En Adalab vamos a aprender a usar las bases de datos SQL porque:

- Son más fáciles de aprender.
- Son las más usadas.

SQL vs. NoSQL

Las bases de datos NoSQL no tienen una estructura fija, por ello nos dan mucha libertad para trabajar como queramos. Esto nos obliga a tener mucho cuidado para que dicha libertad no se convierta en

libertinaje y que no perdamos el control sobre qué datos estamos guardando.

Las bases de datos SQL sí tienen una estructura fija, por lo que son más estrictas. Tenemos las manos más atadas a la hora de meter datos. A cambio, la base de datos sabe qué datos guarda y por ello se pueden configurar muchas cosas, automatizar tareas, hacer búsquedas más óptimas, prohibir que se guarden datos por error, etc.

¿Qué base de datos uso en mi proyecto?

Cuando trabajéis en una empresa y tengáis que elegir una base de datos u otra, debéis analizar qué tipos de datos vais a guardar, y en función de ello elegir.

5.2 SQL y SQLite

> **Nota:** esta mini lección es importante.

Ya sabemos que hay muchos tipos bases de datos:

- Bases de datos que guardan los datos en **colecciones de tipo JSON**
- Bases de datos que guardan los datos en **tablas tipo Excel**
- Otros tipos de bases de datos

En Adalab os vamos a enseñar a trabajar con bases de datos de **tipo tabla**. Dentro de este tipo, hay muchas bases de datos. La más famosa y usada por más empresas es **SQL** (Structured Query Language).

Dentro de SQL a su vez existen muchas versiones o familias de bases de datos, porque en programación cada cual coge algo que ya existe, lo cambia un poquito, dice que ha reinventado la rueda y pide dinero a cambio ;).

Por ejemplo, dentro de **SQL** están MySQL, Microsoft SQL, SQLite...

Lo bueno es que si aprendemos a usar una de estas familias, ya sabremos usar el resto de familias, pues de una a otra cambian muy pocas cosas.

En Adalab vamos a trabajar con **SQL + SQLite**, ya que SQLite está muy bien para aprender y para ser usada en proyectos pequeños.

Cómo instalar SQLite Browser

Para usar una base de datos SQLite dentro de nuestros proyectos de Node JS instalaremos un paquete de NPM. Ya veremos cuál.

De momento necesitamos un entorno gráfico que nos permita trabajar cómodamente, así que vamos a instalar en nuestro ordenador **SQLite Browser**, un visualizador de bases de datos (o el DevTools de SQLite).

Instalación desde Windows

Entra en la página de descargas de SQLite browser y descarga e instala la última versión. Seguramente sea la que pone **DB Browser for SQLite - Standard installer for 64-bit Windows**.

Instalación desde Mac

Entra en la página de descargas de SQLite browser y descarga e instala la última versión. Puedes hacerlo:

- Descargando e instalando la aplicación, o
- Ejecutando `brew install --cask db-browser-for-sqlite` desde una terminal de tu Mac.

> **Nota:** recuerda que a lo mejor tienes que usar `sudo`.

Instalación desde Ubuntu

Entra en la página de descargas de SQLite browser, apartado **Ubuntu and Derivatives** y verás que te pide que ejecutes los siguientes comandos en una terminal:

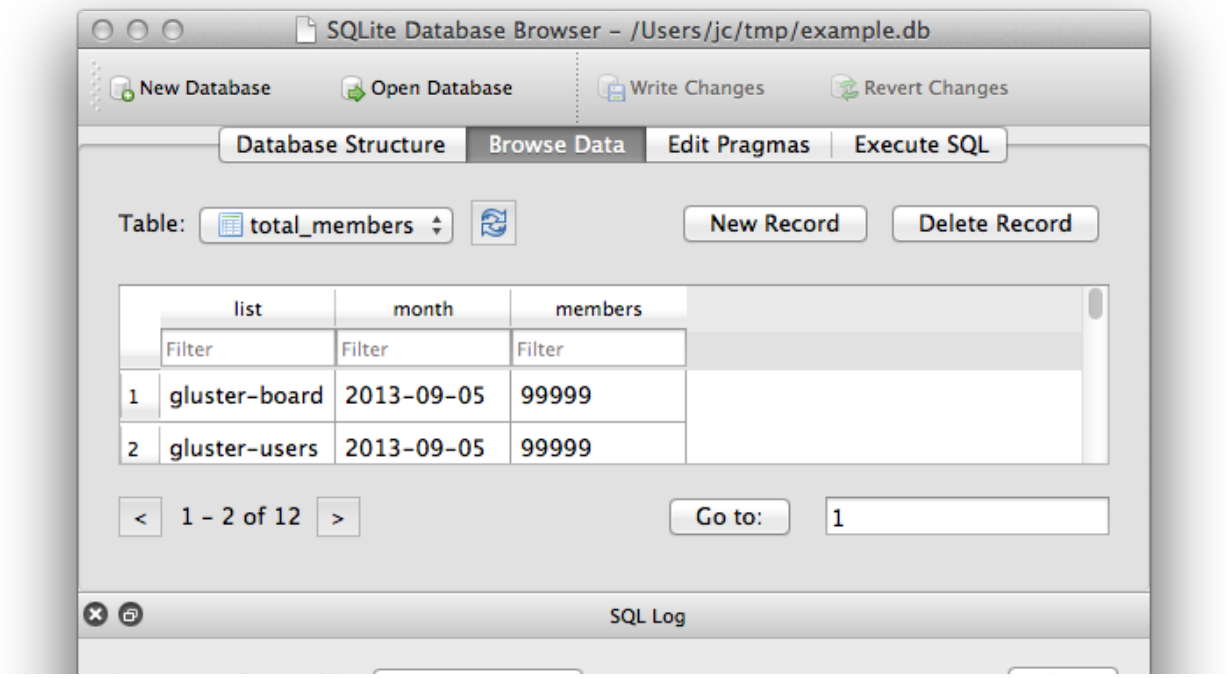
```
sudo add-apt-repository -y ppa:linuxgndu/sqlitebrowser
```

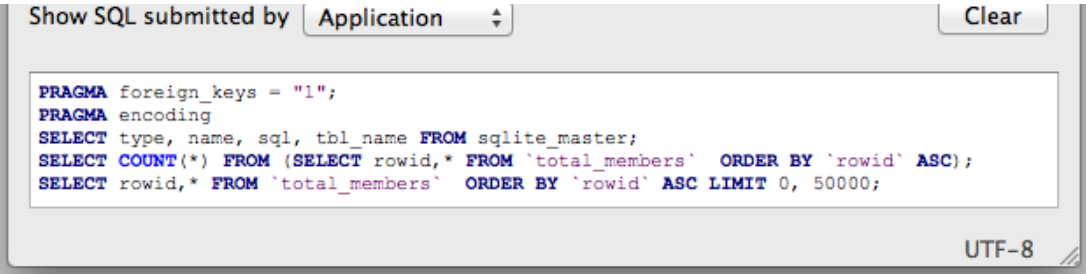
```
sudo apt-get update
```

```
sudo apt-get install sqlitebrowser
```

Cómo abrir SQLite browser

Para saber que lo has instalado bien abre desde tu ordenador SQLite browser y verás este programa:

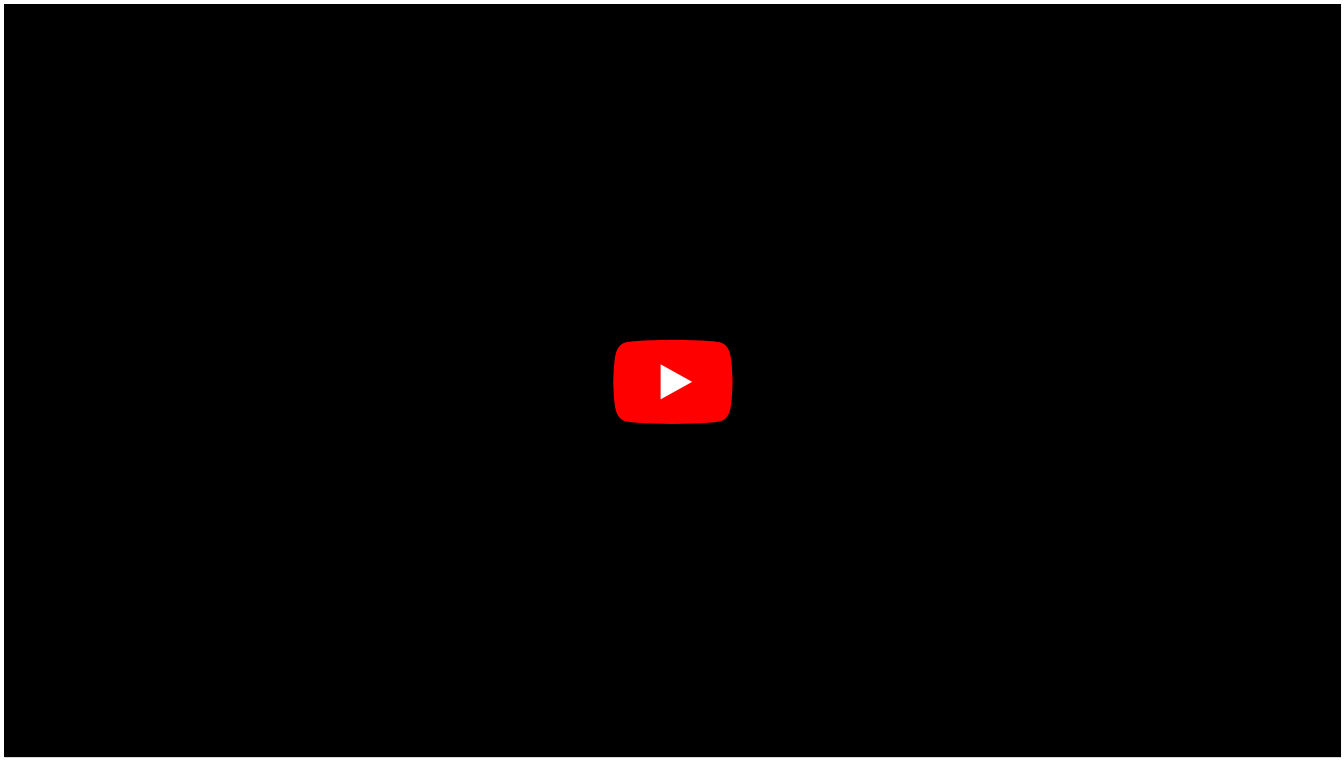




SQLite browser

Nuestra primera base de datos

Ahora que ya tenemos instalado **SQLite browser**, abre la aplicación desde tu ordenador para poder seguir los pasos que damos en este vídeo.



> **Nota:** si sigues los pasos de este vídeo, borra el fichero de la base de datos que está en el ejercicio (`./src/dababase.db`), para crearla tú desde cero.

> Ejercicio del vídeo

> **Importante:** os recordamos que cada vez que cambiamos algo en la base de datos desde SQLite browser hay que pulsar en **Guardar datos**.

Diferencias entre bases de datos, tablas y registros

- **Una base de datos es un conjunto de tablas**, lo que sería **un documento de Excel** que contiene varias hojas. Una aplicación de tamaño normal solo tiene una base de datos.
  - Si la aplicación es muy grande puede tener varias bases de datos.
  - Si la aplicación es todavía más grande puede combinar bases de datos de diferentes tipos.
- **Una tabla es un conjunto de columnas y registros**, lo que sería **una hoja de Excel**. Vamos a tener una tabla por cada una de las cosas que queramos guardar. Por ejemplo, en una tienda online tendremos una tabla para usuarias, otra tabla para productos, otra para pedidos, etc.
- **Un registro es cada una de las filas de una tabla**, lo que sería **una fila** de una hoja de un documento de Excel. Cada registro guarda una de nuestras usuarias, uno de nuestros productos, uno de nuestros pedidos, etc. En cada columna guardamos un dato de cada registro: en la tabla de usuarias en una columna guardamos el email, en otra la contraseña, en otra el nombre...
- **Un campo es cada dato de un registro**, lo que sería **una celda** de una fila de una hoja de un documento de Excel. Podemos configurar qué tipo de dato guarda cada campo o celda: texto, número entero, número con decimales, booleano, etc.

Cada vez que hacemos una **consulta** a la base de datos para leer o escribir utilizamos el término en inglés, **hacer una query**.

Cuando penséis y habléis de bases de datos debéis usar las palabras adecuadas en cada caso.

5.3 SQL Select

> **Nota:** esta mini lección es la más importante de hoy.

Para trabajar con una base de datos podemos hacer 4 operaciones básicas:

- Añadir un nuevo registro
- Borrar un registro existente
- Modificar un registro existente
- Leer uno o varios registros, que es la que vamos a aprender en esta lección

A estas operaciones se les llama **CRUD**: create, read, update and delete.

Consideraciones sobre las operaciones de una base de datos

- **La operación de lectura no modifica la base de datos.** El resto de operaciones sí.
- **Todas estas operaciones se realizan sobre una única tabla.** Por ahora, si queremos hacer operaciones sobre varias tablas tenemos que hacer varias consultas o queries por separado.
- Existen operaciones más complejas, pero siempre son combinaciones de estas 4 operaciones básicas.
- Cada vez que accedemos a una base de datos decimos que hacemos una **query o consulta a la base de datos**.

Sintaxis de SELECT en SQL y SQLite browser

`SELECT` nos permite **seleccionar o leer uno o varios registros de una tabla** de la base de datos.

Supongamos que tenemos esta tabla llamada `users`:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía
3	sofia@yahoo.com	mnbvcdfgu	Sofía

SELECT

Con `SELECT` seleccionamos las columnas que queremos leer y con `FROM` elegimos el nombre de la tabla de donde queremos leer. Por ello:

```
-- Seleccionar el id y el email de todas las usuarias
SELECT id, email FROM users
```

```
-- Seleccionar todas las columnas o datos de todas las usuarias
SELECT * FROM users
-- El símbolo `*` indica que queremos seleccionar todas las columnas de una tabla.
```

`SELECT` y `FROM` son palabras obligatorias al hacer una query de lectura.

WHERE

Con `WHERE` indicamos la condición que debe cumplir el registro (o fila) para poder seleccionarlo. Sería como hacer `filter` o un `if` de JavaScript: queremos leer una fila **si** cumple la **condición** indicada:

```
-- Seleccionar todas las columnas de todas las usuarias cuyo id sea mayor o igual que 2
SELECT * FROM users WHERE id >= 2
-- Esto nos devolverá 0, 1 o varios registros
```

```
-- Seleccionar todas las columnas de la usuaria cuyo id sea igual a 2; esto nos devolverá solo
SELECT * FROM users WHERE id = 2
-- Esto nos devolverá 0 o 1 registros en función de si en la tabla existe el id
```

```
-- Seleccionar todas las columnas de la usuaria cuyo email sea lucia@hotmail.com
SELECT * FROM users WHERE email = 'lucia@hotmail.com'
```

```
SELECT * FROM users WHERE email = 'lucia@hotmail.com' AND password = 'qwertyuiop';
```

`WHERE` no es obligatorio al hacer una query de lectura, pero casi siempre lo usamos. Si una consulta no tiene `WHERE` nos devolverá todos los registros de una tabla.

> **Nota:** `WHERE` también es muy usado en operaciones de modificación y borrado de registros en tablas. Lo veremos más adelante.

ORDER BY

Con `ORDER BY` podemos indicar el orden en el que queremos leer los registros:

```
-- Seleccionar todas las columnas de todas las usuarias ordenadas por nombre de forma ascende
SELECT * FROM users ORDER BY name ASC
```

```
-- ASC es el orden por defecto, así que lo podemos omitir. Esta query es igual que la anterio
SELECT * FROM users ORDER BY name
```

```
-- Seleccionar todas las columnas de todas las usuarias ordenadas por nombre de forma descend
SELECT * FROM users ORDER BY name DESC
```

```
-- Seleccionar todas las columnas de todas las usuarias ordenadas por nombre de forma descend
SELECT * FROM users ORDER BY name ASC, email DESC
```

`ORDER BY` no es obligatorio al hacer una query de lectura.

LIMIT y OFFSET

Con `LIMIT` indicamos el **número máximo de registros** que queremos leer:

```
-- Seleccionar todas las columnas hasta un máximo de 2 usuarias
SELECT * FROM users LIMIT 2
-- Esta query nos devuelve como máximo 2 registros aunque la tabla tenga más
```

Con `OFFSET` indicamos el primer registro a partir del cual queremos leer:

```
-- Seleccionar todas las columnas hasta un máximo de 2 usuarias empezando en la posición 5
SELECT * FROM users LIMIT 2 OFFSET 5
-- Esta consulta nos devolverá los registros 6º y 7º
```

Ni `LIMIT` ni `OFFSET` son obligatorios al hacer una query de lectura.

`LIMIT` y `OFFSET` son muy útiles cuando tenemos muchos registros en una tabla. Por ejemplo, cuando

entramos en nuestro Twitter, la web solo nos muestra los primeros 50 tweets. Si bajamos hasta abajo del todo nos muestra los siguientes 50 tweets. Esto se haría con las siguientes consultas:

```
-- Query de los primeros 50 tweets: seleccionar todos los tweets hasta un máximo de 50 tweets
SELECT * FROM tweets ORDER BY id LIMIT 50
-- Esta consulta nos devolverá los registros desde el id 1 hasta el id 49
```

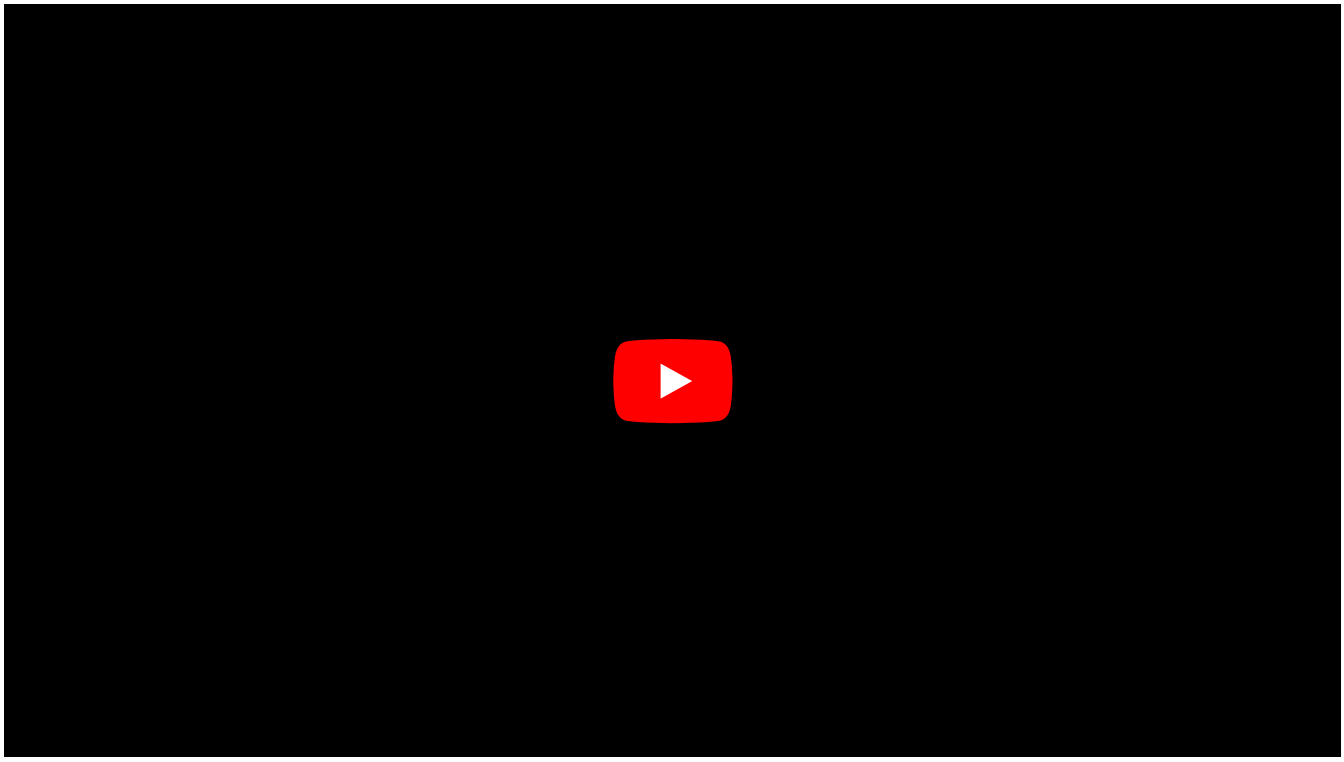
```
-- Query de los siguientes 50 tweets: seleccionar todos los tweets hasta un máximo de 50 twee
SELECT * FROM tweets ORDER BY id LIMIT 50 OFFSET 50
-- Esta consulta nos devolverá los registros desde el id 50 hasta el id 99
```

Conclusiones de la sintaxis de SQL

Nuestro trabajo cuando leemos datos de una tabla es combinar `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `OFFSET` y `LIMIT` para obtener los datos que queremos.

Mayúsculas vs. minúsculas: normalmente escribimos en **mayúsculas** las palabras `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `LIMIT` y otras que aprenderemos en próximas lecciones **porque son palabras del lenguaje SQL**. Y escribimos en **minúsculas o camelCase las columnas y los nombres de las tablas**. Esta forma no es obligatoria pero facilita la lectura de las queries.

Comentarios: todos los lenguajes de programación tienen caracteres para indicar comentarios. En JS ponemos `// mi comentario` o `/* mi comentario */`. En SQL los comentarios se ponen con dos guiones `--` al principio de la query.



> Ejercicio del vídeo

Sintaxis de SELECT en Node JS y Better SQLite 3

Ya sabemos leer de una tabla de base de datos usando el programa SQLite browser. Como hemos dicho este es el DevTools de SQLite. **Nos sirve para hacer pruebas mientras estamos programando.**

A partir de ahora, cuando tengamos que hacer una query, **primero la probaremos en SQLite browser** y cuando encontremos la sintaxis que queremos, la escribiremos en Node JS para que **Node JS haga la query por nosotras**.

Vamos a seguir los pasos para usar bases de datos en un proyecto de Node JS que ya tenemos creado. Tenemos que:

1º Instalar Better SQLite 3

En NPM hay muchos módulos para acceder a todo tipo bases de datos. También hay muchos módulos para acceder a SQLite. En Adalab vamos a aprender a utilizar uno de ellos que se llama **Better SQLite 3**.

Para instalar **Better SQLite 3** escribiremos en la terminal:

```
npm install better-sqlite3
```

Lo que añadirá esta dependencia a nuestro `package.json`.

2º Indicar la base de datos que vamos a usar

A continuación creamos con SQLite browser una base de datos y guardamos el fichero dentro de nuestro proyecto, por ejemplo en `./src/database.db`. Luego escribiremos el siguiente código en `src/index.js`:

```
// importar el módulo better-sqlite3
const Database = require('better-sqlite3');

// indicar qué base de datos vamos a usar con la ruta relativa a la raíz del proyecto
const db = new Database('./src/database.db', {
  // con verbose le decimos que muestre en la consola todas las queries que se ejecuten
  verbose: console.log,
  // así podemos comprobar qué queries estamos haciendo en todo momento
});
```

3º Hacer una query

Una query está compuesta de dos líneas de código:

```
// preparamos la query
const query = db.prepare('SELECT * FROM users');
// ejecutamos la query
const users = query.all();
```

Normalmente haremos nuestras queries desde dentro de un endpoint de Express JS. Por ello el código sería:

```
// creamos el endpoint /users de tipo GET
app.get('/users', (req, res) => {
  // preparamos y ejecutamos la query
  const query = db.prepare('SELECT * FROM users');
  const users = query.all();
  // respondemos a la petición con los datos que ha devuelto la base de datos
  res.json(users);
});
```

4º Leer uno o varios registros: query.all() vs query.get()

Con este código:

```
// preparamos la query
const query = db.prepare('SELECT * FROM users');
// ejecutamos la query para obtener todos los registros en un array
const users = query.all();
// ejecutamos la query para obtener el primer registro en un objeto
const user = query.get();
```

`const users = query.all()` devuelve varios registros, por ello `users` será un **array de objetos**. Si no hay registros en tabla, `query.all()` devolverá un **array vacío**.

`const user = query.get()` devuelve un registro, por ello `user` será un **objeto**. Si no hay registros en la tabla, `query.get()` devolverá un `undefined`.

En resumen, cada vez que queramos leer registros de una tabla, debemos pensar si queremos un listado (y ejecutar `query.all()`) o un único registro (entonces ejecutaremos `query.get()`).

5º Leer uno o varios registros pasando datos

Con este código:

```
// preparamos la query
const query = db.prepare('SELECT * FROM users WHERE id >= ?');
// la ejecutamos indicando: WHERE id >= 2
const users = query.all(2);
```

Estamos indicando que **Better SQLite 3** debe sustituir la `?` por un `2` al ejecutar la query.

Con este código:

```
// preparamos la query
const query = db.prepare(
  'SELECT * FROM users WHERE email = ? AND password = ?'
);
// la ejecutamos indicando: SELECT * FROM users  WHERE email = 'lucia@hotmail.com' AND passwo
```

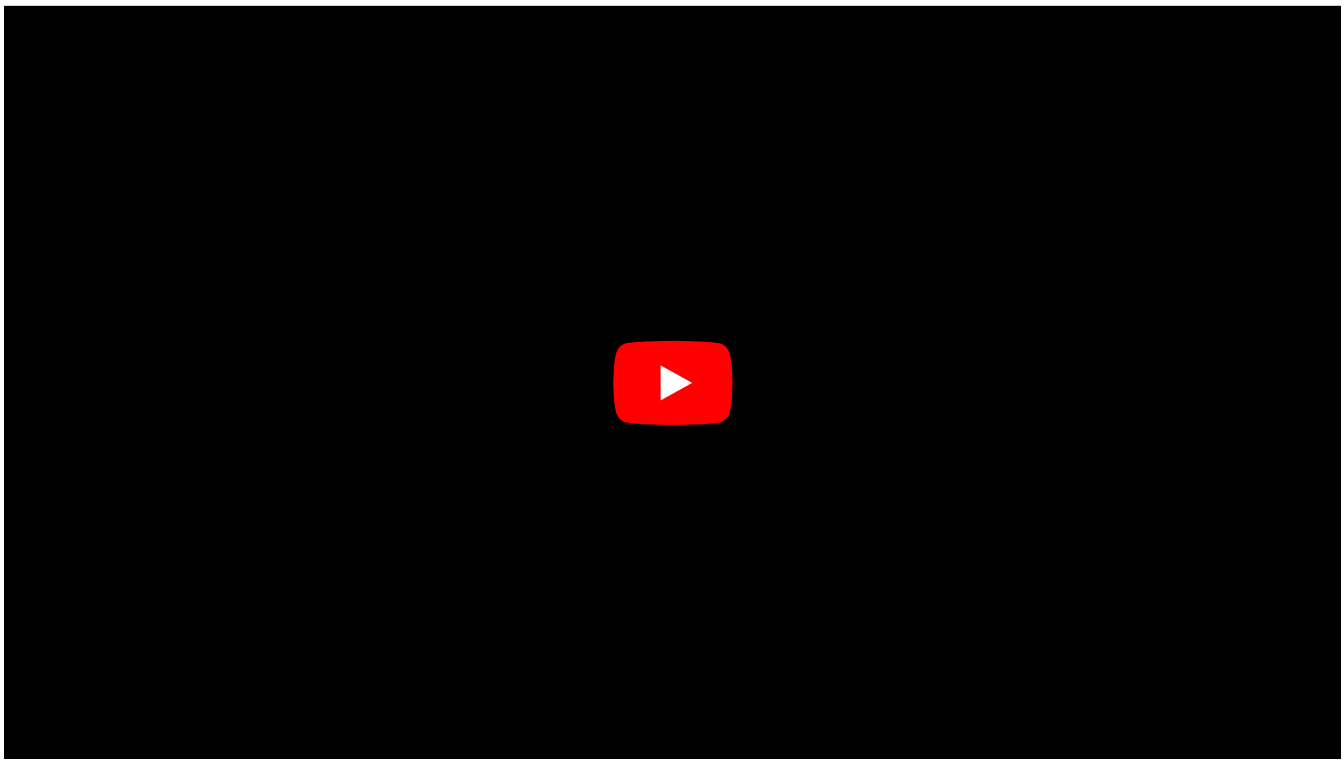
```
const users = query.get('lucia@hotmail.com', 'qwertyui');
```

Estamos indicando que **Better SQLite 3** debe sustituir la primera `?` por `'lucia@hotmail.com'` y la segunda `?` por `'qwertyui'` al ejecutar la query.

> **Por cierto:** con esta query obtenemos un objeto con la usuaria que tenga ese email y esa contraseña. Esta es la típica consulta que se usa en un login para comprobar si la usuaria existe en nuestra base de datos.

Los motivos por los que una query se hace en dos pasos (primero preparar la query y después ejecutarla) son:

- Para que podamos separar:
  - por un lado la sintaxis de la query
  - por otro lado los datos que pasamos a la query
- Para poder crear una query una única vez y ejecutarla muchas veces con muchos datos diferentes
- Para reducir las posibilidades de que nos equivoquemos al hacer una query, ya que es fácil olvidarse de unas comillas, una coma...



> Ejercicio del vídeo

Otras opciones para hacer un SELECT

Las opciones para leer datos de SQL son infinitas. Por ejemplo:

```
-- Seleccionar todas las columnas de todas las usuarias que tengan un id diferente de 2
SELECT * FROM users WHERE id <> 2
```

```
-- Seleccionar todas las columnas de la usuaria que tenga un id = 2 o un email = lucia@hotmai
SELECT * FROM users WHERE id = 1 OR email = 'lucia@hotmail.com'
```

```
-- Seleccionar todas las columnas de todas las usuarias cuyo email contenga la palabra: gmail
SELECT * FROM users WHERE email LIKE '%gmail%'
-- LIKE es muy usado para hacer búsquedas por palabras en una base de datos
```

La sintaxis de SQL es muy amplia

En este curso te enseñamos la sintaxis de SQL más usada y más importante, pero SQL tiene muchísimas opciones. Cada vez que estés programando en Node JS, tengas que hacer una query y no sepas cómo, tienes dos opciones:

- Pedir muchos datos a la base de datos y luego buscar y / o filtrar en Node JS. Esto **no es recomendable** ya que:
  - Estás haciendo tú algo en Node JS que SQL sabe hacer bien, y para lo cual ha sido diseñado.
  - Estás manejando una cantidad de datos grande cuando no es necesario.
  - Requiere más líneas de código.
  - Siempre va a ser un proceso más lento.
- O también **puedes buscar la solución en Internet**. Seguramente SQL disponga de opciones para hacer lo que quieres. Usarlas nos permite:
  - Hacer queries más rápidas, que ya SQL está pensado para ello.
  - Aprender una cosa nueva sobre SQL.
  - Reducir el código de Node JS de nuestro servidor.

Búsquedas en Internet

Como hemos comentado, cuando tengas alguna duda sobre cómo realizar una query de cualquier tipo, busca en Internet. Te recomendamos que al buscar escribas **SQL** y lo que estés buscando, por ejemplo:

- SQL WHERE with AND
- SQL filtrar por id
- SQL obtener los primeros 10 registros
- ...

Es decir, no busques por SQLite, sino por SQL, ya que ambos tienen la misma sintaxis pero en Internet hay muchas más páginas que hablan sobre SQL en general que sobre SQLite en particular.

Conclusiones

`SELECT` nos permite seleccionar o leer uno o varios registros de una tabla.

La sintaxis de `SELECT` es:

- `SELECT` seguido de asterisco para seleccionar todas las columnas o el nombre de las columnas separadas por comas
- `FROM` seguido del nombre de la tabla
- `WHERE` seguido de la condición o condiciones que deben cumplir los datos
- `ORDER BY` seguido del nombre de la columna por la que queremos ordenar, seguido de `ASC` o `DESC` que indica si la ordenación es ascendente o descendente
- `LIMIT` seguido del número máximo de registros que queremos seleccionar
- `OFFSET BY` seguido del número del primer elemento que queremos seleccionar

Para usar SQLite dentro de un servidor de Node JS debemos:

1. Instalar Better SQLite3
2. Crear una constante para la base de datos con `const Database = require('better-sqlite3');`
3. Indicar y configurar la base de datos que vamos a usar
4. Dentro de los endpoints debemos:
    1. Preparar la query con `const query = db.prepare('SELECT * FROM users');`
    2. Ejecutar la query con `const users = query.all();` o `const user = query.get();`
        - `query.all()` devuelve un array con objetos o un array vacío.
        - `query.get()` devuelve un objeto o `undefined`.
    3. Para pasar datos a la query debemos:
        1. Preparar la query con una o varias interrogaciones: `const query = db.prepare('SELECT * FROM users WHERE id = ?')`
        2. Ejecutar la query con los datos que sustituirán las interrogaciones: `const users = query.all(2);`

Ejercicios

1. El blog: tabla de artículos

Vamos a suponer que tenemos que crear una base de datos para gestionar un blog. Entre las muchas tablas que tendría la base de datos, nos toca crear una para almacenar los artículos escritos por la autora:

1. Crea un proyecto de Node JS; si quieres puedes utilizar el ejercicio del vídeo de esta sección.
2. Instala (si no lo has hecho ya) SQLite browser.
3. Crea una base de datos a través SQLite browser dentro del proyecto.
4. Crea una tabla llamada articles con las siguientes columnas:
    1. Título del artículo
    2. Cuerpo del artículo

    3. URL de la imagen principal
    4. Fecha de publicación
    5. Borrador o publicado
    6. Cualquier otro campo que te parezca bien
5. Antes de continuar:
    1. Revisa qué tipos y opciones le has puesto a cada columna.
    2. ¿Has puesto alguna columna para identificar cada artículo?
6. Crea 3 artículos rellenando todas las columnas. Haz que dos de ellos contengan las palabras **bases de datos** en el título.
7. Desde el editor SQL de SQLite browser ejecuta las siguientes queries:
    1. Selecciona todos los artículos.
    2. Selecciona el artículo con el artículo que tiene el `id = 2`.
    3. Selecciona todos los artículos que tengan en el título la palabra **datos**. Te ayudará buscar en Internet **SQL like**.

Después de cada consulta comprueba que los resultados que muestra SQLite browser tienen sentido.

2. La tienda online: tabla de libros

Ahora vamos a suponer que tenemos que crear una base de datos para una tienda online de libros. Sobre el proyecto del ejercicio anterior o sobre uno nuevo:

1. Crea una nueva tabla para guardar la siguiente información:
    1. Nombre del libro
    2. Autora del libro
    3. Resumen del libro
    4. Precio del libro
    5. Stock del libro
    6. ¿Es un libro descargable como un ebook o es un libro físico que debemos enviar por mensajería?
2. Crea 5 libros.
3. Crea una API en Node JS que devuelva la siguiente información en diferentes endpoints:
    1. Un array con todos los libros ordenados de menor a mayor precio
    2. Un array con los libros con precio superior a 5 €
    3. Un array con los libros en stock
    4. Un array con los libros físicos y en stock
    5. Un objeto con el libro con `id = 1`
    6. Un array con los 3 primeros libros ordenados alfabéticamente por nombre
    7. Un array con los 3 siguientes libros ordenados alfabéticamente por nombre

Todos los endpoints que hagas deben:

- Ser con el método `GET`.
- Recibir la información necesaria para hacer la petición por query params.

Revisa que la ruta de los endpoints sea coherente con lo que hacen.

> **Nota:** puedes acceder a tu API directamente desde Postman.

Netflix

Hasta ahora hemos estado trabajando con un servidor sin base de datos. Podríamos decir que nuestra base de datos han sido los ficheros `src/data/movies.json` y `src/data/users.json`. Ahora vamos a crear una base de datos donde vamos a guardar la información que tenemos en estos JSON.

Puesto que vamos a cambiar el código del servidor sin cambiar la funcionalidad, lo que estamos haciendo es una refactorización. El front no se va a enterar de ninguno de los cambios que hagamos en el back.

---

1. Crea la base de datos

1. Con SQLite crea una base de datos en `src/db/database.db`.
2. Añade una tabla llamada `movies` y añade los campos:
   - id
   - name
   - gender
   - image
3. A esta tabla añade los datos para que contenga al menos nuestras dos películas.
4. Guarda y ya está.

---

2. Configura la base de datos en Node JS

1. Instala better-sqlite3 con `npm install better-sqlite3`.
2. En `src/index.js` añade `const db = new Database('./src/db/database.db', { ... });` para decirle a Node que quieres usar esa base de datos.

---

3. Haz un SELECT para obtener todas películas

1. Ahora mismo, dentro del endpoint `GET:/movies` estás cogiendo los datos que has importado desde `src/data/movies.json`. Sustituye el código que tenías dentro de ese enpoint por otro que:
    1. Haga una consulta a la tabla `movies` y seleccione todas las películas.
    2. Responde a la petición con el array que te ha devuelto la base de datos. Si todo está bien, al arrancar tu web se seguirán mostrando las películas normalmente. React no se habrá enterado de que hemos empezado a usar bases de datos en el servidor.

> **Nota:** recuerda que si quieres obtener varios registros de la base de datos debes usar `query.all();` y si solo quieres uno debes usar `query.get();`.

---

4. Mejora tu SELECT

Con el ejercicio anterior se te habrá roto la funcionalidad de filtrar por género y ordenar alfabéticamente. Vamos a arreglarlo.

1. Modifica tu endpoint y tu query para que:
    - Si recibes por query params el género, la base de datos devuelva solo aquellas películas que coinciden con ese género.
    - Si no recibes por query params el género, la base de datos devuelva todas las películas.
2. Modifica la query o las queries que acabas de crear para que siempre devuelvan las películas ordenadas alfabéticamente por nombre:
    - Si recibes el query param `sort = desc`, ordena los resultados de la query de forma descendente.
    - En cualquier otro caso ordena los resultados de forma ascendente.

---

5. SELECT para el motor de plantillas

Si en ejercicios anteriores has hecho el motor de plantillas estarás en un punto incoherente. Para el endpoint `GET:/movies` estarás sacando los datos desde la base de datos, pero para el endpoint `GET:/movie/:movieId` los estarás obteniendo de `src/data/movies.json`.

Para solucionarlo tienes que:

1. Modificar el endpoint `GET:/movie/:movieId` para que no saque los datos de `src/data/movies.json`.
2. En este endpoint haz una query para obtener la película que tenga el id que estás recibiendo por URL params.
3. Ya podrías borrar la importación de `movies.json` que estás haciendo en `src/index.js`, porque en teoría ya no la estás usando.

4. Para comprobar que lo has hecho bien entra en http://localhost:4000/movie/1; a través del motor de plantilas se debería renderizar la película Gambita de dama.

> **Nota:** recuerda que si quieres obtener varios registros de la base de datos debes usar `query.all();`, y si solo quieres uno debes usar `query.get();`.

---

6. Crea la tabla de usuarias

Hasta ahora también estamos cogiendo los datos de las usuarias desde `src/data/users.json`. Repite los pasos que hemos hecho antes para dejar de utilizar este fichero y utilizar la base de datos.

1. Crea una nueva tabla que se llame `users`.
2. Añade los campos.
3. Añade los datos.
4. Haz un SELECT en el endpoint POST:/login con los datos que recibes por body params.
5. Si la query devuelve a una usuaria, retorna su id y si no, retorna el mensaje de error.
6. Borra la carpeta `src/data/` porque ya no la vamos a utilizar.

4.6 Bases de datos II

6.1 SQL Tipos de campos

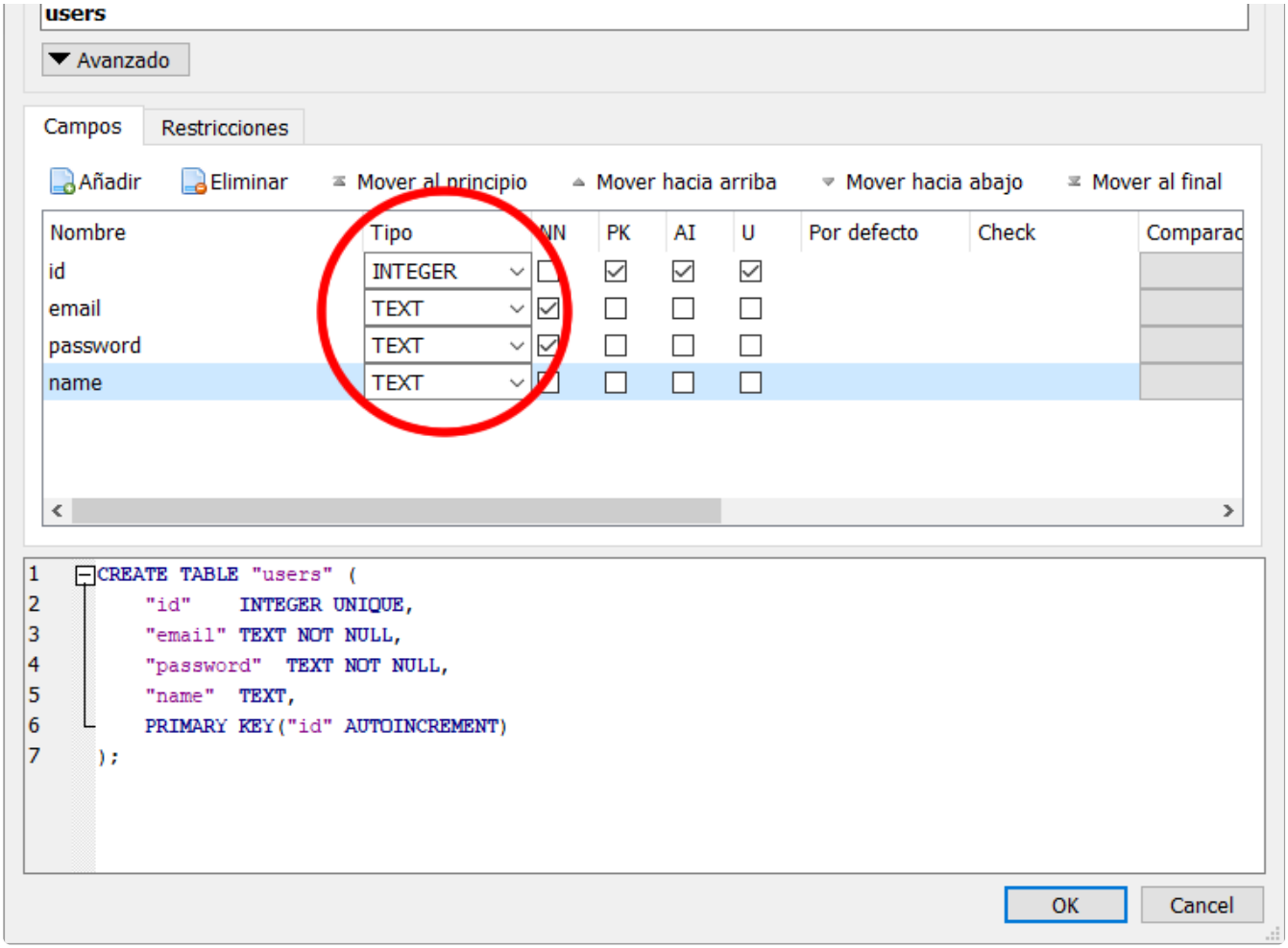
> **Nota:** esta mini lección es importante.

Como hemos comentado en las primeras lecciones de SQL, esta es una **base de datos con una estructura muy estricta**. Esto nos permite controlar muy bien qué datos se están guardando en nuestras tablas.

Cuando estamos creando las tablas de nuestra base de datos podemos elegir **qué tipo de dato** se puede guardar **en cada columna**.

Tipos de datos en SQLite

Podemos configurar el tipo de dato de cada columna de cada tabla de SQLite. Se configuran editando la tabla como muestra esta imagen:



Tipos de columnas en SQLite

- **INTEGER:** para números enteros (sin decimales), positivos y negativos.
- **TEXT:** para textos de cualquier tamaño.
- **BLOB:** para guardar información de cualquier tipo, se suele usar para guardar ficheros en modo binario.
- **REAL:** para guardar números con muchos decimales sin exactitud. Esto de la exactitud es una movida muy chunga de cómo se guardan los datos en el disco duro de un ordenador.
- **NUMERIC:** para guardar números con muchos decimales con exactitud, booleanos, fechas...

SQLite no es tan estricta como otras bases de datos:

- Si intentamos guardar el `string` `'1'` en un campo `integer` en una columna, lo intenta convertir a número y lo guarda como el entero 1. Es decir, **lo va a guardar con el tipo de dato que hemos configurado** en dicha columna.
- Si intentamos guardar el `string` `'Hola'` en un campo `integer` en una columna, no lo puede convertir a número, por lo que guardará el `string` `'Hola'`. Es decir, **no lo guarda con el tipo de dato que hemos configurado** en dicha columna. **Es nuestra responsabilidad no hacer esto.**
- Si intentamos guardar el `string` `'Hola'` en el campo `id`, que es de tipo `integer`, nos lanzará un error. **Esto es porque necesita que todos los `id` sean enteros.**

Seleccionar bien los tipos de columnas

Cuando creamos o editamos una tabla debemos pensar muy bien qué tipo de dato queremos guardar en cada columna. Si empezamos a guardar los datos en un tipo de datos inadecuado, nos será muy difícil cambiarlo en el futuro sin perder datos.

Tipos de datos en MySQL

Ya sabemos que dentro de SQL hay muchas familias. Os queremos enseñar **los tipos de datos que hay en MySQL** con el objetivo de que veáis otras bases de datos:

- Tipos de datos de texto:
  - **CHAR:** para guardar caracteres sueltos: `a`, `z`, `A`, `Z`, `1`, `@`, `€`...
  - **VARCHAR:** para guardar textos con tamaño máximo que podemos configurar nosotras.
  - **TINYTEXT:** para guardar textos de un máximo de 255 caracteres.
  - **TEXT:** para guardar textos de un máximo de 65535 caracteres.
  - **MEDIUMTEXT:** para guardar textos de un máximo de 16777215 caracteres.
  - **LONGTEXT:** para guardar textos de un máximo de 4294967295 caracteres.
  - **SET:** para guardar textos de un determinado conjunto. Por ejemplo si queremos guardar el rol de la usuaria, podemos indicar que solo se puedan guardar los textos: `user`, `admin` y `customer`. Si guardamos otro `string` diferente MySQL lanzará un error.
- Tipos de datos numéricos:
  - **BOOL:** para guardar los números enteros 0 o 1. Muy útil para guardar booleanos. Si guardamos un 0 estamos guardando un `false`, si guardamos un 1 estamos guardando un `true`.
  - **TINYINT:** para guardar enteros entre 0 y 255.
  - **SMALLINT:** para guardar enteros entre 0 y 65535.
  - **MEDIUMINT:** para guardar enteros entre 0 y 16777215.
  - **INTEGER:** para guardar enteros entre 0 y 4294967295.
  - **BIGINT:** para guardar enteros entre 0 y 9223372036854775807.
  - **FLOAT:** para guardar números con decimales.
- Tipos de datos de fechas:
  - **DATE:** para guardar fechas en formato YYYY-MM-DD.
  - **DATETIME:** para guardar fechas en formato YYYY-MM-DD hh:mm:ss.
  - **TIMESTAMP:** para guardar fechas en milisegundos. Concretamente los milisegundos transcurridos desde el 1 de enero de 1970. Esto es otra movida... pero nos permite saber una fecha independientemente de los husos horarios.

Y estos son solo algunos, hay muchos más.

MySQL es muy estricta, y si intentamos guardar un dato en una columna que no corresponde con el tipo de dato nos lanza un error.

Beneficios de configurar bien los tipos de datos

Tamaño de las bases de datos

Como hemos visto, en MySQL cada dato tiene un tamaño máximo. **Eso significa que la base de datos reserva espacio en el disco duro del ordenador para cada dato.**

Supongamos que estamos trabajando con una **base de datos MySQL** y queremos guardar la edad de nuestras usuarias. Debemos utilizar el tipo de dato **TINYINT**, porque queremos guardar una edad que es un número entero entre 0 y 255. No creo que nadie llegue a los 255 años de edad, pero no tenemos un tipo de datos más pequeño que **TINYINT**.

Si en vez de configurar nuestra columna con un **TINYINT** la configuramos con un **BIGINT (enteros entre 0 y 9223372036854775807)** la base de datos estará reservando mucho espacio en el disco duro de nuestro ordenador, por si intentamos guardar la edad 9223372036854775807, cosa que, con toda probabilidad, no va a pasar nunca.

Pensemos que las aplicaciones que haréis cuando trabajéis en una empresa pueden tener varios miles de usuarios, lo que equivale a varios GigaBytes de espacio. La base de datos de GitHub o Twitter puede tener muchos TeraBytes. Como no optimicemos bien el espacio lo estaremos desperdiciando. **Por cierto, los discos duros cuestan dinero.** Si optimizamos bien la base de datos le estaremos ahorrando a la empresa varios cientos de euros al mes.

También imaginemos cuánto tardaría nuestro ordenador en abrir con SQLite browser una base de datos que pese 100 Gigas.

Tiempo de búsquedas

Cada vez que leemos o escribimos en una base de datos, ésta tiene que buscar entre todos los registros cuál es el registro que queremos leer o modificar.

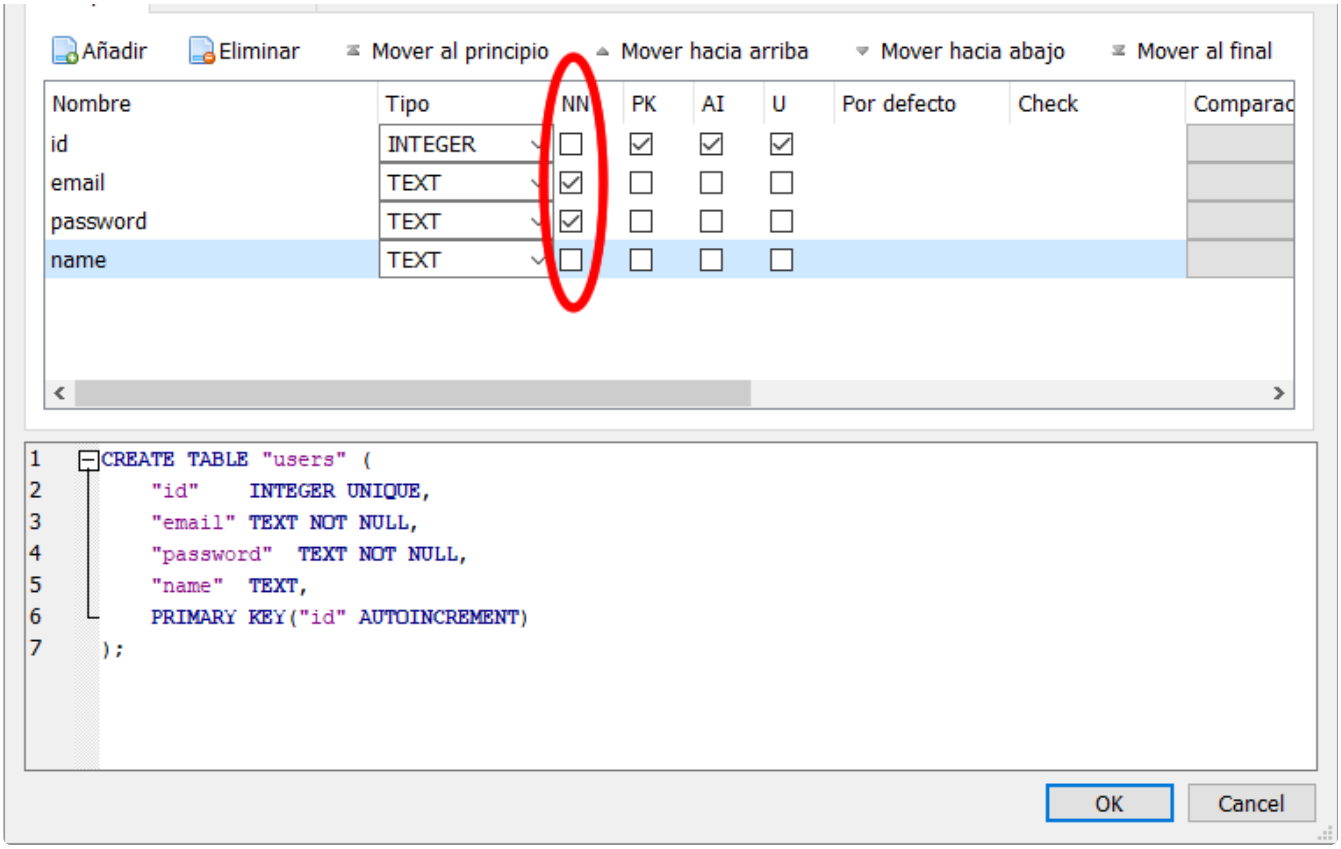
Pensemos que al abrir un email de Gmail los servidores de Google estarán haciendo entre 20 y 50 accesos a sus bases de datos.

Si no optimizamos las bases de datos, las aplicaciones **tienen un cuello de botella** en el acceso a sus datos.

Tipos de datos nulos

Supongamos que estamos programando el registro de usuarias en nuestra web. Podemos pensar que es importante no permitir que ninguna usuaria se registre sin indicar su email o contraseña. SQLite nos da una solución para hacerlo.

SQLite tiene una opción para indicar que una columna debe contener siempre datos, es decir, **ningún registro de esta tabla puede tener vacío el campo de esa columna**:



SQLite Not null

Si activamos esta opción en las columnas email y password, cuando añadimos un registro o lo modificamos y no indicamos el valor del email o el del password, SQLite nos lanza un error y no ejecutará la query.

También debemos programar en Node JS que esto no pueda pasar, pero si a la vez lo activamos en SQLite browser estamos duplicando la seguridad.

Columna especial id

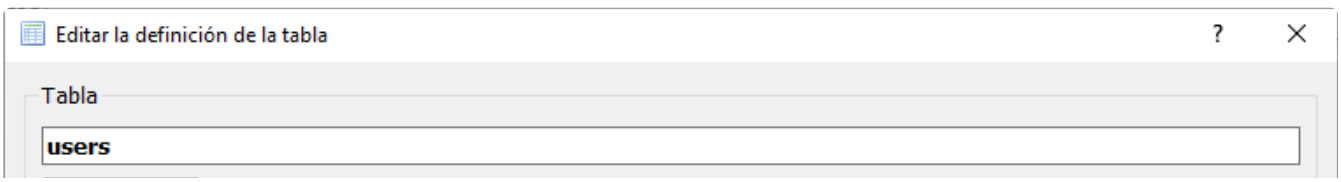
En todos los ejemplos que hemos visto hasta ahora siempre hemos configurado la primera columna como `id`.

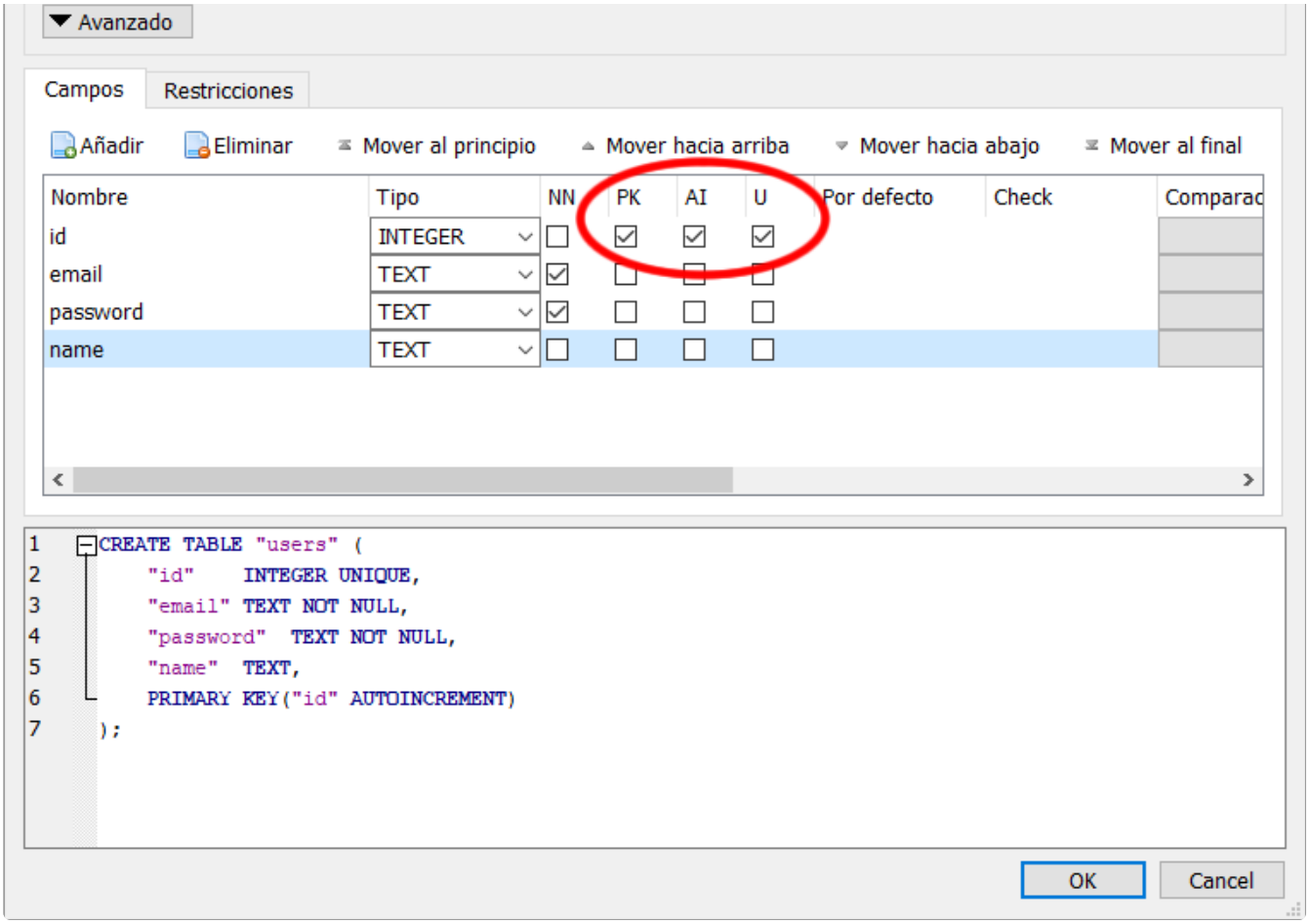
Esta columna funciona como un identificador único, como el DNI de cada registro. De esta forma nos aseguramos de que, aunque cualquier otro campo cambie, este nunca va a cambiar.

No es obligatorio tener una columna `id` en una tabla, **pero es muy muy recomendable**. Así que siempre que creamos una nueva tabla le vamos a poner como primera columna el `id`.

Tampoco es obligatorio que esta columna se llame `id`, puede llamarse `userId`, `user_id`, `emailId`, `tweetId` o como queramos.

También te habrás fijado en que en la columna `id` activamos siempre las opciones `PK`, `AI` y `U`. **Debemos activar estas tres opciones en todas las columnas `id` de todas nuestras tablas.**





SQLite Not null

Primary key (PK)

Activando en la columna `id` la **primary key** le estamos indicando a la base de datos que esta es nuestra columna principal, la que contiene el identificador único.

Gracias a esto podemos relacionar unas tablas con otras y automatizar tareas.

Supongamos que tenemos **dos tablas** en una base de datos de una tienda online:

- `users`, que tiene la `PK` activada en la columna `id`.
- `orders`, que guarda los pedidos de las usuarias y que tiene una columna llamada `userId` en la que indicamos que un pedido pertenece a una usuaria en concreto.

Gracias a primary key de `users` y a un poco de configuración podríamos automatizar tareas, como por ejemplo que si borramos una usuaria de la tabla `users`, automáticamente la base de datos borrará todos los pedidos de esa usuaria en la tabla `orders`. De esta forma no habría nunca pedidos "huérfanos".

Autoincrement (AI)

Cuando aprendamos a añadir registros a una tabla verás que nunca indicamos el `id` de ese registro. Eso es porque SQLite lo añade automáticamente por nosotras.

Y ¿cómo sabe qué `id` debe asignar al nuevo registro? Pues porque le hemos dicho que es auto incremental. La primera vez que añadimos un registro, ya sea desde SQLite browser o desde Node JS, le

asigna el `id = 1`. La segunda vez el `id = 2`, y así sucesivamente.

De esta forma cada registro tiene un `id` único que no se repite.

Unique (U)

Como hemos dicho, el `id` debe ser único. Pues si configuramos el `id` como `unique` nunca podrá haber en una tabla dos registros con el mismo `id`.

Si intentamos modificar el `id` de un registro y ya existe otro con el mismo `id` SQLite nos lanzará un error y no ejecutará la query.

Esto también es útil para otras columnas como el email. Si consideramos que no puede haber dos usuarias en nuestra web con el mismo email, podemos marcar la opción `U` en la columna email. Si la misma usuaria intenta registrarse dos veces con el mismo email SQLite lanzará un error y no lo permitirá.

Otras opciones de SQLite

Por defecto

Podemos configurar la opción **Por defecto** en una columna. Si al añadir un registro a la tabla no indicamos ningún valor para esa columna, SQLite lo añadirá por nosotras poniéndole el valor por defecto.

Check

Podemos marcar la opción **Check** en una columna para indicar algún tipo de restricción que debe cumplir.

Por ejemplo, supongamos que creamos una columna llamada `phone` y en la opción **check** ponemos `length(phone) >= 9`. Si añadimos un nuevo registro y en el valor de la columna `phone` introducimos un número de teléfono de menos de 9 dígitos, SQLite lanzará un error y no ejecutará la consulta.

6.2 SQL Update

> **Nota:** esta mini lección es importante.

`UPDATE` nos permite **modificar uno varios registros de una tabla** de la base de datos a través de una query.

Así como leer registros de una tabla no modifica la base de datos, actualizar uno o varios registros sí que la modifica.

Sintaxis de UPDATE en SQL y SQLite browser

La sintaxis de un `UPDATE` es:

```
```

```
UPDATE nombre_de_la_tabla
SET nombre_de_una_columna = valor_de_una_columna, nombre_de_otra_columna = valor_de_otra_colu
WHERE condicion_que_cumplen_los_registros_a_modificar
```

Supongamos que tenemos esta tabla llamada `users`:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía
3	sofia@yahoo.com	mnbvcdfgu	Sofía

Con la query:

```
-- Actualizar el email de la usuaria que tiene el id = 3
UPDATE users SET email = 'sofia.garcia@yahoo.com' WHERE id = 3
```

La tabla quedará así:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía
3	**sofia.garcia@yahoo.com**	mnbvcdfgu	Sofía

Por otro lado, con la query:

```
-- Actualizar el email y la contraseña de la usuaria que tiene el id = 3
UPDATE users SET email = 'sofia.garcia@yahoo.com', password = 'abcdefgh' WHERE id = 3
```

La tabla quedará así:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía
3	**sofia.garcia@yahoo.com**	**abcdefgh**	Sofía

¡¡¡UPDATE sin WHERE es igual a muerte, destrucción y caos!!!

Si hacemos una query para modificar registros de una tabla y no indicamos la condición `WHERE`, **se modificarán todos los registros de la tabla**. ¡¡¡Todos!!! Esto es porque todos los registros cumplen la condición vacía.

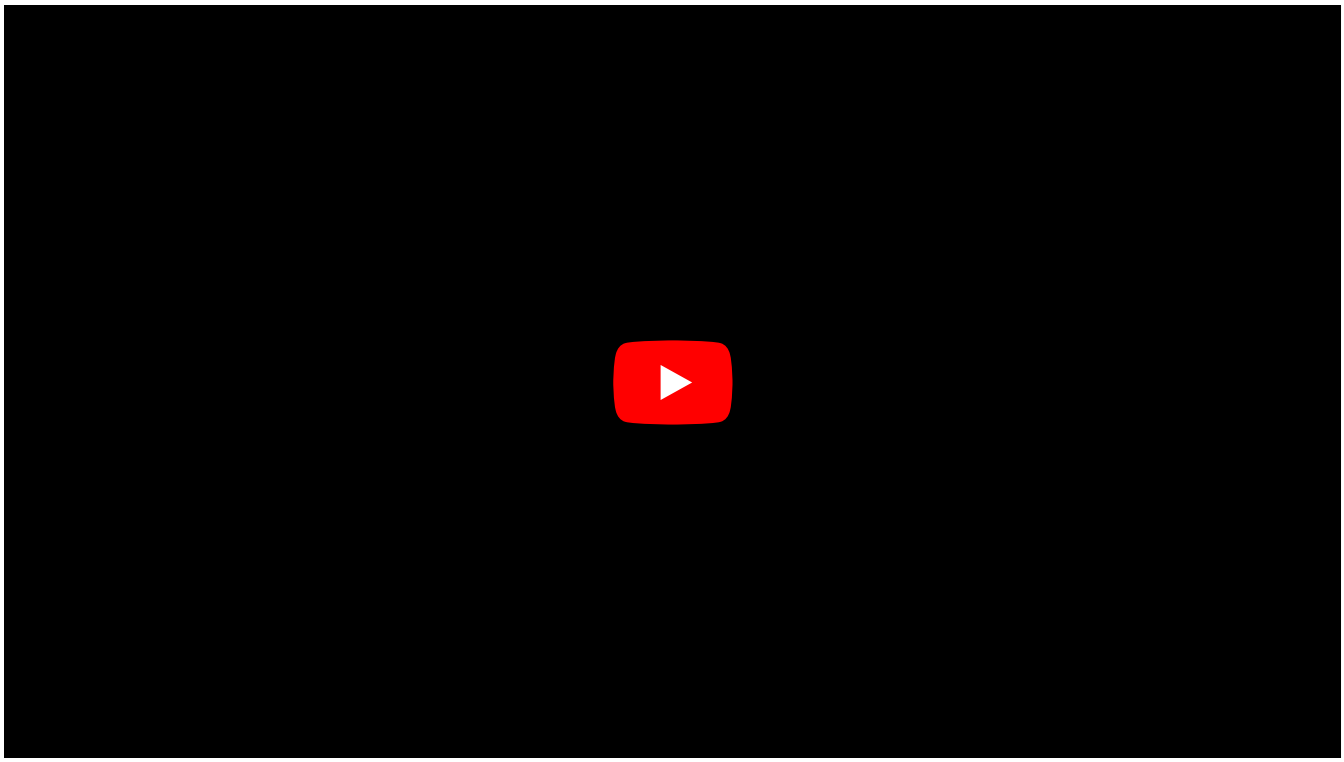
La condición `WHERE` no es obligatoria, pero pocas veces vamos a querer cambiar todos los registros de una tabla. Así que cuando hagas un `UPDATE` piensa dos veces si debes o no poner `WHERE`.

Con `UPDATE` y `WHERE` podemos actualizar uno o varios registros en una sola query. Todo depende de la condición que pongamos en el `WHERE`.

No debemos modificar el campo id

Anteriormente hemos comentado que el campo `id` es como el DNI del registro. Por ello, aunque se puede, no debemos modificarlo, pues nos puede dar muchos problemas.

Vamos a ver todo lo que hemos explicado hasta ahora en este vídeo:



> Ejercicio del vídeo

Sintaxis de UPDATE en Node JS y Better SQLite 3

Antes de modificar uno o varios registros desde Node JS, deberíamos seguir los pasos que hemos aprendido para trabajar con **Better SQLite 3**:

1. **Instalar Better SQLite 3** en el proyecto con `npm install better-sqlite3`
2. **Importar Better SQLite 3** en el proyecto con `const Database = require('better-sqlite3');`
3. **Iniciar y configurar la base de datos** con `const db = new`

`Database('./src/database.db' });`

Una vez hecho esto ya podemos modificar registros. La forma de trabajar es igual que con `SELECT`:

```
// preparamos la query
const query = db.prepare(
  'UPDATE users SET email = ?, password = ? WHERE id = ?'
);
// ejecutamos la query
const result = query.run('sofia.garcia@yahoo.com', 'abcdefgh', 3);
```

Normalmente haremos nuestras queries desde dentro de un endpoint de Express JS. Por ello el código sería:

```
app.post('/users', (req, res) => {
  const query = db.prepare(
    'UPDATE users SET email = ?, password = ? WHERE id = ?'
  );
  const result = query.run('sofia.garcia@yahoo.com', 'abcdefgh', 3);
  res.json(result);
});
```

Normalmente nos enviarán el email y contraseña desde el navegador con un `fetch` por lo que el código sería algo como:

```
app.post('/users', (req, res) => {
  const query = db.prepare(
    'UPDATE users SET email = ?, password = ? WHERE id = ?'
  );
  const result = query.run(req.body.email, req.body.password, req.body.id);
  console.log(result);
  res.json(result);
});
```

query.run()

En estas queries no estamos leyendo datos de la tabla, por ello no utilizamos `query.all()` ni `query.get()`. En esta query utilizamos `query.run()` porque lo que queremos es modificar registros. Los creadores de Better SQLite 3 han elegido esta forma de trabajar porque les ha apetecido, pero si lo piensas tiene bastante lógica.

Información retornada por query.run()

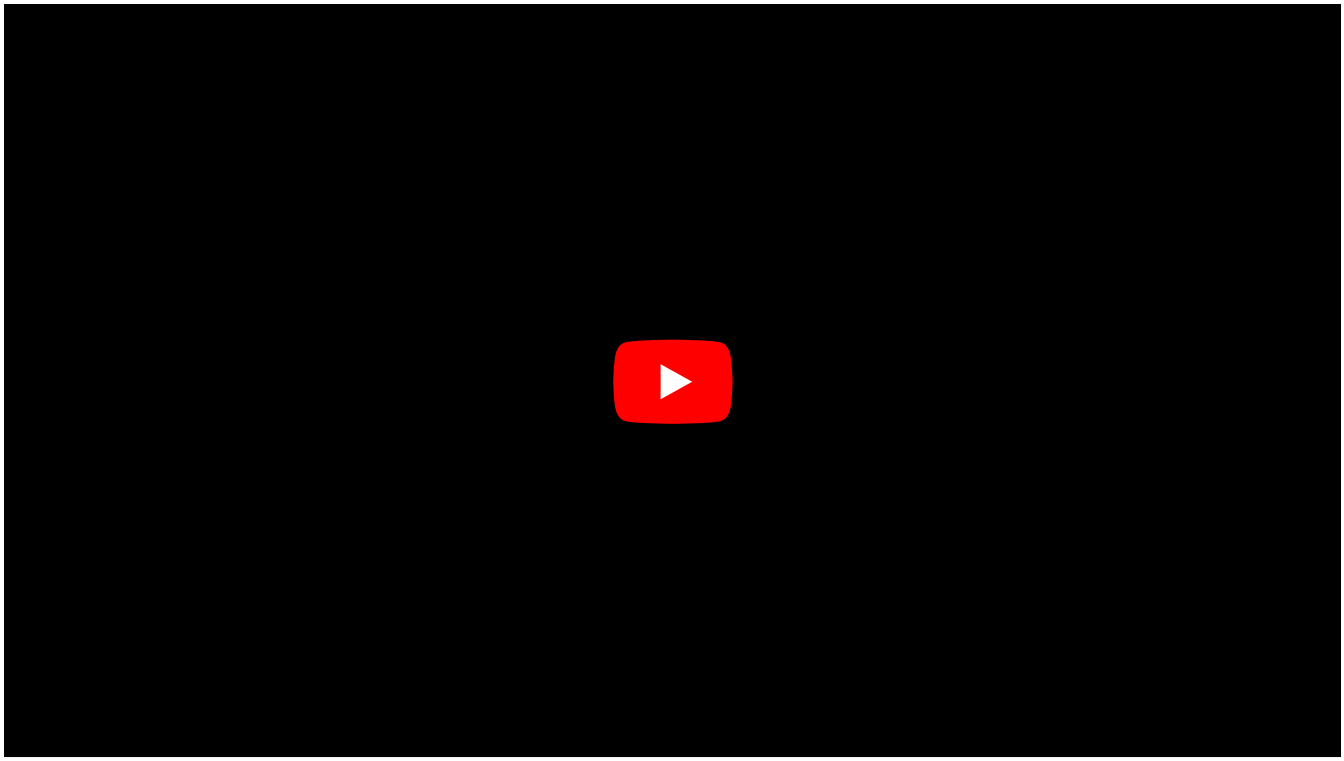
Cuando ejecutamos una query de actualización, a veces nos interesa saber cuántos registros se han modificado.

La información retornada por `query.run()` en el código anterior la estamos consoleando. Por ello la consola mostrará:

```
{
  "changes": 1, // se ha cambiado o añadido un registro
  "lastInsertRowid": 0 // no se ha añadido ningún registro nuevo
}
```

En las consultas de tipo `UPDATE` el valor de `lastInsertRowid` siempre es 0 porque no hemos añadido ningún registro nuevo.

Vamos a ver todo lo que hemos explicado hasta ahora en este vídeo:



> Ejercicio del vídeo

Conclusiones

`UPDATE` nos permite modificar uno o varios registros de una tabla.

La sintaxis de `UPDATE` es:

```
UPDATE nombre_de_la_tabla
SET nombre_de_una_columna = valor_de_una_columna, nombre_de_otra_columna = valor_de_otra_colu
WHERE condicion_que_cumplen_los_registros_a_modificar
```

Para usar SQLite dentro de un servidor de Node JS debemos:

1. Preparar la query con `const query = db.prepare('UPDATE users SET email = ?, password = ? WHERE id = ?');`.
2. Ejecutar la query con `const result = query.run('sofia.garcia@yahoo.com',`

```
'abcdefgh', 3);
```

- `query.run()` retorna:

```
{
  "changes": 1, // número de registros modificados
  "lastInsertRowid": 0 // este dato siempre es 0 en consultas de tipo UPDATE
}
```

Ejercicios:

1. Actualizando libros

Partiendo del ejercicio 2 de la lección de **SQL SELECT**, añade nuevos endpoints a tu API para que hagan los siguientes cambios en la tabla `books`:

- Crea un endpoint de tipo `PATCH` que modifique el título de un libro y:
  - Reciba por URL params el `id` del libro que quieres modificar.
  - Reciba por body params el nuevo título del libro.
- Crea un endpoint de tipo `PATCH` que cambie la cantidad de stock de todos los libros y:
  - Reciba por body params el nuevo valor del stock.
- Modifica el endpoint anterior para que solo se modifiquen la cantidad de stock de los libros que son físicos. Piensa que los ebooks no tienen stock.
- Crea un endpoint de tipo `PUT` que cambie todos los datos de un libro y:
  - Reciba por URL params el `id` del libro que quieres cambiar.
  - Reciba por body params todos los datos del libro que quieres cambiar.

Revisa que la ruta de los endpoints sea coherente con lo que hacen.

> **Nota:** puedes acceder a tu API directamente desde Postman.

6.3 SQL Insert

> **Nota:** esta mini lección es importante, pero es parecida a la anterior. Puedes leerla por encima y revisarla en un par de días.

`INSERT INTO` nos permite **añadir un registro a una tabla** de la base de datos a través de una query.

Así como leer registros de una tabla no modifica la base de datos, añadir un registro sí que la modifica.

Sintaxis de INSERT INTO en SQL y SQLite browser

La sintaxis de un `INSERT INTO` es:

```
INSERT INTO nombre_de_la_tabla (nombre_de_una_columna, nombre_de_otra_columna)
VALUES (valor_de_una_columna, valor_de_otra_columna)
```

Supongamos que tenemos esta tabla llamada `users`:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía
3	sofia@yahoo.com	mnbvcdfgu	Sofía

INSERT INTO indicando solo algunas columnas

Con la query:

```
-- Añadir un nuevo registro indicando algunas columnas y el valor de cada columna
INSERT INTO users (email, password) VALUES ('celia@gmail.com', 'fas09fn32');
```

La tabla quedará así:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía
3	sofia@yahoo.com	mnbvcdfgu	Sofía
4	**celia@gmail.com**	**fas09fn32**	

Como no hemos añadido la columna `name` en la query, la usuaria 4 no tiene nombre.

INSERT INTO indicando todas las columnas

Si a continuación añadimos otro registro con la query:

```
-- Añadir un nuevo registro indicando todas columnas y el valor de cada columna
INSERT INTO users (email, password, name) VALUES ('tania@gmail.com', '09df34D43', 'Tania');
```

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía

3	sofia@yahoo.com	mnbvcdfgu	Sofía
4	celia@gmail.com	fas09fn32	
5	**tania@gmail.com**	**09df34D43**	Tania

Orden de las columnas

Al hacer estas queries tenemos que preocuparnos de que el orden de las columnas coincida con el orden de los valores. Sin embargo, no es necesario que las columnas estén en el mismo orden que en la tabla. Esta query:

```
INSERT INTO users (email, password, name) VALUES ('tania@gmail.com', '09df34D43', 'Tania');
```

Es igual que esta:

```
INSERT INTO users (name, password, email) VALUES ('Tania', '09df34D43', 'tania@gmail.com');
```

Valor de la columna id

La columna `id` es una columna especial que no rellenamos nosotras si no que va a ser rellenada por SQL. Al crear una base de datos debemos activar la opción **auto increment** para la columna `id`. De esta manera, cada vez que añadimos un nuevo registro, SQL calcula automáticamente el valor que debe tener el `id`.

Lo importante es que **nunca debemos indicar el valor de la columna `id`**.

Cómo añadir varios registros

Si queremos añadir varios registros lo tendremos que hacer en varias queries. Para ello tendremos que hacer un `for` en Node JS ejecutando varias veces la query.

Vamos a ver todo lo que hemos explicado hasta ahora en este vídeo:

> Ejercicio del vídeo

Sintaxis de INSERT INTO en Node JS y Better SQLite 3

Antes de añadir un nuevo registro desde Node JS, debemos seguir los pasos que hemos aprendido para trabajar con **Better SQLite 3**:

1. **Instalar Better SQLite 3** en el proyecto con `npm install better-sqlite3`
2. **Importar Better SQLite 3** en el proyecto con `const Database = require('better-sqlite3');`
3. **Iniciar y configurar la base de datos** con `const db = new Database('./src/database.db' });`

Una vez hecho esto ya podemos añadir nuevos registros. La forma de trabajar es igual que con `UPDATE`:

```
// preparamos la query
const query = db.prepare('INSERT INTO users (email, password) VALUES (?, ?)');
// ejecutamos la query
const result = query.run('celia@gmail.com', 'fas09fn32');
```

Normalmente haremos nuestras queries desde dentro de un endpoint de Express JS. Por ello el código sería:

```
app.post('/users', (req, res) => {
  const query = db.prepare('INSERT INTO users (email, password) VALUES (?, ?)');
  const result = query.run('celia@gmail.com', 'fas09fn32');
  res.json(result);
});
```

Normalmente nos enviarán el email y contraseña desde el navegador con un `fetch`. Por ello el código sería algo como:

```
app.post('/users', (req, res) => {
  const query = db.prepare('INSERT INTO users (email, password) VALUES (?, ?)');
  const result = query.run(req.body.email, req.body.password);
  console.log(result);
  res.json(result);
});
```

query.run()

En estas queries no estamos leyendo datos de la tabla, por eso no utilizamos `query.all()` ni `query.get()`. En esta query utilizamos `query.run()` porque lo que queremos es añadir registros. Los creadores de Better SQLite 3 han elegido esta forma de trabajar porque les ha apetecido, pero si lo piensas tiene bastante lógica.

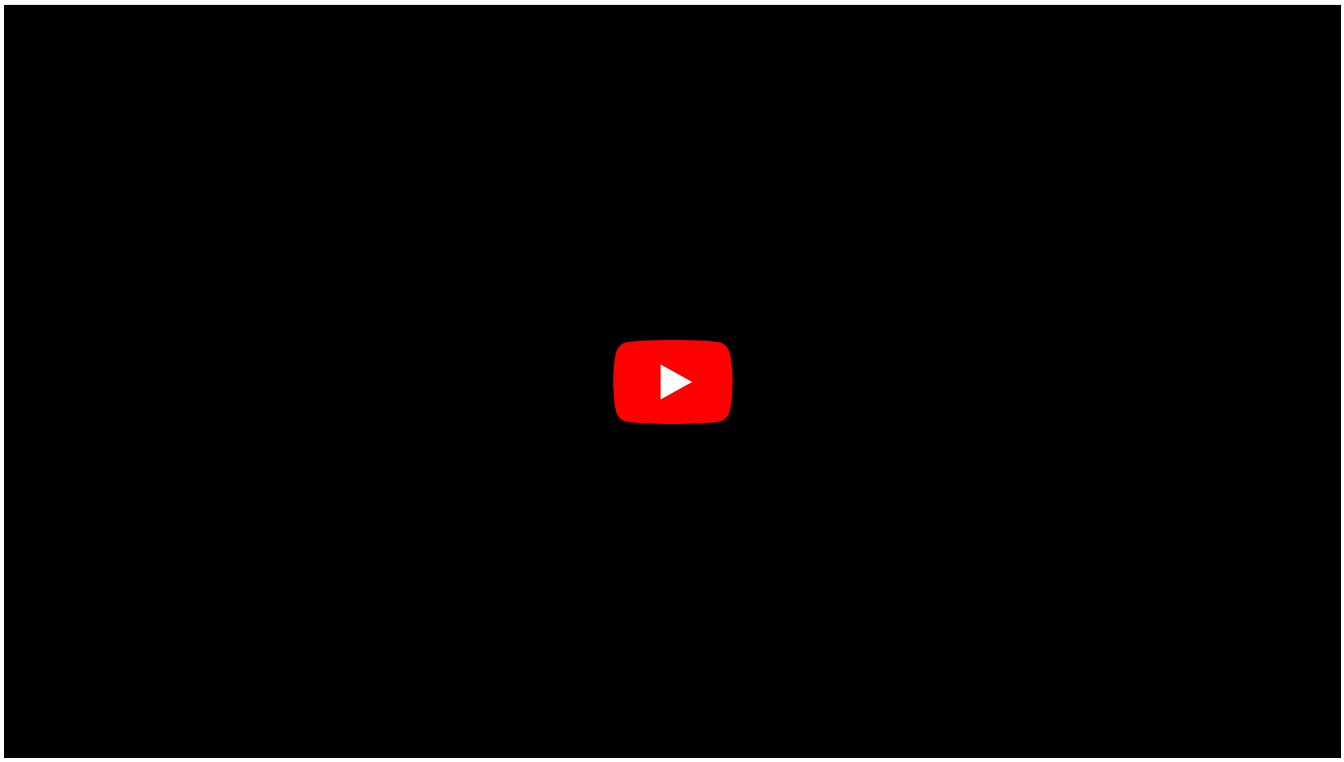
Información retornada por query.run()

Al añadir un nuevo registro nos interesa saber el `id` que SQL le ha asignado. Lo normal es responder a la petición hecha desde el navegador con el `id` de la usuaria creada. Así el navegador podrá hacer nuevas peticiones al servidor indicando qué usuaria está haciendo las peticiones.

Este `id` es retornado por `query.run()`. Por ello la consola del código anterior mostrará:

```
{
  "changes": 1, // se ha cambiado o añadido un registro
  "lastInsertRowid": 6 // el id del registro añadido es 6, porque hasta ahora había 5 registr
}
```

Vamos a ver todo lo que hemos explicado hasta ahora en este vídeo:



> Ejercicio del vídeo

Conclusiones

`INSERT INTO` nos permite añadir un registro a una tabla.

La sintaxis de `INSERT INTO` es:

```
INSERT INTO nombre_de_la_tabla (nombre_de_una_columna, nombre_de_otra_columna)
VALUES (valor_de_una_columna, valor_de_otra_columna)
```

Para usar SQLite dentro de un servidor de Node JS debemos:

1. Preparar la query con `const query = db.prepare('INSERT INTO users (email, password) VALUES (?, ?)');`.
2. Ejecutar la query con `const result = query.run('celia@gmail.com', 'fas09fn32');`
   - `query.run()` retorna:

   ```
   {
     "changes": 1, // número de registros añadidos
     "lastInsertRowid": 6 // id del registro añadido
   }
   ```

Ejercicios

1. Añadiendo nuevos libros

Partiendo del ejercicio 1 de la lección `SQL UPDATE`, añade nuevos endpoints a tu API para meter nuevos libros en la base de datos:

- Crea un endpoint de tipo `POST` que añada un nuevo libro:
  - Y que reciba por body params todos los datos del nuevo libro.
  - Y que responda con el `id` del libro creado.
- Mira en SQLite browser qué ocurre si desde Postman atacas a ese endpoint sin pasar ninguno de los datos del libro.
- Modifica el endpoint anterior para hacer que algunos campos sean obligatorios:
  - Los campos obligatorios son el título y la autora.
  - Si el endpoint no recibe alguno de esos datos o están vacíos debes responder a la petición con el código de error 400 y el mensaje `{ error: 'Invalid input data }`.
  - Si el endpoint recibe el título y la autora pero no recibe ningún otro dato, debes poner por defecto los campos que vengan vacíos, por ejemplo, el valor por defecto del stock es 0.

6.4 SQL Delete

> **Nota:** esta mini lección es importante, pero es parecida a la anterior. Puedes leerla por encima y revisarla en un par de días.

`DELETE` nos permite **borrar uno o varios registros de una tabla** de la base de datos a través de una query.

Así como leer registros de una tabla no modifica la base de datos, borrar uno o varios registros sí que la modifica.

Sintaxis de DELETE en SQL y SQLite browser

La sintaxis de un `DELETE` es:

```
DELETE FROM nombre_de_la_tabla WHERE condicion_que_cumplen_los_registros_a_borrar
```

Supongamos que tenemos esta tabla llamada `users`:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
2	lucia@hotmail.com	qwertyui	Lucía
3	sofia@yahoo.com	mnbvcdfgu	Sofía

Con la query:

```
-- Borrar un registro indicando el id del registro a borrar
DELETE FROM users WHERE id = 2;
```

La tabla quedará así:

id	email	password	name
1	maria@gmail.com	987widJYVxyh	María
3	sofia@yahoo.com	mnbvcdfgu	Sofía

¡¡¡DELETE sin WHERE es igual a muerte, destrucción y caos!!!

Si hacemos una query para borrar registros de una tabla y no indicamos la condición `WHERE` **se borrarán todos los registros de la tabla**. ¡¡¡Todos!!! Esto es porque todos los registros cumplen la condición vacía.

La condición `WHERE` no es obligatoria, pero no se nos debe olvidar nunca a no ser que queramos vaciar una tabla.

Con `DELETE` y `WHERE` podemos borrar uno o varios registros en una sola query. Todo depende de la condición que pongamos en el `WHERE`.

Y para que no se nos olvide hemos compuesto esta canción.

Vamos a ver todo lo que hemos explicado hasta ahora en este vídeo:

> Ejercicio del vídeo

Sintaxis de DELETE en Node JS y Better SQLite 3

Antes de borrar un nuevo registro desde Node JS, deberíamos seguir los pasos que hemos aprendido para trabajar con **Better SQLite 3**:

1. **Instalar Better SQLite 3** en el proyecto con `npm install better-sqlite3`.
2. **Importar Better SQLite 3** en el proyecto con `const Database = require('better-sqlite3');`.
3. **Iniciar y configurar la base de datos** con `const db = new Database('./src/database.db' });`.

Una vez hecho esto ya podemos borrar registros. La forma de trabajar es igual que con `UPDATE` o `INSERT INTO`:

```
// preparamos la query
const query = db.prepare('DELETE FROM users WHERE id = ?');
// ejecutamos la query
const result = query.run(2);
```

Normalmente haremos nuestras queries desde dentro de un endpoint de Express JS. Por ello el código sería:

```
app.delete('/user', (req, res) => {
  const query = db.prepare('DELETE FROM users WHERE id = ?');
  const result = query.run(2);
  res.json(result);
});
```

> **Nota:** con `app.delete("/user", (req, res) => { ...` estamos creando el endpoint `/user` con el verbo DELETE. Hasta ahora solo hemos trabajado con los verbos GET y POST pero hay muchos más.

Normalmente nos enviarán el `id` desde el navegador con un `fetch`. Por ello el código sería algo como:

```
app.delete('/user', (req, res) => {
  const query = db.prepare('DELETE FROM users WHERE id = ?');
  const result = query.run(req.body.id);
  console.log(result);
  res.json(result);
});
```

query.run()

En estas queries no estamos leyendo datos de la tabla, por ello no utilizamos `query.all()` ni `query.get()`. En esta query utilizamos `query.run()` porque lo que queremos es borrar registros. Los creadores de Better SQLite 3 han elegido esta forma de trabajar porque les ha apetecido, pero si lo piensas tiene bastante lógica.

Información retornada por query.run()

Cuando ejecutamos una query de borrado, a veces nos interesa saber cuántos registros se han borrado.

La información retornada por `query.run()` en el código anterior la estamos consoleando. Por ello la consola mostrará:

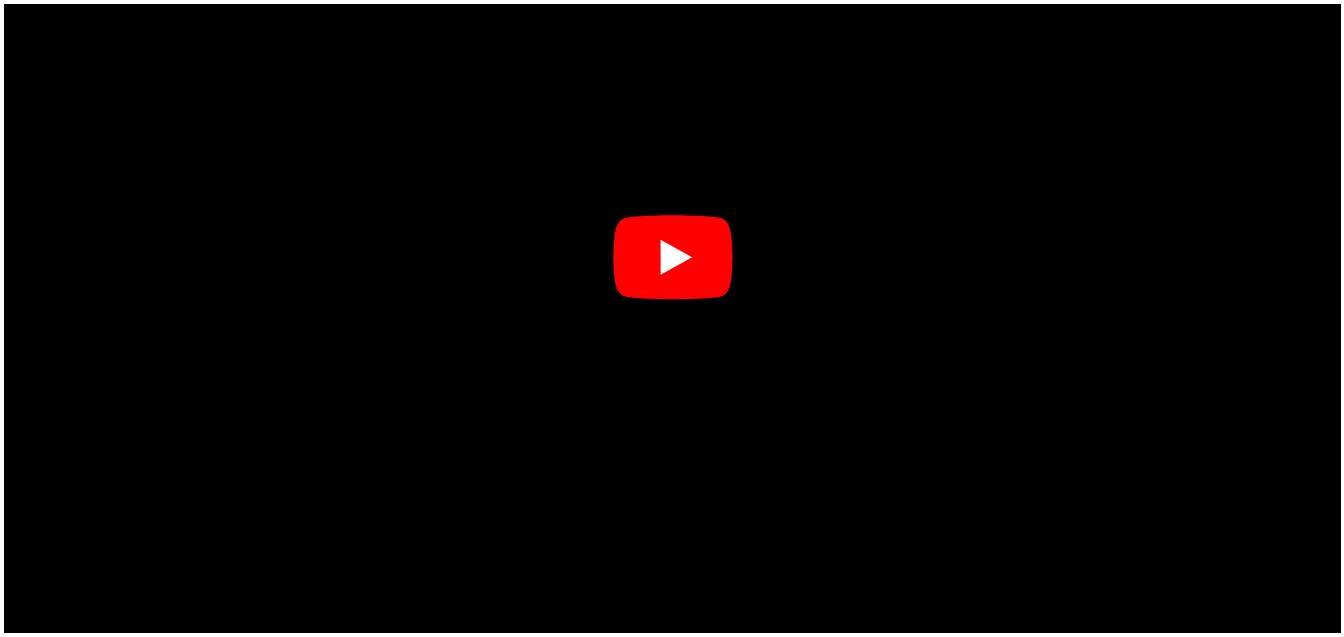
```
{
  "changes": 1, // se ha modificado o borrado un registro
  "lastInsertRowid": 0 // no se ha añadido ningún registro nuevo
}
```

Si en la tabla no hubiese ningún registro con el `id` 2, `query.run()` nos retornaría:

```
{
  "changes": 0, // no se ha modificado ni borrado ningún registro
  "lastInsertRowid": 0 // no se ha añadido ningún registro nuevo
}
```

Lo que significa que no hemos modificado la tabla de la base de datos.

Vamos a ver todo lo que hemos explicado hasta ahora en este vídeo:



> Ejercicio del vídeo

Conclusiones

`DELETE` nos permite añadir un registro a una tabla.

La sintaxis de `DELETE` es:

```
DELETE FROM nombre_de_la_tabla WHERE condicion_que_cumplen_los_registros_a_borrar
```

Para usar SQLite dentro de un servidor de Node JS debemos:

1. Preparar la query con `const query = db.prepare('DELETE FROM users WHERE id = ?');` .
2. Ejecutar la query con `const result = query.run(2);`
   - `query.run()` retorna:

```
{
  "changes": 1, // número de registros borrados
  "lastInsertRowid": 0 // este dato siempre es 0 en consultas de tipo DELETE
}
```

Ejercicios

1. Borrando libros

Partiendo del ejercicio 1 de la lección `SQL INSERT`, añade nuevos endpoints a tu API para borrar libros de la base de datos:

- Crea un endpoint de tipo `DELETE` que borre el libro cuyo `id` llegue por URL params.

- Crea un endpoint de tipo `DELETE` que borre todos los libros que sean físicos y no tengan stock.

Netflix

En los siguientes ejercicios vamos a hacer la actualización del perfil de la usuaria y el registro de nuevas usuarias. Estas dos funcionalidades son independientes. Haz una, la otra o las dos.

---

1. Registro de nuevas usuarias en el front

Si envías el formulario de registro verás que la función `sendSingUpToApi` de `web/src/services/api-user.js` ya está recibiendo por parámetros el email y la contraseña que la usuaria haya introducido en el formulario.

1. Edita el `fetch` de esta función para que envíe por POST los datos al endpoint `/sign-up`. Los datos se deben enviar por body params.
2. Borra los datos falsos del último `then` de este `fetch` para que retorne a React lo que el servidor esté devolviendo. Es decir, pon el código:

   ```
   .then(response => response.json())
   .then(data => {
      return data;
   });
   ```

3. Para ver que todo está bien, comprueba desde DevTools > Network lo que se está enviando al rellenar el formulario.

---

2. Registro de nuevas usuarias en el back

1. Edita el fichero `src/index.js`.
2. Añade un endpoint para gestionar las peticiones `POST:/sign-up`.
3. Recoge los datos que estás recibiendo por body params.
4. Crea una query para insertar en la tabla `users` un nuevo registro con el email y la contraseña de la usuaria.
5. La query te devolverá el id del nuevo registro.
6. Responde a la petición devolviendo los datos:

   ```
   {
     "success": true,
     "userId": "nuevo-id-añadido"
   }
   ```

7. Si tu código es canela en rama, al enviar el formulario de registro desde React, debes ver que la usuaria ha sido logada.

---

3. Comprueba que no haya una usuaria registrada con el mismo email

Para comprobar si ya hay una usuaria registrada con el mismo email debes hacer lo siguiente:

1. En `src/index.js` edita el endpoint `POST:/sign-up`.
2. Recupera el email desde body params.
3. Haz una query de tipo SELECT para obtener las usuarias de la base de datos que tengan el email que estás recibiendo por body params.
4. Si la query te devuelve algún registro es que ya hay una usuaria con ese email. En ese caso responde a la petición con:

```
{
  "success": false,
  "errorMessage": "Usuaria ya existente"
}
```

5. Si la query no devuelve ningún registro es porque no hay ninguna usuaria con ese email. Entonces puedes continuar con el código que has puesto en el ejercicio anterior donde hacíamos la query con el INSERT.
6. Desde el front comprueba que lo estás haciendo bien probando a intentar registrar varias veces el mismo email.

---

4. Actualiza el perfil de la usuaria en el front

Para hacer esta parte de código te recomendamos que vuelvas a leer la lección de **Header params > Identificación de la usuaria**.

Cuando la usuaria modifica su perfil desde el front la función `sendProfileToApi` de `web/src/services/api-user.js` ya recibe el `userId` y los nuevos datos del perfil de la usuaria. Cambia el `fetch` de esta función para:

1. Enviar los datos al endpoint `/user/profile`.
2. Enviar los datos con el verbo POST.
3. Enviar los datos de la usuaria por body params.
4. Enviar el id de la usuaria por header params. Usa `user-id` para el nombre del parámetro de la cabecera.

-

5. Borra los datos falsos que tenemos en el último `then` de este `fetch`.
6. Para comprobar que todo está bien, comprueba DevTools > Network.

---

5. Actualiza el perfil de la usuaria en el back

1. Crea en el back el nuevo endpoint `POST:/user/profile`.
2. Recoge los datos de la usuaria de los body params.
3. Recoge el id de la usuaria de los header params.
4. Dentro del endpoint haz una query a la tabla `users` de tipo UPDATE indicando el id del registro que quieres modificar y los datos que quieres modificar.
5. Responde a la petición con un `{ success: true }`.
6. Para comprobar que todo está bien, envía el formulario desde el front y abre la base de datos con SQLite browser para ver que los datos han cambiado.

---

6. Recupera los datos del perfil de la usuaria desde el front

Queremos que, cuando la usuaria entre en su página de perfil, vea sus datos actuales por si los quiere cambiar. Para ello tenemos que recuperar sus datos de perfil a través de un endpoint.

La función `getProfileFromApi` de `web/src/services/api-user.js` se ejecuta cuando React detecta que la usuaria está logada, es decir, cuando en el estado de React hay un `userId` válido. Esta función ya está recibiendo por parámetros el `userId`.

1. Modifica el `fetch` de esta función para que envíe una petición GET a `/user/profile`.
2. Añade un header param con el `user-id`.
3. Borra el último `then` de este `fetch` porque ahora mismo tiene datos falsos, y sustitúyelo por:

```
.then(response => response.json())
.then(data => {
   return data;
});
```

4. Para comprobar que todo va bien, identifícate en el front y mira DevTools > Network.

---

7. Recupera los datos del perfil de la usuaria desde el back

Como ves, siempre que hacemos un endpoint en el front debemos hacer el mismo en el back. Para ello:

1. Crea un nuevo endpoint en `src/index.js` del tipo `GET:/user/profile`.

2. Recupera el id de la usuaria de los header params.
3. Recupera el nombre, email y contraseña de la tabla `users` con un SELECT, indicando el id de la columna que quieres recuperar.
4. Responde a la petición con el objeto que te ha devuelto la query.
5. Si todo ha ido bien, al entrar en la página del perfil de la usuaria se deberían ver sus datos.

4.7 Bases de datos III

7.1 SQL Relaciones 1 a N

> **Nota:** esta mini lección es importante para entender las siguientes lecciones.

Lo normal es tener **muchas tablas dentro de una base de datos** porque queremos guardar muchos tipos de información.

Por ejemplo, en una tienda virtual vamos a tener una base de datos con **una tabla para cada tipo de información**:

- Una tabla para las usuarias registradas
- Una tabla para los productos
- Una tabla para los pedidos
- Una tabla para las direcciones de entrega de los pedidos

Para cada tipo de información lo creamos en una tabla diferente con sus diferentes campos.

A estos tipos de información les llamamos entidades o entities. Una tabla guarda entidades del mismo tipo, por ejemplo usuarias. Y **cada columna define las características de una entidad**, como el nombre, email o contraseña de la usuaria.

Relaciones entre los registros de dos tablas

Como acabamos de ver, una base de datos suele tener muchas tablas. A menudo nos interesa **relacionar unos registros de una tabla con los registros de otra**.

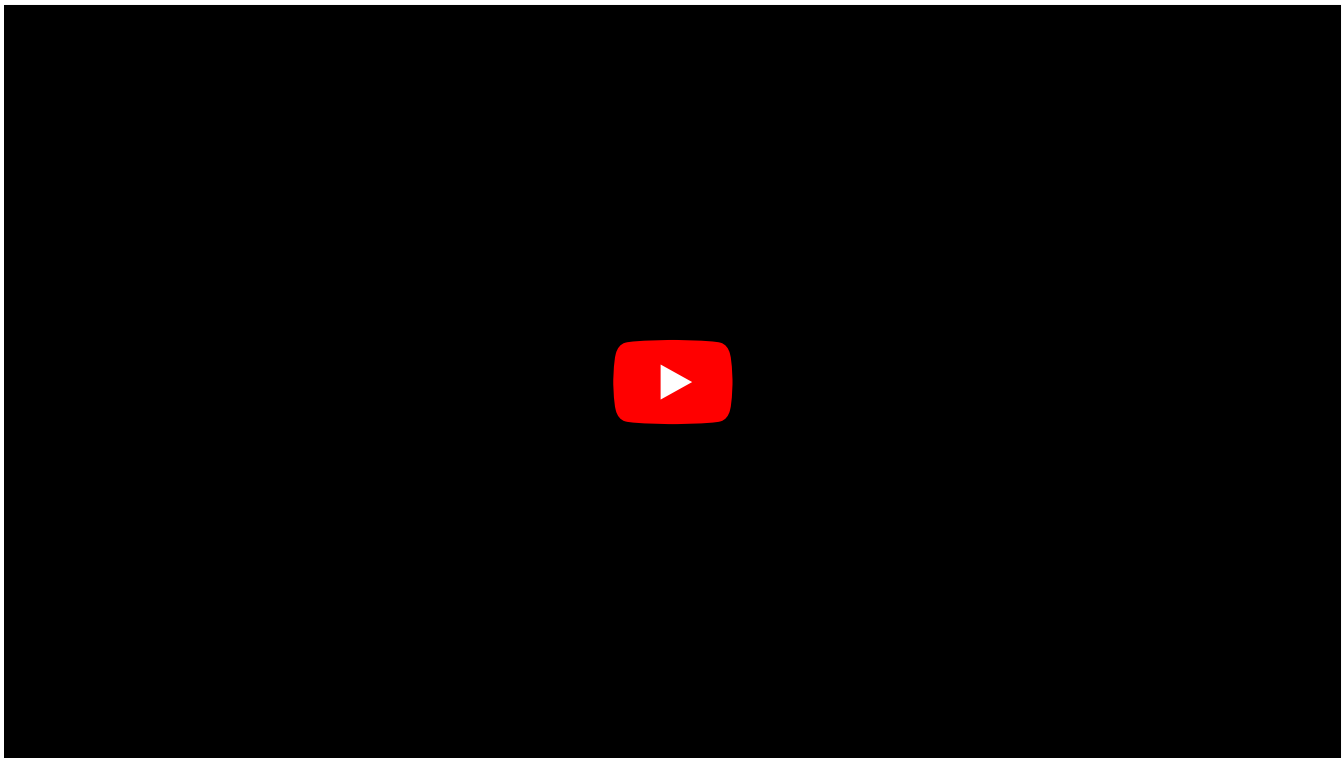
Por ejemplo, en una tienda virtual tenemos una tabla para almacenar las diferentes direcciones de entrega donde enviar los pedidos. Pero necesitamos saber qué dirección de entrega pertenece a una usuaria concreta. **Por ello en la tabla de direcciones de entrega tendremos un campo llamado `userId` que indicará a qué usuaria pertenece cada dirección de entrega.**

Para relacionar los registros vamos a usar los campos `id`, `userId`, etc., ya que son identificadores únicos que nunca van a cambiar.

Por este motivo SQL se define como una base de datos relacional.

Relaciones 1 a N

En siguientes lecciones explicaremos las relaciones 1 a 1 y las relaciones N a N. Ahora vamos a empezar por explicar las relaciones 1 a N que son las más fáciles de entender:



> Ejercicio del vídeo

Las relaciones 1 a N son las que relacionan un registro de una tabla con varios registros de otra tabla

En este caso estamos relacionando **1** usuaria de la tabla `users` con **N** direcciones de la tabla `addresses`.

Ejercicios

1. Categorías de libros

Partiendo del ejercicio 1 de la lección **SQL DELETE** vamos a crear una nueva tabla en nuestra base de datos para añadir categorías de libros. Vamos a **suponer que un libro solo puede pertenecer a una categoría**.

Si 1 libro puede pertenecer a una categoría y una categoría puede tener muchos libros, significa que la relación entre los libros y categorías es de 1 a N.

Para hacer el ejercicio:

1. Crea una nueva tabla en la base de datos llamada `categories`.
    - Esta tabla debe que tener un `id` y un título.
    -

- Añade varias categorías a esta tabla con las categorías: **Historia**, **Novela** y **Ensayo**.

2. ¿Debes crear algo en la tabla `books` para indicar a qué categoría pertenece cada libro?
3. Crea un endpoint de tipo `GET` que devuelva un libro con su categoría y:
   - Reciba por URL params el `id` del libro a seleccionar.
   - Responda con un objeto del tipo:

   ```
   {
     "title": "Don Quijote de la Mancha",
     "author": "Miguel de Cervantes",
     "price": 10,
     "categoryId": 2,
     "categoryName": "Novela"
   }
   ```

   ¿Cuántas queries tienes que hacer a la base de datos para obtener toda la información? ¿Cómo puedes combinar dicha información para devolverla en un solo objeto?

7.2 SQL Relaciones 1 a 1

> **Nota:** esta mini lección es la menos importante de hoy.

SQL es una base de datos relacional. Lo que nos permite **relacionar registros de una tabla con los registros de otra tabla**.

Supongamos que en una tienda virtual tenemos una tabla de usuarias donde guardamos la siguiente información:

- Datos de identificación o acceso de las usuarias:
  - Nombre: texto
  - Email: texto
  - Contraseña: texto
  - Avatar: texto (donde guardamos la URL de la imagen de la usuaria)
  - Fecha en que la usuaria se registró en la tienda virtual: fecha

- Datos relativos a si la usuaria es premium:
  - Usuaria premium: booleano
  - Cuota que ha pagado la usuaria por ser premium: número
  - Fecha en que la usuaria se dio de alta en la versión premium: fecha
  - Fecha en que a la usuaria se le acaba la versión premium: fecha
  - La versión premium se debe autorrenovar: booleano
- Datos relativos al tipo de usuaria por su volumen de compras
  - Usuaria que ha comprado al menos una vez: booleano
  -

    - Usuaria recurrente (que compra al menos una vez al mes): booleano
    - Fecha de la última compra: fecha
- Notificaciones:
    - Usuaria que quiere recibir avisos por las notificaciones del móvil: booleano
    - Usuaria que quiere recibir avisos por email: booleano
- Datos relativos a la newsletter:
    - La usuaria está suscrita a la newsletter de productos: booleano
    - La usuaria está suscrita a la newsletter de novedades: booleano
    - La usuaria está suscrita a la newsletter de nuevas zonas de reparto: booleano
    - Fecha en que la usuaria se dio de alta en la newsletter: fecha
    - Fecha de la última newsletter que le enviamos a la usuaria: fecha
    - Otros tipos de newsletters

Todos estos datos y muchos más podrían estar en la tabla `users` de nuestra base de datos.

Aquí nos surge un problema. Tendríamos la tabla `users` con muchísimas columnas, y por lo tanto tendríamos **muy grande**, lo que genera más problemas a la hora de trabajar con ella.

Además, si una usuaria no es premium las columnas relativas a esta información estarían vacías. Es decir, tendríamos una tabla muy grande con muchísimos campos vacíos.

Ya sabes que cuando tenemos un fichero de código con muchas líneas lo intentamos dividir en ficheros más pequeños para mejorar la legibilidad, reducir errores, separar responsabilidades...

Pues con las tablas vamos a hacer lo mismo: **vamos a dividir una tabla grande en otras más pequeñas**. Normalmente solemos trabajar en una única tabla y cuando vemos que crece mucho tomamos la decisión de dividirla en tablas más pequeñas.

Siguiendo con el ejemplo de la tabla `users` de la tienda virtual de arriba, podríamos dividirla en las siguientes tablas:

- `users`: con datos de identificación o acceso de las usuarias
- `usersPremium`: datos relativos a si la usuaria es premium
- `usersType`: datos relativos al tipo de usuaria por su volumen de compras
- `usersNotifications`: notificaciones
- `usersNewsletters`: datos relativos a la newsletter

Y ahora que tenemos tablas separadas debemos relacionarlas unas con otras. Así que vamos a explicar las relaciones **1 a 1** en este vídeo.

> Ejercicio del vídeo

7.3 SQL Relaciones N a N

> **Nota:** esta mini lección es importante y la más difícil.

Las relaciones N a N relacionan varios registros de una tabla de base de datos con varios registros de otra tabla.

Una relación N a N se produce cuando un registro de la tabla A puede estar relacionado con varios registros de la tabla B, y a su vez un registro de la tabla B puede estar relacionado con varios registros de la tabla A.

Si pensamos en una tienda virtual la relación entre productos y pedidos es la siguiente:

- ¿Un producto puede estar en varios pedidos? Sí, muchas personas pueden comprar el mismo producto. Esto es una relación 1 a N.
- ¿Un pedido puede tener varios productos? Sí, una persona en un solo pedido puede comprar varios productos. Esto es otra relación 1 a N.

Como los pedidos y los productos están relacionados por dos relaciones 1 a N, significa que su relación es N a N.

Vamos a verlo con un ejercicio de verdad:

> Ejercicio del vídeo

Ejercicios

1. Multicategorías de libros

En el ejercicio 1 de la lección **Relaciones 1 a N** hemos creado una tabla para categorías de los libros. Hemos puesto la condición de que un libro solo puede pertenecer a una categoría. Ahora vamos a hacer el mismo ejercicio pero:

- Haciendo que un libro tenga una categoría principal.
- Permitiendo que un libro pertenezca a varias categorías secundarias.

Si un libro puede pertenecer a muchas categorías secundarias y una categoría secundaria puede tener muchos libros significa que ahora la relación entre los libros y las categorías es de N a N.

Partiendo del ejercicio 1 de la lección **Relaciones 1 a N** vamos a empezar por crear la categoría principal:

1. Seguramente en la tabla `books` habías puesto una columna llamada `categoryId` o algo por el estilo. Cambia el nombre de esta columna por `mainCategoryId`.
   - Con este simple cambio estamos diciendo que la relación entre un libro y su categoría principal es 1 a N.

     > Por cierto, al cambiar `categoryId` por `mainCategoryId` en SQLite browser probablemente tendrás que cambiar cosas en el código de tu servidor.

Ahora vamos a crear las multicategorías secundarias:

1. Crea una nueva tabla llamada `secondaryCategories`:
   - Debe tener una columna para indicar el id del libro.
   - Debe tener otra columna para indicar el id de la categoría, el que hemos puesto en la tabla `categories`.
2. Añade datos a mano a través de SQLite a la tabla `secondaryCategories` para que el libro con `id` pertenezca a las categorías secundarias **Novela** e **Historia**.

Ahora vamos a crear un endpoint que devuelva la información de un libro con sus datos, su categoría principal y sus categorías secundarias.

1. Crea un endpoint en tu API para devolver toda la información de un libro. Este endpoint debe:
   - Ser de tipo `GET`
   - Recibir el `id` del libro por URL params
   - Devolver un objeto del tipo:

```
{
  "title": "Don Quijote de la Mancha",
  "author": "Miguel de Cervantes",
  "price": 10,
  "mainCategoryId": 2,
  "mainCategoryName": "Novela",
  "secondaryCategories": [
    {
      "name": "Historia"
    },
    {
      "name": "Novela"
    }
  ]
}
```

7.4 Variables de entorno y Node JS

> **Nota:** esta mini lección es importante, pero solo te será útil cuando quieras publicar tu servidor en Internet. Si quieres no la leas hasta entonces.

En esta lección vamos a explicar qué son las variables de entorno. Para lo cuál antes debemos explicar qué es el entorno.

Qué es el entorno

Según la RAE, entorno se define como:

1. m. Ambiente, lo que rodea.
2. m. Inform. Conjunto de características que definen el lugar y la forma de ejecución de una aplicación.

Es decir, el entorno en programación es el lugar donde se ejecuta un programa. En nuestro caso **el entorno es donde se ejecuta el código de nuestro servidor**.

Siempre vamos a ejecutar nuestro servidor en al menos dos entornos:

- Entorno local o de desarrollo, que es nuestro ordenador.
-

> Entorno de producción, que es el servidor donde vamos a desplegar nuestro código para que las usuarias lo usen a través de Internet.

¿Conocemos algún entorno o servidor de producción?

Sí, GitHub Pages. En este módulo vamos a aprender otro que se llama **Heroku**, porque GitHub Pages no funciona con Node JS.

Cómo desplegar nuestro código en un entorno

Pues como lo hemos hecho hasta ahora. Ya estamos creando ramas en el Git de nuestro proyecto que son ramas "principales", como dev, master o main. Cuando queremos que nuestro código se despliegue en el entorno de producción, lo que hacemos es configurar GitHub Pages para que coja el código de la rama main o master.

¿Existen más entornos?

Sí, los que queramos. En cada proyecto se crean tantos entornos como se quiera (cada uno de ellos asociado a una rama de Git), entre ellos:

- Entorno de prueba, quality assurance (QA), testing, etc., donde se prueban las funcionalidades antes de pasarlas al siguiente entorno.
- Entorno de release, donde se agrupan todas las funcionalidades ya listas, testeadas y preparadas para ser subidas a producción cuando se quiera.
- Entornos de desarrollo especiales donde las programadoras probamos las cosas que no se pueden probar en local.

El equipo entero de desarrollo es el que decide cuántos entornos necesita.

Variables de entorno

Las variables de entorno son unas variables que el entorno, en nuestro caso a través de Node JS, **le pasa a nuestra aplicación web o a nuestro servidor para decirle en dónde se está ejecutando nuestro código**.

Para qué se usan las variables de entorno

A lo mejor te estás preguntando **para qué quiero yo saber en qué entorno se está ejecutando mi código**, si este tiene que funcionar exactamente igual tanto si se ejecuta en mi ordenador como si se ejecuta en el servidor de producción.

Pues bien, tu código **no siempre se va a ejecutar exactamente igual en un entorno que en otro**. Veamos un ejemplo:

Imaginemos que tienes dos proyectos, uno de back y otro de front, que tienen que comunicarse entre ellos. Imaginemos que el proyecto de back lo estás ejecutando en el puerto 3000. Por ello cuando quieres hacer un fetch desde el front al back harás algo como `fetch('http://localhost:3000/ruta-de-mi-endpoint')`.

A continuación imaginemos que subes tu código a un servidor de producción en el dominio `https://misuperweb.com`. Cuando una usuaria entre en esta página, la web hará peticiones fetch a `fetch('http://localhost:3000/ruta-de-mi-endpoint')` pero como `localhost:3000` solo existe dentro de tu ordenador, tu página fallará.

Tú lo que quieres es que cuando tu código esté en el servidor de desarrollo, los fetch sean `fetch('http://localhost:3000/ruta-de-mi-endpoint')` y cuando estés en el entorno de producción, sean `fetch('http://misuperweb.com/ruta-de-mi-endpoint')`

La solución es que Node JS te avise de en qué entorno se está ejecutando tu código y con esa información tú ya puedes hacer los fetch a `localhost:3000` o a `http://misuperweb.com`, por ejemplo con un `if`.

Variables de entorno en Node JS

Node JS tiene una variable global llamada `process` que guarda información sobre el servidor. Dentro tenemos la variable `process.env` que contiene diferente información del **environment** (o entorno). Cada entorno decide qué información meter en `process.env`, por ello tenemos que consultar este objeto para saber en qué entorno estamos.

Variables de entorno en Node JS + React

Cuando ejecutamos React (que se está ejecutando sobre Node JS), nos mete un dato en `process.env.NODE_ENV`.

Prueba lo siguiente en cualquier proyecto de React:

1. Mete la línea `console.log('Entorno:', process.env.NODE_ENV)` en el fichero del componente principal de tu app de React.
    1. Arráncala con `npm start`.
    2. Verás que en consola aparece `Entorno: development`.
2. Crea el código de producción con `npm run build`.
    1. Sube tu código a GitHub Pages y pruébalo.
    2. Verás que en consola aparece `Entorno: production`.

Por ello en nuestro proyecto de React podemos poner algo como:

```
const serverUrl = process.env.NODE_ENV === 'production' ? 'https://misuperweb.com' : 'http://

const getDataFromApi = () => {
  fetch(`${serverUrl}/ruta-de-mi-endpoint`)
   .then(response => response.json())
   .then(...)
};
```

> **Nota:** debes hacer este cambio en tus proyectos front antes de subirlos a un servidor que funcione con Node JS.

Truco

Siempre que quieras saber qué variables de entorno te están pasando, haz lo siguiente:

1. Añade a tu código la línea `console.log('Entorno:', process.env)`.
    1. Ejecútalo en local (entorno de desarrollo) y mira la consola.
2. Súbelo a tu servidor de producción.
    1. Ejecútalo y mira la consola.

Seguro que dentro del objeto `process.env` encontrarás diferencias que te indiquen en qué entorno estás.

Conclusiones

En Node JS las variables de entorno están en `process.env`.

Añade la línea `console.log('Entorno:', process.env)` a tu código y ejecuta tu proyecto en diferentes entornos para conocer qué información guardan las variables de entorno.

Netflix

Ejercicios

Durante todos estos ejercicios hemos trabajado con dos conceptos o entidades que hasta ahora no estaban relacionadas: **las películas y las usuarias**.

Ahora queremos relacionarlas, para saber cuáles son las películas favoritas de las usuarias. Películas y usuarias son datos, por ello las vamos a relacionar en la base de datos.

1. Piensa qué relación es

Lo primero al crear una relación en base de datos es preguntarnos:

- Una usuaria, ¿puede tener 1 película favorita o N (varias)?
- Una película, ¿puede ser una de las favoritas de 1 usuaria o de N?

Si tu respuesta a ambas preguntas es 1, es una relación 1 a 1. Si has respondido a una pregunta con N y a la otra con 1, es una relación 1 a N. Si has respondido a ambas con N, la relación es N a N.

2. Crea una relación N a N

Para crear una relación N a N entre dos tablas de una base de datos siempre vamos a tener que crear una tercera tabla que guarde la información de las relaciones:

1. Crea una nueva tabla en tu base de datos que se llame `rel_movies_users` y que tenga los siguientes campos:
   - `id`: ya sabes que debemos añadir la columna id a todas las tablas.
   - `userId`: que debe ser del mismo tipo que el id de la tabla `users`.
   - `movieId`: que debe ser del mismo tipo que el id de la tabla `movies`.
2. A continuación rellena los datos de esta tabla para que por ejemplo:
   - La usuaria con id 1 tenga como favoritas las películas con id 1 y 2.
   - La usuaria con id 2 tenga como favorita solo la película con id 2.

Si quisieras añadir más información relacionada con esta relación tendrías que añadir más columnas a esta tabla. Por ejemplo, si quieres saber qué puntuación le da cada usuaria a una película deberías añadir una nueva columna llamada `score`. Y así, los registros de esta tabla tendrían la información de que a la usuaria **Y** le gusta la película **Z** y la ha puntuado con **X** puntos sobre 10.

3. Crea el endpoint en el front

Vamos a empezar por recuperar desde el front las películas de una usuaria:

1. La función `getUserMoviesFromApi` de `web/src/services/api-user.js` ya está recibiendo el id de la usuaria logada en la web.
2. Modifica el `fetch` de esta función como hemos hecho ya tantas veces en estos ejercicios:
   - Debe utilizar el verbo GET porque solo vamos a consultar datos, no a modificarlos.
   - Debe tener la ruta `/user/movies`.
   - Debe enviar el `userId` por header params. Usa `user-id` para el nombre del parámetro de la cabecera.
   - Borra el último `then` de este `fetch`, que ahora mismo tiene datos falsos, y sustitúyelo por:

     ```
     .then(response => response.json())
     .then(data => {

        return data;
     });
     ```
3. Para comprobar que todo va bien, identifícate en el front y mira DevTools > Network.

4. Crea el endpoint en el back

Primero vamos a crear el correspondiente endpoint en el back:

1. Crea un nuevo endpoint con la ruta `/user/movies`.
2. Responde a la petición con el objeto:

   ```
   {
     "success": true,
     "movies": []
   }
   ```

3. Si está todo bien, tanto el front como el back funcionarán correctamente, pero en el apartado de **Mis películas aparecerá un listado de películas vacío**.

Por ahora estamos respondiendo con un array vacío, pero el objetivo es que al final de estos ejercicios, este endpoint responda con un array con películas del tipo:

```
{
  "success": true,
  "movies": [
    {
      "id": "2",
      "title": "Friends",
      "gender": "Comedia",
      "image": "https://via.placeholder.com/150" // o la imagen que tú hayas puesto
    }
  ]
}
```

5. Obtén los id de las películas de la usuaria

Dentro del endpoint que acabas de crear:

1. Recupera el id de la usuaria desde los header params.
2. Haz una query a la tabla `rel_movies_users` para obtener **el listado** de todos los `movieId` cuyo `userId` sea el de la usuaria.
   - La query debería ser algo parecido a:

   ```
   const movieIdsQuery = query.prepare(
     'SELECT movieId FROM rel_movies_users WHERE userId = ?'
   );
   ```

   - Para ejecutar la query haríamos:

   ```
   const movieIds = movieIdsQuery.all(req.header('user-id'));
   ```
3. Consolea `movieIds` y si en la terminal aparece algo como `[ { movieId: 1 }, { movieId: 2 }, ... ]` es que lo has hecho bien.

6. Consigue todos los datos de las películas de la usuaria

Ya sabemos los id de las películas favoritas de la usuaria. Ahora necesitamos obtener todos los datos de las películas que tengan esos id.

Vamos a utilizar el operador `IN` de SQL. Este operador nos permite seleccionar registros cuyo id (u otro campo) esté en un array de valores. Para que lo entiendas mejor vamos a ver la query que queremos ejecutar:

```
SELECT * FROM movies WHERE id IN (1, 2); // queremos obtener todas las columnas de la tabla m
```

Para ello Better SQLite 3 nos pide que expresemos la query así:

```
db.prepare('SELECT * FROM movies WHERE id IN (?, ?)');
```

Pero claro, al escribir nuestro código **no sabemos cuántos id nos ha devuelto la query** del ejercicio anterior. Puede que tengamos que hacer la query:

```
db.prepare('SELECT * FROM movies WHERE id IN (?)');
```

o

```
db.prepare('SELECT * FROM movies WHERE id IN (?, ?)');
```

o

```
db.prepare('SELECT * FROM movies WHERE id IN (?, ?, ?)');
```

Por ello hay que crear esta query dinámicamente. Como es un poco complejo os vamos a dar el código de este endpoint:

```
server.get('/user/movies', (req, res) => {
  // preparamos la query para obtener los movieIds
  const movieIdsQuery = db.prepare(
    'SELECT movieId FROM rel_movies_users WHERE userId = ?'
  );
  // obtenemos el id de la usuaria
  const userId = req.header('user-id');
  // ejecutamos la query
  const movieIds = movieIdsQuery.all(userId); // que nos devuelve algo como [{ movieId: 1 }, 

  // obtenemos las interrogaciones separadas por comas
  const moviesIdsQuestions = movieIds.map((id) => '?').join(', '); // que nos devuelve '?, ?'
  // preparamos la segunda query para obtener todos los datos de las películas
  const moviesQuery = db.prepare(
    `SELECT * FROM movies WHERE id IN (${moviesIdsQuestions})`
  );

  // convertimos el array de objetos de id anterior a un array de números
  const moviesIdsNumbers = movieIds.map((movie) => movie.movieId); // que nos devuelve [1.0, 
  // ejecutamos segunda la query
  const movies = moviesQuery.all(moviesIdsNumbers);

  // respondemos a la petición con
  res.json({
    success: true,
    movies: movies,
  });
});
```

> **Nota:** Para entender este código mejor te recomendamos que consolees cada constante que hemos usado.

Para comprobar que tu aplicación funciona, copia y pega este código en tu `src/index.js` y entra en la web > Mis películas. Se deberían mostrar las películas de la usuaria. A continuación edita la tabla `rel_movies_users` para cambiar las películas de una usuaria y vuelve a probar la página.

Conclusiones

Con este ejercicio has practicado todo lo que necesitas saber **para programar un servidor y comunicar una aplicación web y un servidor**. Ya puedes programar cualquier web y servidor de tamaño pequeño y mediano.

Lo que te faltaría por saber es cómo programar webs y servidores de gran tamaño. Hay muchas formas de hacerlo y siempre depende de las características propias del servicio o producto, como si se van a manejar gran cantidad de datos, o si la seguridad es importantísima, o si es necesario que los datos se comuniquen en tiempo real...

Cuando llegues a un proyecto complejo, en tu empresa te contarán cuál es la mejor forma de abordar cada uno de estos desarrollos.

Extras

Ya tenemos terminada nuestra aplicación. Ahora te damos ideas para que sigas trabajando en ella (cuando te sobre tiempo):

Maquetación

Añade tus propios estilos a la página.

Dos contraseñas en el registro

Como ya sabes, los formularios de registro de las páginas suelen tener dos campos de contraseña para que la usuaria la introduzca dos veces y solo si coinciden le permitimos registrarse.

Para hacer esto añade otro input de contraseña al formulario de registro en `web/src/components/SignUp.js` y en la función `handleForm`:

- Comprueba que el valor de las dos contraseñas es el mismo.
  - Si lo es, ejecuta el lifting para enviar el formulario.
  - Si no es, muestra un mensaje de error a la usuaria.

Comprueba la fortaleza de la contraseña

En la función `handleForm` de `web/src/components/SignUp.js`:

-

- Comprueba que la primera contraseña tiene al menos 8 caracteres:
  - Si los tiene, ejecuta el lifting para enviar el formulario.
  - Si no los tiene, muestra un mensaje de error a la usuaria.

(Des)marca como favoritas las películas de la usuaria

Hasta ahora solo podíamos marcar como favorita una película editando a mano la base de datos. Ahora lo vamos a hacer dinámicamente desde la web:

- En React:
  - Añade a las películas un icono de un corazón.
  - Si la usuaria pulsa en el corazón:
    - Envía una petición desde el front al back con los datos:
      - `userId` en el header params.
      - `movieId` en el body params.
- En Node JS:
  - Crea el correspondiente endpoint. En él:
    - Haz un `SELECT` para saber si en `rel_movies_users` hay algún registro que tenga el `movieId` y el `userId`.
    - Si la query devuelve un registro, es que es favorita; entonces haz un `DELETE` para borrar todos los registros de `rel_movies_users` que tengan el `movieId` y el `userId`.
    - Si la query no devuelve nada, es que no es favorita; entonces haz un `INSERT` para añadir un registro con el `movieId` y el `userId`.

Para comprobar que todo está bien, si la usuaria marca una película como favorita deberá aparecer en la web > Mis películas al refrescar la página.

Indicadnos qué ejercicios habéis hecho (friendly reminder)

Recordad que para hacer un mejor seguimiento de vuestra evolución durante este módulo, os pedimos que en el README de vuestro repo marquéis con una X los ejercicios hayáis conseguido terminar.

Repaso final

Ejercicio de repaso

En este ejercicio vamos a desarrollar un servidor web. Durante el módulo 2 durante el pair programing hemos desarrollado la aplicación Adakitten que usaba un servidor creado por las profesoras de Adalab, que para nosotras es una caja negra. **Hasta ahora no sabíamos cómo funcionaba por dentro.**

Ahora queremos crear un servidor que tenga la misma funcionalidad que el servidor creado por las profesoras de Adalab. **Al finalizar el desarrollo podremos utilizar la página AdaKitten del módulo 2 con**

el servidor realizado en este módulo

Endpoints a desarrollar en el servidor AdaKittens

En todos los endpoint desarrollados el servidor debe comprobar que los datos recibidos desde el navegador son correctos.

- En caso de que los datos **no** sean correctos, el servidor debe devolver una respuesta de error.
- En caso de que los datos **sí** sean correctos, el servidor debe hacer la operación necesaria para cada caso.

Listado de endpoints

1. Endpoint `/api/kittens/:user` para obtener el listado de gatitos donde `user` es un parámetro que especifica el usuario y solo devolver los gatitos de ese usuario. La respuesta del servidor debe ser un `json` con la siguiente información:

```
{
    info: {
        count: numOfkittens, //número de elementos del listado
    },
    results: kittenData //listado de gatitos
}
```

2. Endpoint `/api/kittens/:user` para insertar un nuevo gatito donde `user` es un parámetro que especifica que usuario está insertando el gatito. Este endpoint también espera por el `body` de la petición toda la información del gatito a insertar . La respuesta del servidor debe ser un `json` especificando si la solicitud fue exitosa o no.

```
{
    success: false //Puede ser true o false
}
```

3. Endpoint `/api/kittens/:user/:kitten_id` para actualizar un gatito ara donde `user` es un parámetro que especifica que usuario está actualizando el gatito y `kitten_id` es el identificador del gatito que se quiere actualizar. Este endpoint también espera por el `body` de la petición toda la información del gatito. La respuesta del servidor debe ser un `json` especificando si la solicitud fue exitosa o no.

```
{
    success: false //Puede ser true o false
}
```

4. Endpoint `/api/kittens/:user/:kitten_id` para eliminar un gatito donde `user` es un parámetro que especifica que usuario está eliminado el gatito y `kitten_id` es el identificador del gatito que se quiere eliminar. La respuesta del servidor debe ser un `json` especificando si la solicitud fue exitosa o no.

```
{
    success: false //Puede ser true o false
}
```

Bases de datos AdaKittens

Os proporcionamos la estructura de datos para guardar la información de los gatito. Tenemos una tabla con la siguiente información:

- **id**: número entero que representa el identificador del gatito, es una valor único, auto incremental y es la clave principal de la tabla.
- **owner**: cadena de texto que representa el usuario asociado a ese gatito.
- **url**: cadena de texto que representa la dirección de la foto del gatito.
- **name**: cadena de texto con el nombre del gatito.
- **race**: cadena de texto con la raza del gatito.
- **desc**: cadena de texto con la descripción del gatito.

¡Al turrón!