

Programación en JavaScript para programadores

Autor: David Lampon Diestre

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Joan Soler

Adillon

PID_00269223

1. Tooling

- 1.1. Presentación
- 1.2. Node.js y npm
- 1.3. Instalación de node.js y npm
- 1.4. Creando la base de la aplicación
- 1.5. Instalando una dependencia
- 1.6. ¿Qué es `parcel`?
- 1.7. Nuestro primer *script*
- 1.8. Primeros pasos con nuestra aplicación
- 1.9. Preparando el entorno de *testing*
- 1.10. Jest y tipos de test
- 1.11. Primeros pasos con TDD
- 1.12. Módulos ES6
- 1.13. Estilo de código y autocorrección
- 1.14. Git (opcional)
- 1.15. Resumen
- 1.16. Bibliografía

2. Introducción a JavaScript

- 2.1. Planteamiento
- 2.2. Introducción
- 2.3. Historia y contextualización
- 2.4. Las características fundacionales de JavaScript
- 2.5. Evolución
- 2.6. Situación actual
- 2.7. Buceando en JavaScript
 - 2.7.1. Introducción
 - 2.7.2. Módulos
- 2.8. Conceptos básicos
 - 2.8.1. Lenguaje no tipado
 - 2.8.2. Objetos
 - 2.8.3. Clases
 - 2.8.4. Scope

2.8.5. This

2.8.6. Closure

2.8.7. Prototype

3. Gestión de la asincronía

3.1. Planteamiento

3.2. Introducción

3.3. HTTP

3.4. Del monolito web al *front-end* y *back-end* y SPA

3.5. Interfaz y API

3.5.1. Introducción

3.5.2. XMLHttpRequest

3.5.3. jQuery y Ajax

3.5.4. Fetch

3.6. Control de flujo en el código

3.6.1. Introducción

3.6.2. *Callbacks*

3.6.3. *Promises*

3.6.4. Async/await

3.7. Notas sobre asincronía en JavaScript

3.8. Bibliografía

4. Manipulación del DOM

4.1. Planteamiento

4.2. Introducción

4.3. Consultar el DOM: peticiones

4.4. Element, Node y NodeList

4.5. Modificar el DOM: acciones

4.6. Añadir «escuchadores» de eventos

4.7. Bibliografía

5. Diseño de API en Node

5.1. Presentación

5.2. Introducción

5.2.1. Qué es una API

5.2.2. Qué es Node

5.2.3. Qué es Express

5.2.4. Qué es MongoDB

5.3. Metodología del ejercicio

5.3.1. Introducción

5.3.2. Requisitos

5.3.3. Dependencias del proyecto

5.3.4. Comandos

5.3.5. Elementos adicionales

5.4. Express

5.4.1. Introducción

5.4.2. Routing

5.5. Modelo de datos con MongoDB, schemas

5.6. Controladores

5.7. Autorización

5.8. Bibliografía

1. Tooling

1.1. Presentación

Tooling es el término anglosajón usado para describir un **conjunto de herramientas** de forma genérica.

Cuando hablamos del *tooling* dentro del ecosistema de **JavaScript** nos referimos a todas aquellas utilidades auxiliares que nos servirán de apoyo y ayuda durante el desarrollo de nuestros proyectos.

Gran parte de las acciones que llevamos a cabo durante la fase de desarrollo son mecánicas y repetitivas. Tener un entorno o *tooling* que automatice estos procesos nos permitirá minimizar el tiempo que les dedicamos y centrarnos de manera más eficiente y efectiva en el desarrollo de nuestras aplicaciones.

En la historia moderna de la automatización de los entornos de desarrollo *front-end* se han quemado fases con protagonistas muy concretos, cronológicamente:

- [Grunt](#)
- [Gulp](#)
- [Webpack](#)

Las diferencias entre ellos se deben a concepciones muy diferentes de cómo atacar la automatización de procesos, y deben entenderse no como competidores sino como evoluciones. Es cierto que todos ellos comparten aún el mismo ecosistema, pero ello se debe a la adopción de cada uno de ellos de forma cronológica. Actualmente todo nuevo proyecto debe considerar **Webpack** como la opción por defecto.

En nuestro caso vamos a utilizar [Parceljs](#), una herramienta muy similar a **Webpack** que proporciona una serie de herramientas que requieren mucha menos configuración que este pero que ofrecen la misma flexibilidad y potencia.

A continuación vamos a realizar una serie de pasos de forma guiada para implementar el *tooling* de una aplicación moderna en **JavaScript**, que será nuestro entorno de trabajo para el resto del curso, tanto para los ejercicios de la teoría como para las prácticas.

Sin embargo, antes de ponernos a automatizar procesos, analizaremos cada uno de los desafíos que se nos presentan, las tecnologías que entran en juego y los problemas que quedan resueltos por cada una de ellas.

1. Tooling

1.2. Node.js y npm

Históricamente, JavaScript había sido un lenguaje ejecutado en el navegador. Los navegadores son aplicaciones de escritorio que albergan en su interior un motor (*engine*) que compila y ejecuta Javascript.

Sin embargo, Ryan Dahl cambió este paradigma el 8 de noviembre de 2009 con su trabajo y presentación de node.js en la JSConf.

Node.js es **un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome**, dicho de modo muy simple, una emulación de lo que sucede en el navegador sin el renderizado visual (no existe el objeto *window*).

Este hecho fue disruptivo, ya que permitió que JavaScript, uno de los lenguajes de programación más populares y con una curva de entrada más baja, pasara a ser, simultáneamente, un lenguaje de cliente y de servidor, lo que permitía que muchos ingenieros de *front-end* pudieran dar el salto a *back-end* sin tener que cambiar de lenguaje.

Node.js permitió el desarrollo de muchas aplicaciones no destinadas directamente a los navegadores y facilitó la aparición de un ecosistema de utilidades a las que se les llama paquetes (*packages*).

Al instalar **node.js** en nuestro ordenador se instala también un gestor de paquetes: el **npm**, **node package manager** (gestor de paquetes de node). Esta utilidad nos permite (entre otras cosas):

- Crear nuestros propios paquetes (nuestras aplicaciones).
- Instalar y usar paquetes desarrollados por terceros en nuestra aplicación.

Para poder tener una idea del tipo de aplicaciones a las que nos referimos podemos pasarnos por npmjs.com, el repositorio de paquetes oficial de **node.js**, una especie de supermercado de utilidades. Podemos, por ejemplo, instalar React, la famosa librería de **Facebook**, mediante las instrucciones presentadas en esta página.

1. Tooling

1.3. Instalación de node.js y npm

Ahora que sabemos qué es y para qué sirven **node.js** y **npm**, podemos dirigirnos a su [página oficial](#) para descargarnos el binario más adecuado para nuestra plataforma de desarrollo.

Se nos presentan dos opciones: última versión o LTS (*long term support*); esta última es la opción más segura y aconsejable en nuestro caso, ya que no tenemos necesidad de hacer uso de las últimas novedades ni versiones experimentales con poco soporte.

Para comprobar que la instalación ha sido correcta, debemos abrir un terminal y ejecutar los siguientes comandos:

```
node -v
```

```
npm -v
```

Obtendremos como respuesta de cada uno de estos comandos la versión de node y de npm respectivamente.

Nota

El porcentaje de usuarios que utilizan un entorno Linux/Mac en el desarrollo de aplicaciones JavaScript supera de manera notable a los que usan entornos Windows. De aquí en adelante los comandos presentados serán todos en formato Linux/Mac. Los conceptos serán los mismos e independientes de la plataforma, pero los usuarios de Windows quizás deban investigar en el caso de que los comandos no sean directamente ejecutables en la consola.

La utilidad **npm** es lo que se conoce como una **interfaz de línea de comandos** (CLI) y es lo que nos permite importar código de terceros desde el [registro de npm](#). Esto implica que el uso de la herramienta debe hacerse mediante la consola de nuestro sistema operativo. A continuación podéis encontrar unos enlaces en los que podéis familiarizáros con estas aplicaciones que serán los entornos en los que trabajaremos esta práctica:

- [Mac](#)
- [Linux](#)
- Windows (dos opciones): [cmd / terminal](#)

Nota

Los usuarios de [Visual Studio Code](#) (nuestro editor recomendado) pueden obviar este paso, ya que lleva incluida una herramienta de consola. Si queréis ahorrar el paso anterior, no dudéis en instalarlos desde [aquí](#) este editor y abrir la consola de [este modo](#).

1. Tooling

1.4. Creando la base de la aplicación

Nuestros ejercicios, en el caso de esta asignatura hablamos de las PEC, no son necesariamente aplicaciones de escritorio. Sin embargo, el *tooling* alrededor de nuestra aplicación sí lo será. Queremos poder hacer uso de algunas de las funcionalidades que podemos encontrar en la librería de paquetes de npm para nuestro proyecto y, como dijimos, automatizar y optimizar nuestros procesos en fase de desarrollo.

Para arrancar la creación de nuestro proyecto, debemos buscar una carpeta en nuestro disco duro que nos parezca apropiado. En mi caso personal suele encontrarse en la ruta `\\Projects` en la raíz del disco duro (cuyo formato será diferente para los usuarios de Windows y Linux/Mac).

Una vez tengamos ubicada nuestra carpeta de proyectos, crearemos una nueva carpeta destinada a albergar nuestra aplicación y navegaremos hacia ella, por ejemplo:

```
mkdir javascript_para_programadores && cd javascript_para_programadores
```

Una vez en su interior, llamamos al inicializador de paquetes de node.js:

```
npm init
```

Podemos pasar por todas las preguntas que nos solicita el prompt con `enter` para aceptar los valores por defecto o directamente usar el comando con un parámetro adicional para obviarlas:

```
npm init -y
```

Esto que puede parecer tremendamente sofisticado únicamente genera un fichero llamado `package.json` en la raíz de nuestro proyecto con los campos y valores que hemos determinado.

El formato JSON (JavaScript Object Notation) es un estándar de la industria, no exclusivo de JavaScript, para representar objetos. Los objetos son pares de *key/value* (clave/valor) envueltos entre los caracteres `{}`.

Si echamos un vistazo a nuestro `package.json` veremos algo parecido a esto:

```
{
  "name": "my-tooling",
  "version": "1.0.0",
  "description": "My super awesome tooling",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "\"David <dlampon@uoc.edu>\"",
  "license": "ISC"
}
```

Esta es la firma de nuestro paquete. Todos los campos son autodescriptivos excepto dos, que pueden requerir algo más de explicación:

- `main`: es el punto de entrada de nuestra aplicación, normalmente `index.html` para una aplicación de navegador e `index.js` para una SPA y aplicaciones de escritorio o servidor.
- `scripts`: es un objeto en el que podemos definir todos los comandos asociados a nuestro proyecto y que podrán ser ejecutados desde la raíz usando el comando `npm run <nombre_del_script>`

Nota

Todo este proceso lanzado mediante `npm init` podría haberse hecho de forma manual creando el fichero con el comando `touch package.json` y definido en su interior el objeto con las propiedades de nuestro proyecto.

Aprovecharemos también para cambiar el punto de entrada de nuestra aplicación:

```
{  
  "main": "src/index.html",  
}
```

1. Tooling

1.5. Instalando una dependencia

Aunque hasta ahora solo tengamos una carpeta con un fichero de definición del paquete, lo que hemos habilitado, entre otras muchas cosas, es la posibilidad de instalar dependencias.

Una dependencia es un paquete externo alojado en un [registro](#) y desarrollado por terceros, de nuestro propio repositorio interno (como hacen muchas empresas) o bien local, de otro paquete/aplicación que hayamos desarrollado y almacenado en nuestro disco duro.

El primer paquete que vamos a instalar es [Parcel.js](#) usando el comando:

```
npm install --save-dev parcel-bundler
```

O su versión abreviada:

```
npm i -D parcel-bundler
```

Esto dará paso a la instalación del paquete. Veremos una barra de proceso, en la que se descargará tanto `parcel` como sus propias dependencias, que quedarán instaladas en la carpeta `node_modules`. Aquí es donde residirán todas las dependencias de nuestro proyecto (y las subdependencias de nuestras dependencias).

Todos los ficheros deben estar en local para poder hacer uso de estas utilidades de [node](#). Sin embargo, si trabajamos de forma conjunta con otros miembros del equipo o en proyectos *open source*, podría ser bastante tedioso tener que compartir esta carpeta, ya que suele ser bastante pesada.

Lo brillante de este sistema de dependencias es que solo necesitamos indicar en nuestro fichero de firma del proyecto las dependencias y su versión para que todos los que trabajen en el mismo proyecto tengan el mismo estado de la aplicación.

Es decir, en caso de trabajar en un proyecto con otros desarrolladores, al entrar en una empresa nueva o al compartir nuestro trabajo con el mundo, nuestro proyecto queda definido en el fichero `package.json` y todo el mundo tendrá las mismas versiones y dependencias al ejecutar el comando:

```
npm install
```

O su versión reducida:

```
npm i
```

Haced la prueba:

1. Buscad y borrad la carpeta `node_modules` en la raíz de vuestro proyecto con el comando:

```
rm -rf ./node_modules
```

2. Ahora estáis en el mismo estado que alguien al que le paséis vuestro `package.json` por primera vez. Ahora ejecutad:

```
npm i
```

Los mismos ficheros, dependencias y subdependencias de estas se han vuelto a instalar y están en el mismo estado que antes de borrar la carpeta `node_modules`.

Nota

Se espera que en las entregas de las PEC **no se incluya la carpeta node_modules** puesto que la carga y descarga de los ejercicios aumenta de manera innecesaria.

Es un sistema muy potente que facilita el trabajo en equipo, la sincronización de los cambios del código y las versiones de los paquetes, muy importante para el trabajo con gestores de versiones tipo [Git](#).

Nota

A pesar de que Git sería el sistema de trabajo deseable, se escapa del temario comprendido en la asignatura puesto que supone una capa extra de complejidad. Todo aquel que quiera trabajar contra un repositorio Git y entregar así los ejercicios es totalmente libre de hacerlo siempre que el repositorio sea privado y se dé acceso al profesor para corregir la práctica. Este punto es totalmente opcional y no afectará a la puntuación final de los ejercicios, puesto que solo modifica el formato de entrega.

Si revisamos nuestro `package.json` , veremos que ha aparecido un nuevo par de valores:

```
"devDependencies": {  
  "parcel-bundler": "^1.12.3"  
}
```

Nota

Las versiones de los paquetes pueden ser diferentes a las recogidas en este documento debido a actualizaciones de estas posteriores a la redacción del material.

Hay dos tipos de dependencias: `dependencies` y `devDependencies` . Las primeras son las que necesitaremos tanto en desarrollo como en producción, como podría ser alguna librería tipo `react` o `d3` . Las segundas son las dependencias de desarrollo, como puede ser la librería de testing `jest` . No necesitaremos testing en nuestra versión de producción expuesta al público, pero sí lo necesitaremos en fase de desarrollo para poder testear nuestra aplicación.

1. Tooling

1.6. ¿Qué es `parcel`?

Parcel es un `bundler` para proyectos de **JavaScript** relativamente nuevo y cuyo potencial viene dado por ofrecer altas prestaciones con casi nula configuración (a diferencia de **Webpack**). Un `bundler` es una herramienta que, mediante un punto de entrada en el proyecto, rastrea todas las dependencias y entrega un único fichero final con todo nuestro código en su interior.

Lo que nos ofrece **Parcel** (entre otras cosas) es:

- *Live server* para nuestras aplicaciones
- *Hot reload* de nuestros cambios
- *Bundling* rápido de JS
- Poder usar módulos de ES6 (*import/export*)

Todo esto puede no significar mucho ahora mismo, pero volveremos a estos puntos más adelante según empecemos a desarrollar y aclararemos el significado de cada uno de ellos.

1. Tooling

1.7. Nuestro primer script

Los *scripts* son comandos de Bash propios de cada proyecto que podemos definir para abstraer partes de la lógica asociada a los procesos de desarrollo. Vamos a crear un *script* para lanzar el *live server* para nuestra aplicación.

Vamos a añadir esta línea en nuestro fichero `package.json` :

```
"scripts": {  
  "dev": "parcel src/index.html --open",  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

Nota

El *flag* “--open” hace que el navegador se abra automáticamente cuando el servidor está listo. Eliminad este *flag* si queréis el comportamiento por defecto.

Para poder ejecutar el comando `dev` , debemos hacerlo de este modo desde la raíz del proyecto:

```
npm run dev
```

Nota

Si lo ejecutamos, veremos un error por pantalla puesto que no encuentra `index.html` .

Los beneficios inmediatos de esto son:

- No necesitamos que Parcel sea una dependencia global del sistema (ved la nota más abajo).
- Hemos abstraído el punto de entrada de la aplicación a un *script* del proyecto.
- Hemos generalizado la carga del proyecto a un comando `dev` de npm que es un estándar.

El uso de *scripts* es mucho más valioso cuando los comandos tienen muchas opciones o *flags* adicionales. Este es, por tanto, un ejemplo sencillo del uso que puede darse a los *scripts*.

Nota

De no ser así, los usuarios que accedieran al proyecto y no tuvieran Parcel como dependencia de sistema recibirían un error al ejecutar el *script*. Es mejor que todas las dependencias de un proyecto se incluyan en él.

1. Tooling

1.8. Primeros pasos con nuestra aplicación

Todo lo que hemos hecho ahora ha sido definir nuestro `tooling` y la base de nuestra aplicación. Vamos a añadir tres ficheros de html, css y js para entender cómo nos va a ayudar `parcel` en fase de desarrollo.

Nota

Ha llegado el momento de empezar a escribir código y toca decidir qué editor usar. Mi consejo y recomendación es usar [Visual Studio Code](#). Es mi editor preferido y el que uso a nivel profesional. Cualquier otra opción que os resulte cómoda y útil, y que os permita sacar vuestro trabajo adelante, es igualmente válida. No debemos olvidar que, a fin de cuentas, se trata solo de una herramienta más y lo único importante es que nos permita ser productivos.

Creamos una nueva carpeta `src` en la raíz de nuestro proyecto:

```
mkdir src && cd src
```

Si queréis, podéis aprovechar la ocasión para crear las subcarpetas de las actividades del curso para definir la estructura del proyecto:

```
mkdir pec1 && mkdir pec2 && mkdir pec3
```

Abrimos nuestro editor y creamos en la ruta recién creada los siguientes ficheros.

Creamos el fichero `index.html` con este contenido:

```
<html>
  <head>
    <title>JS para programadores</title>
    <link rel="stylesheet" type="text/css" href="styles.css" />
  </head>
  <body>
    <div id="main"></div>
    <script src=".index.js"></script>
  </body>
</html>
```

Creamos el fichero `styles.css` con este contenido:

```
#main {
  width: 100%;
  height: 100%;
  background-color: orange;
}
```

Creamos el fichero `index.js` con este contenido:

```
console.log('Bienvenido a JS para programadores');
```

Estos tres archivos son una de las representaciones más básicas de una aplicación de navegador con la estructura en html, los estilos en css y la interactividad proporcionada por JavaScript.

Volveremos a nuestro terminal y ejecutaremos:

```
npm run dev
```

Con esto le estamos diciendo a `parcel` que cargue el fichero `index.html` , recupere todas sus dependencias, cree una carpeta de distribución y la sirva en un servidor local en el puerto `1234` . Veremos en el terminal el mensaje:

```
Server running at http://localhost:1234
✖ Built in 14ms.
```

Si accedemos a esa dirección en el navegador, podremos ver el contenido de nuestra página.

Nota

Para poder ver el mensaje del `console.log` en nuestro fichero de JavaScript, debemos abrir las *developer tools* de nuestro navegador [Chrome](#), [Firefox](#), [Edge](#) y acceder a la pestaña de `console` .

Han sucedido procesos bastante sofisticados que han quedado escondidos a nuestros ojos para que podamos ver el resultado de la creación de nuestros archivos renderizados en el navegador.

Sin embargo, lo que puede resultarnos más útil para la realización de las prácticas es el hecho de que mientras no cerramos la consola en la que hemos ejecutado `parcel` , podremos modificar cualquiera de los ficheros y los cambios se recargarán automáticamente en el navegador.

Probad a añadir algún texto en el `index.html` , modificar alguno de los estilos o cambiar el texto del `console.log` y veréis que el navegador se autorrefresca y recarga los cambios.

Nota

Es recomendable tener una distribución de pantalla partida en la que una mitad la ocupe el navegador con `localhost:1234` y la otra mitad vuestro editor para trabajar y ver los resultados de los cambios en la misma pantalla.

1. Tooling

1.9. Preparando el entorno de testing

Para poder implementar *testing* en nuestra aplicación y poder, en mayor o menor medida, trabajar con metodologías TDD, vamos a instalar la dependencia Jest, la suite de *testings* desarrollada por los ingenieros de **Facebook**.

En este punto tenemos dos opciones:

- Abrir y usar una nueva consola para instalar los nuevos paquetes.
- Terminar la ejecución, instalar y volver a arrancar el *script* de desarrollo.

Elegid una de las dos opciones e introducid este comando en el terminal:

```
npm i -D jest
```

Veremos que nuestro fichero `package.json` contiene una nueva dependencia de desarrollo:

```
"devDependencies": {  
  "jest": "^24.9.0",  
  "parcel": "^1.12.3"  
}
```

Si ejecutamos nuestro script de `test` que venía por defecto al instalar la aplicación mediante:

```
npm run test
```

Obtendremos un mensaje de error puesto que es lo que queda especificado en el fichero con `echo \\"\$Error: no test specified\\\" && exit 1`. Modificaremos su valor para llamar al binario de la librería de *testing*:

```
"scripts": {  
  "dev": "parcel src/index.html",  
  "test": "jest --passWithNoTests --silent"  
}
```

Nota

El *flag* `--passWithNoTests` evita que se muestre un mensaje de error por consola en caso de que no haya ningún test implementado. El *flag* `--silent` evita que se muestren errores adicionales no relacionados con los test fallidos.

Ejecutamos este comando para ver la librería de *testing* en funcionamiento:

```
npm run test
```

1. Tooling

1.10. Jest y tipos de test

Jest es una librería de **testing unitario**, que son los que nos permiten someter una función o una parte de nuestro código a unas entradas establecidas por nosotros y valorar su resultado de modo que procesen los datos como nosotros esperamos. Esto nos da consistencia en el código desde que creamos la función (unidad) y durante todo el proceso de desarrollo mientras aumentamos la base de código sin tener que manualmente comprobar que el comportamiento se mantiene.

El siguiente tipo son los **test de integración**, que nos permiten valorar cómo se comporta un servicio contra otro. Esto podría ser también a nivel de microservicios en el servidor o bien partes diferentes de la aplicación. Se puede hacer con Jest pero valorando que trabajaremos no de manera exclusiva con nuestra función desarrollada, sino contra otras partes del código o servicios de terceros.

Por último, tenemos los **E2E (end to end testing)**, que son los que tradicionalmente han realizado los ingenieros de QA (*quality assurance*) de forma manual. En cada nueva iteración del código debían realizar de modo manual una serie de pasos y procesos desde el navegador como si fueran usuarios pasando por los *test cases* definidos para someter a prueba la aplicación. Este proceso manual puede automatizarse con herramientas como [Selenium](#) y [Cypress](#). Estas librerías y frameworks han dado paso a una nueva categoría profesional que son ingenieros de *automation QA*.

Este último tipo de test son los que deberíamos implementar para poder valorar de forma automatizada el comportamiento de nuestra aplicación. Sin embargo, puesto que su implementación requiere conocer el uso de esos frameworks, la carga adicional para esta asignatura sería excesiva.

1. Tooling

1.11. Primeros pasos con TDD

Test Driven Development es una práctica de desarrollo en la que se definen en forma de test qué funcionalidades debe pasar un programa (o una parte de este) para ser válido, y posteriormente se implementa el código que cumple ese requisito.

Vamos a ver un ejemplo sencillo. Debemos crear un fichero `sum.js` y otro `sum.spec.js` .

Nota

Es la parte `spec` del nombre la que define que se trata de un archivo de test. **Jest** recogerá todos y cada uno de los ficheros `spec`, pasará los test uno por uno y mostrará el resumen por consola.

En el fichero `sum` vamos a añadir la siguiente función:

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

module.exports = sum;
```

En el fichero `spec` vamos a añadir el siguiente test:

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(4);
});
```

Nota

Estas dos sintaxis son equivalentes:

- `function() {}`
- `() => {}`

La segunda se conoce como función flecha (*arrow function*) y es una novedad introducida en ES2015.

Si ahora ejecutamos el comando de test, veremos que se salta un error. Según la filosofía TDD, para que un test sea válido primero hemos de asegurarnos de que falla en un caso incorrecto para luego corregirlo, ver que pasa con éxito y seguir iterando. Modificamos el contenido del test por su valor correcto:

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
```

```
    expect(sum(1, 2)).toBe(3);  
});
```

Ahora, al ejecutar el comando de `test`, veremos que los resultados son correctos.

1. Tooling

1.12. Módulos ES6

Hay apartados específicos en la teoría de la asignatura donde entraremos con más detalle en la teoría de módulos y ES6. Presentarla aquí alargaría innecesariamente este apartado introductorio.

Sin embargo y resumiendo mucho, podemos asimilar que un módulo es cada uno de los ficheros de JavaScript de nuestro proyecto y ES6 (EcmaScript), la última revisión revisada y lanzada de JavaScript.

Lo que pretendemos en este punto es habilitar la posibilidad de usar módulos y que en los ficheros de test podamos usar la nomenclatura de módulos `ES6` (`import/export`) en lugar de los `ES5/CommonJS` (`module.exports/require`).

`Parcel` nos permite usarlos a nivel de nuestra aplicación pero no para los test, ya que no son parte directa de esta (los test no van en un *bundle* de producción); Jest es quien trabaja los ficheros de test y por ello debemos instalar un nuevo paquete para poder habilitar esta funcionalidad para nuestros test.

```
npm i -D babel-jest @babel/core @babel/preset-env
```

Nuestro fichero `package.json` tendrá nuevas entradas en su apartado de dependencias de desarrollo:

```
"devDependencies": {  
  "@babel/core": "^7.5.5",  
  "@babel/preset-env": "^7.5.5",  
  "babel-jest": "^24.9.0",  
  "jest": "^24.9.0",  
  "parcel-bundler": "^1.12.3"  
}
```

Babel es un transpilador entre versiones de ECMAScript (la especificación de JavaScript). Nos permite escribir ES6 y automáticamente hará una conversión a ES5 para que node.js pueda ejecutarlo como si fuera el mismo código (node.js no soporta aún de forma nativa los módulos ES6).

Nota

Enlaces a los paquetes instalados [babel-jest](#), [@babel/core](#), [@babel/preset-env](#).

Con estos paquetes ya tenemos instaladas las funcionalidades para poder usar la sintaxis de módulos en nuestros test de aplicación. Sin embargo, necesitamos indicar mediante un archivo adicional de configuración cómo vamos a hacer esa conversión. Debemos crear un nuevo fichero `babel.config.js` en la raíz del proyecto (no en `/src`) con este contenido:

```
module.exports = {  
  presets: [  
    [  
      '@babel/preset-env',  
      {  
        targets: {  
          node: 'current'  
        }  
      }  
    ]  
  ]  
}
```

```
    ]
  ];
};
```

Ahora cambiaremos el formato de nuestro fichero `sum.js` por:

```
export default function sum(a, b) {
  return a + b;
}
```

Y el de `sum.spec.js` por:

```
import sum from './sum';

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Procederemos a ejecutar de nuevo los test y veremos cómo, a pesar del cambio de sintaxis, no se nos presenta ningún de error de ejecución, ya que hemos habilitado el uso de las nuevas funcionalidades en nuestros test como lo hace `parcel` en nuestros ficheros de aplicación.

1. Tooling

1.13. Estilo de código y autocorrección

Un problema habitual es que al trabajar en equipo los estilos de código sean diferentes: hay quien prefiere tabulaciones a espacios o las comillas simples a las dobles. Es conveniente llegar a acuerdos de estilos para evitar que cada usuario que toque un fichero modifique el fichero en función de sus preferencias personales.

Para poder unificar estilos, vamos a tener que instalar dos utilidades para nuestro IDE. Esto nada tiene que ver con la configuración de nuestro proyecto, sino con las funcionalidades de nuestro editor. Los que usemos el editor **VSCode** deberemos tener instaladas las extensiones **ESLint** y **Prettier**. Simplificando mucho las funcionalidades de ambos, **ESLint** define el estilo de código y detecta las incongruencias con las reglas establecidas y **Prettier** las corrige.

Instalaremos estas dependencias de desarrollo mediante:

```
npm install -D eslint prettier eslint-config-prettier
```

Y obtendremos estos resultados en nuestro `package.json`:

```
"devDependencies": {
  "@babel/core": "^7.5.5",
  "@babel/preset-env": "^7.5.5",
  "babel-jest": "^24.9.0",
  "eslint": "^6.3.0",
  "eslint-config-prettier": "^6.1.0",
  "jest": "^24.9.0",
  "parcel-bundler": "^1.12.3",
  "prettier": "^1.18.2"
}
```

Ahora podríamos definir nuestras propias reglas o reutilizar conjuntos de reglas que ya han sido probadas y se han convertido en estándar, o al menos en la base de todo proyecto, y modificarlas o extenderlas a necesidad.

Usaremos las reglas básicas y recomendadas, por lo que a continuación crearemos un archivo `eslintrc.json` en la base del proyecto (al mismo nivel que `package.json`, no dentro de la carpeta `src`) con este contenido:

```
{
  "extends": ["eslint:recommended", "prettier"],
  "plugins": [],
  "parserOptions": {
    "ecmaVersion": 2018,
    "sourceType": "module"
  },
  "env": {
    "es6": true,
    "browser": true,
    "node": true,
    "jest": true
  }
}
```

Y otro `prettierrc` también en la raíz del proyecto con este contenido:

```
{  
  "printWidth": 100,  
  "singleQuote": true  
}
```

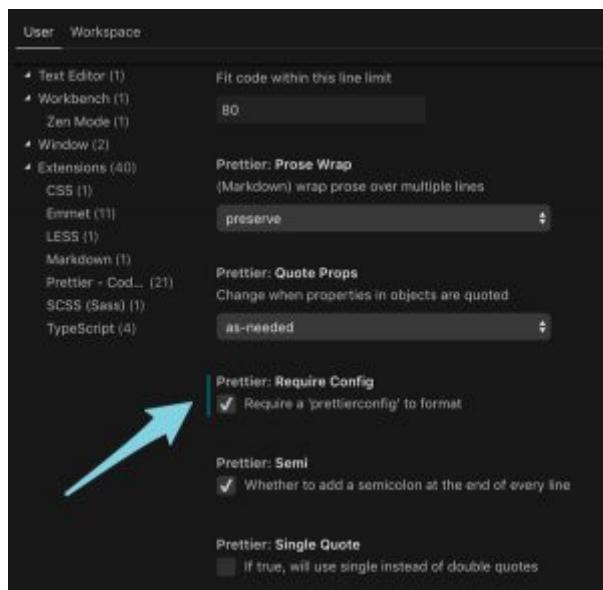
Añadiremos los siguientes *scripts*:

```
"scripts": {  
  "dev": "parcel src/index.html",  
  "format": "prettier --write \"src/**/*.{js,ts}\"",  
  "lint": "eslint \"src/**/*.{js,ts}\" --quiet",  
  "test": "jest --passWithNoTests --silent"  
},
```

Y por último, habilitaremos que nuestro editor formatee el fichero que tengamos abierto cada vez que salvemos su contenido. iremos al menú `Code > Preferences > Settings`, buscaremos la opción `Format On save` y la activaremos.

Los usuarios de VS Code pueden descargar las extensiones de [ESLint](#) y [Prettier](#) dentro del propio VS Code y luego activando las opciones:

- Prettier > Require config > Require a `prettierconfig` to format,
- Poner las opciones `requireConfig` y `editor.formatOnSave` a `true`.



Con esto conseguimos, por ejemplo, que se eliminen de manera automática los espacios y las líneas extra, así como intercambiar las comillas dobles por las simples de forma automática garantizando, sobre todo, que al tratarse de una configuración de proyecto (habilitada por las extensiones añadidas al editor) todos los participante del proyecto (o en este caso del aula) tengan el mismo tipo de código y apliquen las mismas reglas de manera automática.

1. Tooling

1.14. Git (opcional)

Antes se habló de la posibilidad de usar Git como sistema de trabajo. Esto es totalmente opcional, pero en caso de hacerlo incluid un fichero `gitignore` en la base del proyecto que incluya los siguientes archivos y directorios:

```
node_modules  
.cache/  
dist/  
.env  
.DS_Store  
coverage/  
.vscode/
```

1. *Tooling*

1.15. Resumen

Hemos conseguido paso a paso configurar un entorno de trabajo común para todos con funcionalidades muy potentes y avanzadas. Han quedado algunos temas que desarrollar, especialmente el tema de módulos, pero que requieren su propio espacio de desarrollo y explicación.

1. Tooling

1.16. Bibliografía

- [Gestor de versiones de node](#)
- [El formato JSON](#)
- [Yarn](#)
- [Git: control de versiones](#)
- [Jest](#)
- [Pirámide de testing](#)
- [¿Qué es TDD?](#)
- [Nuevas funcionalidades ES6](#)
- [Babel](#)
- [npx](#)
- [ESLint](#)
- [Prettier](#)

2. Introducción a JavaScript

2.1. Planteamiento

En la práctica de **Tooling** hemos desarrollado **JavaScript** sin habernos aventurado aún en las particularidades del lenguaje. Sin prácticamente darnos cuenta hemos usado para el desarrollo TDD la funcionalidad que nos va a servir de hilo argumental para esta unidad: **los módulos**.

Un **módulo** es, en esencia, una pieza de código reusable que exporta objetos específicos y les permite ser importados y usados por otros módulos.

Esta funcionalidad que está intrínsecamente integrada en paradigmas como la programación orientada a objetos no existía en la especificación original de **JavaScript**, puesto que su propósito no incluía un grado de complejidad suficiente para requerirlos.

El porqué, el cuándo y el cómo se añadió la funcionalidad de modularizar partes de código JavaScript es lo que va a articular el texto de esta primera unidad.

2. Introducción a JavaScript

2.2. Introducción

JavaScript comparte muchos de sus rasgos y funcionalidades con otros lenguajes con los que habéis podido tener contacto previamente, pero posee características que lo hacen singular.

Lo que distingue a JavaScript es que es (casi) el único lenguaje de *scripting* de navegador. Su curva de aprendizaje es baja gracias a su flexibilidad y permisividad, por lo que perfiles júnior pueden ser completamente productivos a nivel laboral. Sin embargo, es precisamente este «exceso» de flexibilidad lo que ha condenado a JavaScript a ser considerado históricamente un lenguaje de segunda, comparado con Java, PHP o .Net.

Nota

Recordad que con vuestro usuario de la UOC tenéis acceso a la biblioteca técnica y a los cursos de formación de la plataforma Safari de O'Reilly. Os dejaré indicados los enlaces a dicha plataforma con el sufijo Safari en el nombre de los enlaces.

El ya clásico y fenomenal libro de [Douglas Crockford](#), [JavaScript The Good Parts \[Safari\]](#), ha sido siempre una referencia desde su primera edición, cuando el lenguaje era mucho más inmaduro y menos usado que actualmente.

Hasta ese momento nadie en la industria se tomaba en serio **JavaScript**, en una época en la que el contenido de las páginas web venía renderizado desde el servidor y la interacción en la parte de usuario se limitaba únicamente a los formularios.

El libro tiene un planteamiento y un contenido brillantes, y permitió que la gente apreciara las partes realmente potentes del lenguaje, pero al ganar tracción provocó que se estigmatizara JavaScript como un lenguaje mal planificado, diseñado e implementado, puesto que igual que tenía *good parts* seguía teniendo *bad parts*.

Vamos a intentar hacer un recorrido histórico para entender de dónde venimos, dónde estamos, a dónde vamos y comprender cada una de las transformaciones del lenguaje.

2. Introducción a JavaScript

2.3. Historia y contextualización

Difícilmente, en 1995, [Brendan Eich](#) podía anticipar para lo que se estaría usando el lenguaje en el que estaba trabajando 25 años después.

JavaScript no es perfecto, pero todo lo que se especificó tenía un sentido, aunque para algunos que vengan de otros lenguajes no les parezca evidente. Es aceptable valorar a nivel personal y señalar aspectos del lenguaje como *bad parts*, pero sería injusto obviar partes del lenguaje sin realmente haber sometido a un sesudo juicio su posible potencial.

1. [Motivación inicial de Netscape para crear JavaScript](#)
2. [Infografía sobre la cronología de JavaScript](#)
3. [Explicación extendida sobre la cronología](#)
4. [dotJS 2017 – Brendan Eich – A Brief History of JavaScript \[Youtube\]](#)

Tened presente y valorad que cuando **Netscape** contrató a **Brendan Eich** en abril de 1995, se le dijo que tenía diez días para crear y producir un prototipo funcional de un lenguaje de programación que se ejecutara en el navegador de Netscape.

Os recomiendo encarecidamente que veáis el vídeo de Brendan Eich que os he dejado en los enlaces de más arriba puesto que nadie mejor que él puede entender y valorar el propósito inicial de su proyecto, su evolución, su situación actual y el futuro que le depara.

2. Introducción a JavaScript

2.4. Las características fundamentales de JavaScript

Para entender la visión inicial de JavaScript, debemos conocer las tres características que lo hacen tan especial:

1. Lenguaje funcional

2. Herencia

3. Cadena de prototipo

Revisad esta explicación en [MDN](#) y esta de [Tyler McGinnis](#). Con diez días para diseñar un lenguaje no pudo pensar todo desde cero y recibió influencias de muchos otros lenguajes que presentaban sintaxis y paradigmas que tuvo a bien considerar para la primera implementación de JavaScript.

Tanto **herencia** como **cadena de prototipo** son dos aspectos básicos del funcionamiento y la concepción de JavaScript. Ambas siguen siendo partes intrínsecas al lenguaje, aunque en mi experiencia no se usan activamente para planificar y desarrollar proyectos.

Sin que ello signifique que no se puedan hacer propuestas válidas bajo estos paradigmas, **JavaScript** incluye actualmente herramientas y funcionalidades más convencionales e igualmente potentes para realizar programación de aplicaciones modernas.

El último ejemplo más o menos exitoso en el que recuerdo la aplicación de la cadena de prototipo de forma exhaustiva es [Backbone](#), el primer *framework* de amplia adopción que abrió el camino para sus hermanos [Angular](#), [Ember](#) y [React](#). La propuesta era mucho más rudimentaria que las actuales API de estos últimos, pero un paso necesario al fin y al cabo.

2. Introducción a JavaScript

2.5. Evolución

JavaScript evoluciona según un proceso de realimentación en el que las exigencias de las nuevas aplicaciones obligan a desarrollar de manera orgánica y espontánea nuevas funcionalidades como extensiones que los propios ingenieros implementan.

Cada cierto tiempo el grupo de mantenimiento y desarrollo **ECMAScript (TS39)** lanza una nueva especificación con las funcionalidades más comúnmente usadas o requeridas como estándares oficiales.

Podéis comprobar la evolución de las diferentes versiones de ECMAScript en este enlace a [Wikipedia](#); la última versión, **ES6**, es la que usamos actualmente.

En este punto hay que puntualizar cómo funciona la adopción de estos estándares por parte de los diferentes navegadores: cuando una nueva versión de la especificación se hace pública ningún navegador lo soporta de forma oficial. Las nuevas versiones de los navegadores van incluyendo paulatinamente las nuevas funcionalidades en sus motores e intérpretes de JavaScript.

Por poner los dos ejemplos más claros:

1. [Mozilla Firefox Nightly](#).

2. [Google Chrome Canary](#).

Las diferentes plataformas ponen a disposición de los usuarios las últimas versiones de sus navegadores, en las que trabajan en desarrollar las últimas funcionalidades para que los desarrolladores puedan probar la especificación de forma nativa.

Sin embargo, a pesar de lo satisfactorio y fluido de esto, hay que pensar que nuestros clientes, los usuarios finales de las páginas y aplicaciones del lado de cliente que creemos, no necesariamente tienen que tener la última versión de nuestro navegador. Por ello, nuestras herramientas de *tooling* deben someter nuestro código **ES6** (*import/export*) a un proceso de transpilación a un estándar previo mucho más extendido, **ES5**, para asegurar que la mayoría de nuestros usuarios tengan soporte en su navegador.

En **JavaScript** la herramienta que realiza este proceso más aceptada es [Babel.js](#). Para que os hagáis una idea de un ejemplo real: [la función flecha \(arrow function\)](#).

```
// Nuestro código ES6 con el setup de la práctica de Tooling
[1, 2, 3].map((n) => n + 1);

// El código final transpilado a ES5
[1, 2, 3].map(function(n) {
    return n + 1;
});
```

Si queréis hacer pruebas sobre la transformación de código, podéis usar el [REPL de Babel.js](#).

Es interesante echar un vistazo a las reflexiones del creador de **JavaScript** para entender las [cosas que quizás hubiera hecho diferente](#) de haber anticipado el marco y uso actual de esta tecnología que desarrolló.

2. Introducción a JavaScript

2.6. Situación actual

Al empezar el curso os hablé de lo cambiante y exigente que es el escenario **front-end**; una industria que no para de avanzar y donde los estándares, las tecnologías y los paradigmas cambian, avanzan y mejoran a velocidades de vértigo.

Aterrizamos en la asignatura, el lenguaje y la tecnología en un momento de cambio y es eso precisamente –junto con el incremento de la demanda de profesionales– lo que hace que cada día se abran más plazas en el sector:

1. El incremento en valor absoluto de la demanda de profesionales.
2. Los huecos dejados por profesionales que no se adaptan a los nuevos estándares.

Aprovechar esta ventana de oportunidad requiere una constante inversión en aprendizaje y actualización y, en el caso de atacar por primera vez esta tecnología, además, contextualizar lo actual con lo pasado y lo futuro.

Os recomiendo que uséis **ES6** en las prácticas. Creo que hacer el esfuerzo de aprender los últimos estándares es lo más adecuado, pero también es necesario que conozcáis su equivalente **ES5** para poder entrar a comprender cualquier base de código en **JavaScript** independientemente de la versión de la especificación usada.

2. Introducción a JavaScript

2.7. Buceando en JavaScript

2.7.1. Introducción

Nota

En anteriores ediciones de la asignatura hemos utilizado Eloquent Javascript como hilo conductor, pero debido a los comentarios y las valoraciones de los alumnos hemos sometido el temario a cambios y ajustes. Durante este trimestre usaremos dos materiales diferentes de forma simultánea, por lo que os agradecería que me hagáis llegar cualquier feedback sobre el valor de uno frente al otro o sobre vuestra percepción de facilidad de aprendizaje o conveniencia en cuanto a aprendizaje. El contenido y los temarios se solapan, así que si consideráis que con uno de los dos materiales tenéis suficiente, no tenéis por qué revisar el otro siempre y cuando hayáis entendido y asimilado los conceptos asociados.

Como comentábamos al principio, **Douglas Crockford** ayudó a dar empaque a **JavaScript** como lenguaje de programación a costa de separar las partes buenas de las malas.

Nota

No dudéis en revisar esta clase maestra sobre [JavaScript, the Good Parts \[Safari\]](#).

Tiempo después, Kyle Simpson argumentaba que no había tales partes malas, sino una incapacidad de comprensión total de las motivaciones tras esas partes, y por tanto se les daba un mal uso por condicionantes previos por no encajar con las propuestas de otros lenguajes y metodologías.

Kyle Simpson nos presenta un **JavaScript** donde todas sus funcionalidades son buenas y malas en función del uso que se les dé y será su serie de libros You Don't Know JavaScript la que usaremos para aprenderlas.

La **PEC** de esta unidad consiste en una serie de ejercicios sencillos agrupados por temática y, en general, de diferente complejidad que nos servirán como estiramiento y práctica de las peculiaridades de **JavaScript**. Es común en lo relativo a programación referirnos a estos ejercicios como katas, término propio de las artes marciales pero con el mismo propósito: practicar y aprender.

2. Introducción a JavaScript

2.7. Buceando en JavaScript

2.7.2. Módulos

Llegados a este punto, tenemos suficiente experiencia y conocimiento para entender la evolución de la modularización de JavaScript. Hemos usado módulos en el desarrollo TDD: implementamos la función en un módulo y la exportamos para más tarde importarla en su fichero spec y poder pasársela todos los test.

Para ganar contexto, os recomiendo las explicaciones del genial ingeniero de Google [Addy Omani](#) sobre:

1. [El patrón módulo \(*module pattern*\)](#).
2. [El patrón de módulo «revelador» \(*revealing module pattern*\)](#).

Estas dos maneras de organizar el código permiten:

1. Mejorar el **mantenimiento** del código al trabajar con unidades más pequeñas e independientes.
2. Separar funcionalidades y facilitar su **reusabilidad** en diferentes partes del código.
3. Permitir la **privacidad** de métodos y variables gracias al closure de las funciones.

Esta funcionalidad tan obvia y útil no existía de inicio, aunque los patrones de diseño mencionados anteriormente habilitaban los proyectos para tener cierta modularización. En 2009 apareció la especificación de [CommonJS](#), que permitía reutilizar partes del código con una API más estricta y asentada.

Sin entrar en muchos detalles, hay una versión asíncrona de CommonJS llamada AMD que se desarrolló para su integración en las peticiones del navegador. Conceptualmente es el mismo procedimiento pero con el añadido de la gestión asíncrona de las peticiones.

Por último y más reciente, la última revisión de ECMAScript introdujo dentro del estándar por primera vez el concepto de módulos, que es lo que hemos utilizado en nuestro desarrollo TDD y que volveremos a utilizar de manera exhaustiva en las próximas PEC:

1. [YDKJS] [Módulos ES5 y Módulos ES6](#)
2. [Eloquent JS] [Módulos](#)
3. [FreeCodeCamp] [Módulos en JavaScript](#)

2. Introducción a JavaScript

2.8. Conceptos básicos

2.8.1. Lenguaje no tipado

JavaScript es un lenguaje no tipado. Esto significa que cuando definimos las variables con `var`, `let` o `const` no especificamos el tipo de valor que van a albergar y solo son espacios de memoria reservados para lo que queramos almacenar en ellas. El lenguaje es tan flexible que hasta podemos almacenar distintos tipos de valores durante la vida de una misma variable.

Esto es muy potente y agradecido, pero es el foco potencial de muchos problemas derivados de la mala gestión del tipo de datos. Por ejemplo:

```
var miVariable = 'esto es un string'; // 'esto es un string'  
miVariable.toString(); // 'esto es un string'  
miVariable.toUpperCase(); // 'ESTO ES UN STRING'  
  
miVariable = 9; // 9  
miVariable.parseInt(); // 9  
miVariable2 = miVariable.toString(); // '9'  
miVariable2.toUpperCase(); // '9'  
miVariable.toUpperCase(); // error
```

Nota

1. [Number.parseInt\(\)](#).
2. [Number.toString\(\)](#).
3. [String.toString\(\)](#).
4. [String.toUpperCase\(\)](#).

En este ejemplo hemos cambiado el tipo de valor almacenado en una variable y ejecutado sobre él métodos propios de la nativa padre (`Number` y `String`), uno común `toString` y otros propios de cada uno de ellos (`Number.parseInt()` y `String.toUpperCase()`). Solo nuestro control sobre lo que hace el código y el control sobre el tipo de datos podrá evitar que los programas presenten errores de este tipo.

Por supuesto, siempre podemos hacer condicionales de comprobación de tipo:

```
if (typeof miVariable === "number") {  
    miVariable.parseInt(); // 9  
}
```

Aunque nuestro código puede volverse excesivamente verboso en caso de necesitar comprobaciones de este tipo a cada invocación. Según mi experiencia, esto nunca está de más, puesto que hace más fuerte el código, pero solo es realmente necesario en las interacciones con bases de datos o el DOM en forma de campos de formulario, ya que aunque lo que recibamos sea un número, este puede venir en forma de `string` y, como hemos visto, no es lo mismo tener una variable con el valor `9` que con el valor '`9`'.

También podemos simplificar nuestro código creando funciones auxiliares tipo:

```
function isNumber(num) {  
    if (typeof num === "number") {  
        return true;  
    } else {  
        return false;  
    }  
}
```

O su versión reducida:

```
function isNumber(num) {  
    return typeof num === "number";  
}
```

JavaScript contempla los siguientes tipos de datos:

1. Boolean
2. Null
3. Undefined
4. Number
5. String

Y los **Objetos** como estructura que combina un grupo de variables de los tipos anteriormente citados.

2. Introducción a JavaScript

2.8. Conceptos básicos

2.8.2. Objetos

Cuando programamos solemos querer o bien simular algún modelo de datos existente fuera del entorno de nuestro código o bien parametrizar algún aspecto de una realidad. Prácticamente todas las aplicaciones clásicas de escritorio pretenden traspasar al mundo digital una realidad analógica, como pueden ser las cuentas bancarias o los expedientes médicos.

Estos modelos suelen ser complejas relaciones entre elementos de diverso tipo. Por ejemplo: un expediente médico tiene un titular, un año de nacimiento, un histórico de recetas, un historial de visitas...

No existe una estructura de datos nativa a ningún lenguaje que pueda parametrizar ese modelo directamente. Para poder modelizar estos datos es para lo que se usan los objetos.

Recuperando el objeto del expediente médico, podríamos tener un objeto que recogiera ese tipo de datos:

```
var expediente = {
    paciente: 'David Lampon',
    nacimiento: 1979,
    recetas: [
        {
            medicamento: 'aspirina',
            fecha: '2009-10-12',
            dias: 5,
            recurrente: false
        },
        {
            medicamento: 'gelocatil',
            fecha: '2010-04-21',
            dias: 15,
            recurrente: true
        },
    ],
    visitas: [
        {
            medico: 'Jorge Sola',
            id: 10098453
            fecha: '2009-10-12'
        },
        {
            medico: 'Jorge Sola',
            id: 13415012
            fecha: '2010-04-21'
        },
    ],
    recuperarVisitas: function() {
        return this.visitas;
    }
};
```

En este ejemplo vemos varias estructuras complejas anidadas que conforman el modelo de datos que podríamos tener en un CAP. La estructura de un objeto se basa en pares de llaves y su valor. La potencia de los objetos radica en hacer flexible la estructura de las nativas y poder extenderla, agruparla y adaptarla a nuestro gusto.

El acceso a datos puede hacerse mediante sintaxis de corchetes o de puntos. En nuestro caso estas dos líneas recuperarán el mismo dato:

```
var nombre;
nombre = expediente['nombre'];
nombre = expediente.nombre;
```

Ambas consiguen el mismo propósito y la empleada usualmente es la de puntos, aunque en algunos casos, como cuando heredamos nombres de llave con espacios, preferimos sintaxis de corchetes.

Para accesos más profundos a la estructura de datos debemos encadenar selectores de este modo (supongamos primera receta):

```
var medicamento = expediente['recetas'][0]['medicamento'];
var medicamento = expediente.recetas[0].medicamento;
```

Aquí podemos ver que, a pesar de lo verboso de ambas notaciones, la de puntos es más directa.

Podemos concluir que los objetos son lo que nos permite dotar a nuestras aplicaciones de estructuras de datos complejas y mimetizar sistemas analógicos o tradicionales en el mundo digital.

2. Introducción a JavaScript

2.8. Conceptos básicos

2.8.3. Clases

Las clases son una evolución de estas estructuras. En muchos otros lenguajes las clases son la base de todos los programas, todo empieza y acaba en las clases en los modelos de **programación orientada a objetos** (OOP).

En JavaScript, sin embargo, tenemos una estructura del lenguaje basada en herencia de prototipo, que es un paradigma distinto en concepción al de trabajo con clases.

Debido a la progresiva adaptación de JavaScript a más y más entornos, a la aparición de node como lenguaje de *back-end* y a la necesaria conversión de ingenieros de sistemas tradicionales a JavaScript para cubrir la demanda de mercado en la especificación ES6 o ES2015, se introdujo la sintaxis de clases en JavaScript, inexistente hasta entonces.

La sintaxis de clases esconde por debajo una gestión del prototipo que resulta transparente para el desarrollador, pero que permite tener la potencia de ambos paradigmas sin tener que renunciar a ninguno de ellos.

Pasando el ejemplo anterior de objeto a clase, podríamos obtener algo tal que así:

```
class Expediente {
  constructor(paciente, nacimiento, recetas, visitas) {
    this.paciente = paciente;
    this.nacimiento = nacimiento;
    this.recetas: recetas;
    this.visitas: visitas,
  }

  get visitas() {
    return this.visitas;
  }
}

const nuevoExpediente = new Expediente("David Lampon", 1979, [], []);
```

Las diferencias entre el ejemplo del objeto y el de clase es que con el objeto tenemos una única instancia de esos datos y con la clase, una plantilla para crear todas las instancias que queramos. A efectos prácticos, ambos tienen propiedades y métodos que pueden ejecutarse sobre sí mismo usando el `this` como palabra clave de acceso a los datos existentes.

Las clases, en caso de necesitar inicializarse con datos al instanciarse, deben implementar el método `constructor`, un método al que se le pasan los datos que se asignarán a las propiedades de la instancia.

Las clases pueden extenderse unas sobre otras, y existen patrones más avanzados de herencia y ofuscación. Un ejemplo es la manera de crear componentes en React como podemos ver [aquí](#):

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Sin embargo, debido al marco limitado de esta asignatura, obviaremos estas funcionalidades avanzadas puesto que no las vamos a utilizar.

2. Introducción a JavaScript

2.8. Conceptos básicos

2.8.4. Scope

Entendemos por scope cada una de las zonas que limitan el acceso a los datos dentro de nuestro código. Es la forma de poner límites a qué código ve qué cuando se ejecuta.

Imaginemos este caso:

```
var personaje = 'Sonic';

function funcionQueDefineUnScopeLocal() {
    console.log(personaje); // Sonic
}

console.log(personaje); // Sonic
```

Existe un scope global que abarca todo nuestro proyecto y el código que escribamos conocido como scope global, que va desde la línea número cero hasta la última. Todo lo que definamos en este scope será accesible por todos los scopes creados de forma anidada.

Los corchetes {} definen un nuevo scope; en este ejemplo `funcionQueDefineUnScopeLocal` crea su propio scope, pero es capaz de leer el global al ser uno superior. Por ello podemos acceder al valor de la variable `personaje`.

Modificar el valor dentro de un scope local lo extiende al scope global:

```
var personaje = 'Sonic';

function funcionQueDefineUnScopeLocal() {
    personaje = Tails;
    console.log(personaje); // Tails
}

console.log(personaje); // Tails
```

Pero ¿qué sucedería en este caso?

```
var personaje = 'Sonic';

function funcionQueDefineUnScopeLocal() {
    var nemesis = "Robotnik";
    console.log(personaje); // Sonic
    console.log(nemesis); // Robotnik
}

console.log(personaje); // Sonic
console.log(nemesis); // undefined
```

Podemos comprobar cómo los scopes anidados pueden leer las variables de scopes superiores pero no al revés.

Existe un caso extremo en el que no podríamos recuperar el valor de un scope superior, y es si definimos en nuestro scope local una variable del mismo nombre que uno superior:

```
var personaje = 'Sonic';

function funcionQueDefineUnScopeLocal() {
    var personaje = 'Tails';
    console.log(personaje); // Tails
}

console.log(personaje); // Sonic
```

En esta situación no habría manera de recuperar el valor de «personaje» del scope global dentro del scope local.

Hasta ES6, JavaScript era un lenguaje de scope funcional; solo las funciones definían scopes. Desde ES6 cualquier par de corchetes {} definen un scope, por lo que ahora disponemos del ya existente *functional scope* pero también de *lexical scope*. La aparición del scope léxico hizo necesaria la creación de nuevas definiciones de variables que respetaran ese nuevo scope léxico, ya que **var** está diseñada para ser usada en un paradigma de scope funcional, lo que da lugar a **const** y **let**. Veamos este ejemplo para comprobar la diferencia:

```
var personaje = 'Sonic';

if(personaje === "Sonic") {
    var personaje = 'Tails';
    console.log(personaje); // Tails
}

console.log(personaje); // Tails
```

```
var personaje = 'Sonic';

if(personaje === 'Sonic') {
    let personaje = 'Tails';
    console.log(personaje); // Tails
}

console.log(personaje); // Sonic
```

Aunque **var** tiene un sentido y unos casos de uso claros, es habitual para la mayoría de los programadores hacer un uso exclusivo de **let** y **const** en lugar de **var** por hacer más coherente las bases de código en las que se trabaja. Es interesante y útil entender y conocer los usos y las particularidades de **var** pero, en mi experiencia práctica, la realidad es que se usa cada vez menos.

2. Introducción a JavaScript

2.8. Conceptos básicos

2.8.5. This

Como hemos ido viendo hasta ahora e intuyendo un poco su uso, this es una referencia propia a una estructura de datos de forma dinámica. El valor de this depende del momento y del lugar desde el que se indique.

La palabra **this** se refiere al objeto que está en ejecución en el código en ese preciso instante. Es decir, cada función, mientras se ejecuta, tiene una referencia a sí misma accesible mediante el **this**. Gracias a esto podemos acceder a su contexto de ejecución, que representa el cómo se ha llamado esa función. **El this solo depende de cómo y dónde se llame la función, no de su implementación.**

```
function fontanero() {
    console.log(this.nombre);
}

var nombre = "Mario";
var luigi = { nombre: "Luigi", color: 'verde', fontanero };

fontanero();          // "Mario"
luigi.fontanero();   // "Luigi"
```

En este código la función **fontanero()** presenta por pantalla this.name y su valor depende de cada contexto de ejecución. En el caso de la primera llamada, no se especifica ningún contexto, por lo que se toma el contexto por defecto, que se trata del espacio global. Existe una variable nombre en el espacio global que, al ejecutar **fontanero()**, se recupera y se presenta por pantalla. En el caso de ejecutar la función sobre el objeto luigi, el contexto de ejecución pasa a ser ese objeto, que tiene una propiedad hombre y que en este momento de la ejecución responde a la petición de **this.nombre** por parte de la función **fontanero()**.

Existen maneras adicionales de especificar el contexto de ejecución usando los métodos call, apply y bind.

```
function fontanero() {
    console.log(this.nombre);
}

var mario = {nombre: 'Mario', color: 'rojo' };
var luigi = { nombre: 'Luigi', color: 'verde' };

fontanero.call(mario);   // "Mario"
fontanero.call(luigi);  // "Luigi"
```

Cualquier función puede ejecutarse con un call o apply al final, pero en caso de añadir un parámetro en la llamada este definirá el contexto de ejecución.

El caso de bind es ligeramente diferente porque especifica el contexto de ejecución pero no lo ejecuta. Digamos que prepara la función para ser ejecutada con el contexto deseado pero la deja en *standby* hasta que se desee ejecutar.

```
element.addEventListener('click', this.onclick.bind(this))
```

Este comportamiento es propio de los **eventListeners** cuando se trabaja con el DOM, ya que se preparan pero se ejecutarán de manera asíncrona cuando el usuario lance acciones o navegue por nuestro sitio o aplicación web. De todos modos, esto solo

resulta necesario si deseamos acceder al contexto de ejecución de nuestro módulo (datos o funciones).

2. Introducción a JavaScript

2.8. Conceptos básicos

2.8.6. Closure

Un closure es la combinación de una función y el scope léxico en el que se declaró dicha función. Es decir, la función definida en el closure «recuerda» el ámbito en el que se ha creado y tiene acceso a los datos de forma persistente.

Cuando creamos una función que devuelve un valor:

```
function muestraNombre(nombre) {
    console.log(nombre);
}

muestraNombre("Megaman");
```

Si pensamos en el ciclo de vida de la variable nombre, vemos que es completamente volátil. Existe en el momento en el que se invoca la función, se presenta en el console.log y desaparece.

Hay situaciones en las que queremos poder tener cierta persistencia, poder recordar cómo hemos construido esa función, con qué parámetro trabajamos de inicio.

```
function potenciaDe(base) {
    return function(exponente) {
        return Math.pow(base, exponente);
    }
}

const potenciaDeDos = potenciaDe(2);
const potenciaDeDosALaCuatro = potenciaDeDos(4); // 16

const potenciaDeTres = potenciaDe(3);
const potenciaDeDosALaDos = potenciaDeTres(2); // 9
```

Cuando ejecutamos potenciaDeDos(4) y potenciaDeTres(2) mantenemos un estado dentro de nuestra función para recordar cuál era la base con la que definimos nuestra función original invocando a potenciaDe(base).

Como receta, la manera de implementar un closure, una memoria para una función, es devolviendo una función como resultado en la que se use el parámetro inicial como parte del valor retornado.

2. Introducción a JavaScript

2.8. Conceptos básicos

2.8.7. Prototype

Todos los tipos en JS provienen de una nativa original como **String** o **Number** (con mayúscula inicial). Estas nativas tienen un prototipo (por ejemplo, `String.prototype`) que a efectos prácticos es como un objeto donde residen todos los métodos aplicables sobre un *string*.

Cuando instanciamos un nuevo *string* así:

```
const miString = "esto es un string";
```

Hacemos (más o menos) lo mismo que:

```
const miString = new String("esto es un string");
```

Que quizá queda más claro a nivel de instanciación. Al instanciar, nuestra nueva variable hereda en ese proceso todos los métodos del prototipo de la nativa. Cuando hablamos de JavaScript como lenguaje de herencia de prototipo, nos referimos a esto.

Podemos extender este prototipo para añadir un nuevo método. No es en absoluto complejo si pensamos en el prototipo como un objeto ya existente al que quieras añadirle un método. Tan simple como eso:

```
String.prototype.miNuevoMetodo = function(param) {  
    [...]  
};
```

Una vez hecho esto, dentro de nuestro código todos los *strings*, tanto los nuevos como los existentes, tendrán acceso a *miNuevoMetodo*.

Este método se invocará como cualquier otro de String, como por ejemplo, `toUpperCase`. Si revisamos su implementación, podríamos ver cómo podemos acceder al *string* que invoca el método y cómo se puede trabajar sobre él. Por ejemplo:

```
String.prototype.numeroDeAparicionesDeLaLetra = function(letra) {  
    let cont = 0;  
  
    for(let i = 0; i <= this.length; i = i + 1) {  
        if (this[i] === letra) {  
            cont = cont + 1;  
        }  
    }  
  
    return cont;  
}  
  
const miString = "Asturias";  
console.log(miString.numeroDeAparicionesDeLaLetra("a")); // 1
```

En este caso solo encontrará la segunda «a», la minúscula. De querer contabilizarlas todas independientemente de si son mayúscula o minúscula, deberíamos hacer transformaciones previas con `.toUpperCase()` o `.toLowerCase()` para igualarlo.

3. Gestión de la asincronía

3.1. Planteamiento

Ahora que ya hemos estirado y calentado mediante katas nuestro conocimiento y habilidad con JavaScript, vamos a estudiar y trabajar una de sus características esenciales que tiene que ver con la esencia de lo que conocemos como desarrollo web: la **asincronía**.

3. Gestión de la asincronía

3.2. Introducción

Si le preguntamos a cualquier usuario qué cree que sucede cuando solicita una página web, nos estará explicando qué es la asincronía de la manera más llana y sencilla.

Cuando un usuario solicita una página web puede ser:

1. Escribiendo la url en la barra del navegador.
2. Accediendo a través de un clic en un enlace.

En ambos casos lo que sucede es una serie de acciones en las diferentes capas de la estructura OSI que acabarán, en el mejor de los casos, con la renderización en el navegador de la página solicitada.

Sin embargo, desde que se efectúa la solicitud hasta que la página se presenta en el navegador transcurre un tiempo, es decir, no es inmediato, y eso es precisamente lo que se conoce como asincronía en el mundo web: **la diferencia temporal desde que se hace una petición de datos hasta que estos están disponibles.**

3. Gestión de la asincronía

3.3. HTTP

La asincronía debida al protocolo HTTP es la más evidente y de la que todos somos conscientes. La otra que veremos y con la que trabajaremos es intrínseca al desarrollo web y su gestión no ha sido una práctica extendida hasta hace pocos años. Veremos el porqué más adelante.

Repasando conceptos básicos y sin entrar en muchos detalles, el retardo acumulado desde que se solicita una página web hasta que podemos visualizar el contenido se debe a:

1. Contacto con el servicio de DNS.
2. Resolución del nombre de dominio en relación con la IP.
3. Devolución de la IP.
4. Contacto con el servidor cuya dirección IP hemos obtenido.
5. Devolución del fichero solicitado (si existe) o cabecera de error.
6. Interpretación y renderizado del fichero obtenido o mensaje de error.

Este proceso no debería tardar más de un segundo en ningún caso con una conexión de internet de alta velocidad (como las que tenemos desde hace tiempo) y con unos equipos con capacidad de procesamiento alta (como los que tenemos desde hace tiempo). En países o contextos con conexiones lentas y dispositivos móviles antiguos, el proceso de petición, carga y renderizado puede alargarse más tiempo.

Los ficheros que pueden solicitarse a un servidor pueden ser imágenes, ficheros de vídeo y audio, elementos de *stream* y, en general, cualquiera aceptado por las políticas del servidor, pero el caso que vamos a utilizar es el esencial y básico: los ficheros html.

1. http: Hypertext Transfer Protocol
2. html: HyperText Markup Language

Estos dos elementos (protocolo y especificación de lenguaje) van muy de la mano y son los que juntos cimientan **internet** como la conocemos.

Cuando un navegador recibe la respuesta a su solicitud (que suele ser el archivo index.html en la raíz del servidor de sitios como Google, la UOC o cualquier otro) lo que sucede es que se interpreta su contenido empezando por la cabecera del <head>, que es donde se encuentran los metadatos.

Recordad que lo que recibe el navegador es un fichero html. En el momento de recepción no tenemos ni estilos, ni fuentes, ni imágenes ni JavaScript. Vamos a ver cómo todo eso se solicita y en qué orden.

MDN web docs
mos://a

Buscar

Tecnologías ▾ Referencias y guías ▾ Comentarios ▾ Iniciar sesión

Solicitudes síncronas y asíncronas

Idiomas Editar

Saltar a: Peticiones asíncronas Synchronous request See also

Traducción en curso

XMLHttpRequest: soporta solicitudes síncronas y asíncronas, pero la más preferida es la asíncrona por razones de rendimiento

Las solicitudes síncronas bloquean la ejecución del código, mientras se procesa la solicitud, dejando a la pantalla congelada y dando una experiencia de usuario poco agradable

Network tab showing network requests for the synchronous request page. The table lists 17 requests with details like name, method, status, type, initiator, size, time, and waterfall chart.

| Name | Method | Status | Type | Initiator | Size | Time | Waterfall |
|--------------------------------------|--------|--------|--------|---------------------|---------|------|-----------|
| Synchronous_and_Asyncronous_Re... | GET | 200 | doc | Other | 18.4... | 1... | |
| ZillaLab-Regularsubset.0bc3f0b147... | GET | 200 | font | Synchronous and... | 33.6... | 2... | |
| mdn.b65fb8be1142.css | GET | 200 | styl | Synchronous and... | 15.4... | 1... | |
| wild.41c0d34961.css | GET | 200 | styl | Synchronous and... | 15.0... | 1... | |
| locale-es-520edocstable.css | GET | 200 | styl | Synchronous and... | 1.1... | 1... | |
| javascript.10ef83ae097.js | GET | 200 | script | Synchronous and... | 4.0... | 1... | |
| main.0ec22a564ec2.js | GET | 200 | script | Synchronous and... | 46.5... | 7... | |
| wild.810e965e020.js | GET | 200 | script | Synchronous and... | 8.2... | 1... | |
| newsletter.85c5c2892e5.js | GET | 200 | script | Synchronous and... | 1.9... | 1... | |
| analytics.js | GET | 200 | script | Synchronous and... | 17.8... | 1... | |
| web-docs-sprites.22a9a085cf14.svg | GET | 200 | img | Synchronous and... | 3.8... | 2... | |
| panel.0ca34437cd111.svg | GET | 200 | img | Synchronous and... | 1.1... | 1... | |
| file.7ed103789865.svg | GET | 200 | img | Synchronous and... | 923.0 | 1... | |
| ZillaLab-Boldsubset.e96c1958c58... | GET | 200 | font | Synchronous and... | 33.8... | 2... | |
| collect?r=14_wi73&ep=14a+8963... | GET | 200 | gif | analytics.js15 | 63 B | 1... | |
| syntax-prism.81bfe1eaaf8.js | GET | 200 | script | wiki.1Dab266a929... | 8.7... | 5... | |
| favicon32-737da72d0ea1.png | GET | 200 | png | Other | 101... | 3... | |

17 requests | 211 KB transferred | Finish: 1:31 s | DOMContentLoaded: 990 ms | Load: 1.25 s

Peticiones asíncronas

Si se utiliza XMLHttpRequest de forma asíncrona, recibirá una devolución de llamada cuando los datos se hayan recibido. Esto permite que el navegador continúe funcionando de forma normal mientras se procesa la solicitud.

Ejemplo: Enviar un archivo a la consola

Este es el uso más simple de la asíncrona XMLHttpRequest.

En esta imagen de la pestaña Network podéis ver cada una de las solicitudes que tiene lugar en forma de cascada. Pensad que esto es lo que sucede la primera vez que solicitáis la página. Para las posteriores, la caché interna del navegador puede evitar algunas solicitudes innecesarias si los assets de la página no han cambiado.

Revisemos este ejemplo de fichero html:

MDN web docs
mos://a

Buscar

Tecnologías ▾ Referencias y guías ▾ Comentarios ▾ Iniciar sesión

Solicitudes síncronas y asíncronas

Idiomas Editar

Saltar a: Peticiones asíncronas Synchronous request See also

Traducción en curso

XMLHttpRequest: soporta solicitudes síncronas y asíncronas, pero la más preferida es la asíncrona por razones de rendimiento

Las solicitudes síncronas bloquean la ejecución del código, mientras se procesa la solicitud, dejando a la pantalla congelada y dando una experiencia de usuario poco agradable

Network tab showing network requests for the asynchronous request page. The table lists 17 requests with details like name, method, status, type, initiator, size, time, and waterfall chart.

| Name | Method | Status | Type | Initiator | Size | Time | Waterfall |
|--------------------------------------|--------|--------|--------|---------------------|---------|------|-----------|
| Synchronous_and_Asyncronous_Re... | GET | 200 | doc | Other | 18.4... | 1... | |
| ZillaLab-Regularsubset.0bc3f0b147... | GET | 200 | font | Synchronous and... | 33.6... | 2... | |
| mdn.b65fb8be1142.css | GET | 200 | styl | Synchronous and... | 15.4... | 1... | |
| wild.41c0d34961.css | GET | 200 | styl | Synchronous and... | 15.0... | 1... | |
| locale-es-520edocstable.css | GET | 200 | styl | Synchronous and... | 1.1... | 1... | |
| javascript.10ef83ae097.js | GET | 200 | script | Synchronous and... | 4.0... | 1... | |
| main.0ec22a564ec2.js | GET | 200 | script | Synchronous and... | 46.5... | 7... | |
| wild.810e965e020.js | GET | 200 | script | Synchronous and... | 8.2... | 1... | |
| newsletter.85c5c2892e5.js | GET | 200 | script | Synchronous and... | 1.9... | 1... | |
| analytics.js | GET | 200 | script | Synchronous and... | 17.8... | 1... | |
| web-docs-sprites.22a9a085cf14.svg | GET | 200 | img | Synchronous and... | 3.8... | 2... | |
| panel.0ca34437cd111.svg | GET | 200 | img | Synchronous and... | 1.1... | 1... | |
| file.7ed103789865.svg | GET | 200 | img | Synchronous and... | 923.0 | 1... | |
| ZillaLab-Boldsubset.e96c1958c58... | GET | 200 | font | Synchronous and... | 33.8... | 2... | |
| collect?r=14_wi73&ep=14a+8963... | GET | 200 | gif | analytics.js15 | 63 B | 1... | |
| syntax-prism.81bfe1eaaf8.js | GET | 200 | script | wiki.1Dab266a929... | 8.7... | 5... | |
| favicon32-737da72d0ea1.png | GET | 200 | png | Other | 101... | 3... | |

17 requests | 211 KB transferred | Finish: 1:31 s | DOMContentLoaded: 990 ms | Load: 1.25 s

Styles tab showing CSS styles for the page. It highlights several red boxes around specific elements and properties, such as the `margin`, `border`, `padding`, and `background-color` properties.

Peticiones asíncronas

Si se utiliza XMLHttpRequest de forma asíncrona, recibirá una devolución de llamada cuando los datos se hayan recibido. Esto permite que el navegador continúe funcionando de forma normal mientras se procesa la solicitud.

Ejemplo: Enviar un archivo a la consola

Este es el uso más simple de la asíncrona XMLHttpRequest.

Ahora tenemos el html y con él cierta metainformación de la página tanto para el control de la presentación de la página, para el indexado en buscadores como para la ubicación de todos los assets relacionados con la correcta visualización de la página solicitada.

Según el orden secuencial del código, los rectángulos rojos resaltan:

1. archivo js para el control de la analítica web

2. tipo de fuente externo (una no incluida en las fuentes de sistema)

3. estilos css

Lo que hará el navegador es entender y preparar la estructura de etiquetas de html para presentarla en el navegador y posteriormente realizar las solicitudes de assets de la web. Para simplificar la comprensión, vamos a imaginar que todos los assets se piden a la vez de forma simultánea (aunque esto es únicamente cierto en servidor con http2).

La primera solicitud es la del fichero js de analytics y contiene el tag **async** en la etiqueta. Al solicitar un fichero js, por defecto se bloqueará el hilo principal del navegador y el renderizado de la web quedará bloqueado hasta que el fichero haya llegado y se haya ejecutado.

¿Imagináis qué pasaría si por algún motivo ese fichero no existiera o se perdiera por el camino o hubiera cambiado de nombre? El navegador aparecerá en blanco y no respondería, esperando que llegue el fichero y poder ejecutarlo. Esta no es para nada una buena experiencia de usuario ni planificación del camino crítico de la página.

Para evitar esto, se pone el tag **async**, que permite que el hilo principal de renderizado del navegador no quede bloqueado y el usuario pueda tener una respuesta visual inmediata e ininterrumpida.

Históricamente no se hacía así, puesto que la propiedad **async** es propia de **HTML5**. Lo que se hacía anteriormente era incluir la solicitud del fichero al final de la etiqueta <body>, por lo que el navegador renderizaba y luego lanzaba la *request*, con lo cual el contenido ya estaba renderizado en el navegador y el usuario podía consumir e interactuar con la información. Podemos ver esto en la página de MDN para la carga de ficheros JavaScript:

The screenshot shows the MDN page for XMLHttpRequest. The top navigation bar includes links for "Peticiones asincronas", "Synchronous request", and "See also". Below the navigation, there's a note about a translation being worked on. The main content area contains two sections: one about XMLHttpRequest supporting synchronous and asynchronous requests, and another about synchronous requests blocking execution while processing the request, leading to a poor user experience. A horizontal line separates this from the "Peticiones asíncronas" section. This section has a heading and a note that it uses XMLHttpRequest asynchronously, receiving a callback when data is ready. Below this is an "Ejemplo: Enviar un archivo a la consola" (Example: Send a file to the console) section with a code snippet. To the right of the content, the browser's developer tools are visible, specifically the Elements and Styles tabs. The Elements tab shows the DOM structure of the page, including the header, main content, and footer. The Styles tab shows the CSS styles applied to various elements, such as the font-family for the body element being set to "Lato, 'Lucida Grande', Tahoma, Sans-Serif;".

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/bar/foo.txt", true);
xhr.onload = function (e) {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      console.log(xhr.responseText);
    } else {
      console.error(xhr.statusText);
    }
  }
};
xhr.onerror = function (e) {
  console.error(e.message);
};
```

En el caso de la petición de una fuente externa, podría suceder que el contenido se renderizara previamente a recibir e interpretar el archivo del tipo de fuente con el *fallback* definido en los estilos.

The screenshot shows a CSS editor with a single rule: "html { font-family: Lato, 'Lucida Grande', Tahoma, Sans-Serif; }". The editor interface includes a toolbar at the top, a code editor window with syntax highlighting, and a sidebar on the right containing style definitions and a preview pane.

Al recibir la fuente se aplicará el estilo y habrá un salto en la representación de la página ajustando los textos, el espacio y los estilos con los datos de la nueva fuente recién descargada.

De manera parecida a **async**, para <script>s de **JavaScript** tenemos preload en la segunda *request* para la fuente. Con esto le decimos al navegador que no es un asset esencial pero que lo necesitaremos para presentar correctamente el contenido. El

navegador interpreta y hace su magia con los metadatos especificados.

En el último rectángulo vemos la petición de los ficheros css con los estilos de los diferentes elementos de la página web. Este es el caso más sencillo de todos. En caso de existir y recibirlos correctamente se aplicarán; en caso contrario, veremos un error en consola que indica que no se han podido descargar los archivos y, potencialmente, veremos un renderizado que no se corresponde con el diseño que esperaba transmitir quien implementó el sitio.

Si queréis indagar un poco más en lo que sucede mientras se realizan todas estas peticiones y cómo las gestiona el navegador, podéis revisar el pestana **Performance** de las *dev tools* y poner a grabar mientras recargáis una página web:

The screenshot shows a browser window with the URL https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Usage. The page content discusses XMLHttpRequest and its support for synchronous and asynchronous requests. It includes a note about the XMLHttpRequest API being deprecated in favor of Fetch.

In the top right corner, the browser's developer tools are open, specifically the Network and Performance tabs. The Network tab displays a timeline of requests and responses. The Performance tab provides detailed metrics for the page load, including CPU usage, memory consumption, and network activity. A circular chart at the bottom of the Performance tab shows the distribution of time spent in different stages: Loading (85.5 ms), Scripting (720.0 ms), Rendering (161.7 ms), Painting (77.1 ms), Other (387.3 ms), and Idle (710.0 ms).

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/bar/foo.txt", true);
xhr.onload = function (e) {
  if (xhr.readyState === 4) {
```

3. Gestión de la asincronía

3.4. Del monolito web al **front-end** y **back-end** y SPA

Hasta aquí hemos revisado lo que sucede a modo estructural con el protocolo y el fichero html recibido, pero no a modo de aplicación.

Si nos remontamos cronológicamente a los inicios del desarrollo web, veremos que por aquel entonces teníamos páginas web estáticas formadas por varios ficheros de html, css y, en el mejor de los casos, scripts muy sencillos de **JavaScript** para control de formularios o alguna funcionalidad muy primitiva.

En esa época las páginas se implementaban una por una sobre la base de su contenido. Mantener la estructura del sitio y los estilos de forma coherente es un trabajo poco tecnificado, tedioso y potencialmente repetitivo, muy proclive a fallos, errores e inconsistencias.

Conforme la tecnología se asentaba, el tamaño de los sitios aumentaba y la necesidad de servir contenidos de manera dinámica aparecía, un nuevo paradigma surgió a partir de generar contenidos de forma dinámica en el servidor.

Lenguajes de servidor como **PHP** o **Ruby** permitían definir en el lado del servidor plantillas de html que, a partir de variables de sesión, usuario o de petición, presentaban unos datos u otros, lo que permitía el dinamismo de las páginas:

```
<title><?php print $PAGE_TITLE;?></title>

<?php if ($CURRENT_PAGE == "Index") { ?>
<meta name="description" content="" />
<meta name="keywords" content="" />
<?php } ?>

<link rel="stylesheet" type="text/css" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
<style>
#main-content {
    margin-top: 20px;
}
.footer {
    font-size: 14px;
    text-align: center;
}
</style>
```

Este código nos recuerda a html pero con variables. El servidor insertaba en los fragmentos de código `<?php ?>` las variables relativas o datos procesados de cada petición y devolvía el html con los datos solicitados. El html recibido era estático, pero su generación era dinámica.

Este paradigma es el que se conoce como **monolito** porque no existe una diferencia real entre la parte de cliente y la de servidor. Estructura html, estilos css y la gestión y el procesado de datos desde la base de datos hasta su representación conviven en un único ecosistema.

Durante este tiempo la diferenciación entre los **especialistas en presentación y manipulación de datos** fue creciendo, lo que dio lugar a perfiles **front-end** y **back-end**. Las tecnologías avanzaron en complejidad y, cada vez, resultaba más difícil seguir siendo **webmaster**, el actual **fullstack**, para acabar teniendo que tender profesionalmente a un perfil u otro.

Con el incremento de la potencia de los equipos, la parte más ligera de la manipulación de datos se desplazó paulatinamente del servidor al cliente aunque eso implicaba tener que enviar ciertos datos sensibles con los consecuentes problemas de seguridad y privacidad. Estas peticiones a demandas desde el cliente al servidor son la parte asíncrona que vamos a tratar en esta parte de la asignatura.

Esta migración de carga computacional al cliente propició la aparición del paradigma de aplicaciones **SPA** (*single page applications*). Pasamos de tener páginas estáticas generadas dinámicamente en el servidor a tener una aplicación en el cliente que realizaba peticiones a demanda contra el servidor. Esta es la situación actual, en la que la mayoría de los sitios ya no pueden considerarse páginas web como tal, sino como aplicaciones, en muchos casos habilitadas por *frameworks* como **Angular** o librerías como **React** o **Vue**.

3. Gestión de la asincronía

3.5. Interfaz y API

3.5.1. Introducción

Nosotros no vamos a llegar a desarrollar una SPA. Nos vamos a quedar en un paso anterior, con un sitio que solicite datos de forma dinámica al servidor. Primero vamos a repasar las diferentes interfaces y API de **JavaScript** y de navegador que nos hemos encontrado a lo largo del tiempo para realizar peticiones asíncronas:

1. XMLHttpRequest
2. jQuery y Ajax
3. Fetch

Nota

Vamos a enfocar tanto la teoría como la práctica exclusivamente en el consumo de datos de una API. Aunque es posible realizar otras acciones (CRUD), vamos a limitar la dificultad de las explicaciones y los ejercicios a la petición de datos.

3. Gestión de la asincronía

3.5. Interfaz y API

3.5.2. XMLHttpRequest

Inicialmente solo disponíamos de la interfaz `XMLHttpRequest` para poder realizar consultas asíncronas desde cliente. Estaba poco optimizada (no en su funcionamiento sino en su sintaxis) puesto que era poco usada, ya que no había una necesidad generalizada de este tipo de consultas al servidor y, por ello, la construcción de una petición era realmente tediosa:

```
// 1. Create a new XMLHttpRequest object
let xhr = new XMLHttpRequest();

// 2. Configure it: GET-request for the URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Send the request over the network
xhr.send();

// 4. This will be called after the response is received
xhr.onload = function() {
    if (xhr.status != 200) { // analyze HTTP status of the response
        alert(`Error ${xhr.status}: ${xhr.statusText}`); // e.g. 404: Not Found
    } else { // show the result
        alert(`Done, got ${xhr.responseText} bytes`); // responseText is the server
    }
};

xhr.onprogress = function(event) {
    if (event.lengthComputable) {
        alert(`Received ${event.loaded} of ${event.total} bytes`);
    } else {
        alert(`Received ${event.loaded} bytes`); // no Content-Length
    }
};

xhr.onerror = function() {
    alert("Request failed");
};
```

Fuente: <https://javascript.info/xmlhttprequest>

Cierto es que todo esto podría implementarse en una función y reutilizarse a conveniencia parametrizando las partes de la petición que quisiéramos hacer variables, pero no dejaba de ser un método poco optimizado y muy proclive a errores.

3. Gestión de la asincronía

3.5. Interfaz y API

3.5.3. jQuery y Ajax

De nuevo sin entrar en muchos detalles, **jQuery** es una librería que ofrecía una interfaz común para las distintas implementaciones de la especificación ECMAScript en diferentes navegadores.

Como quizás recordéis, hubo una época en la que había una guerra abierta entre navegadores, básicamente **Internet Explorer** contra **Netscape Navigator**. Su implementación del motor de JavaScript no era igual y uno, como desarrollador y usuario, podía tener una experiencia parcial o sesgada en caso de visitar un sitio con un navegador u otro.

Un grupo de brillantes y pioneros desarrolladores trabajaron en una librería llamada **jQuery**, que, al ser incluida como dependencia de un proyecto, habilitaba una capa de traducción que unificaba los accesos a las diferentes API de los navegadores eliminando la problemática de desarrollar para uno u otro.

Uno de los métodos que ofrece jQuery es **Ajax**, que permite simplificar la realización de peticiones con un código similar a este:

```
$ .ajax( {
    method: "POST",
    url: "some.php",
    data: { name: "John", location: "Boston" }
})
.done(function( msg ) {
    alert( "Data Saved: " + msg );
});
```

Fuente: <http://api.jquery.com/jquery.ajax/>

Estaremos de acuerdo en que la propuesta de **jQuery** no solo ayudaba a unificar diferentes interfaces, sino que simplificaba mucho a nivel de desarrollo la realización de peticiones.

Si visitamos el [repositorio de jQuery en Github](#), podremos ver cómo su implementación hace uso del interfaz XMLHttpRequest (extracto de código):

```
jQuery.ajaxSettings.xhr = function() {
    try {
        return new window.XMLHttpRequest();
    } catch ( e ) {}
};
```

3. Gestión de la asincronía

3.5. Interfaz y API

3.5.4. Fetch

La evolución y maduración de **JavaScript** como lenguaje, así como las continuas actualizaciones de la especificación y las progresivas implementaciones de los navegadores, hicieron más superficial la necesidad de **jQuery** como capa de traducción.

Sin embargo, muchos desarrolladores habían desarrollado dependencia a su uso por la facilidad y simplicidad de algunos de sus métodos y la resistencia a olvidar viejos hábitos.

Consejo

Evitad el uso de **jQuery**. Su propósito inicial ya no tiene razón de ser y los casos de uso justificados son escasos y muy puntuales (especialmente en aplicaciones antiguas).

La API [Fetch \(más info\)](#) es el equivalente del método **Ajax** de **jQuery** pero dentro de la especificación de los navegadores. **jQuery** (a mi modo de ver) ya no tiene un lugar en el desarrollo moderno, pero durante mucho tiempo marcó el rumbo que debería seguir **JavaScript** para alcanzar la madurez y estandarización que tiene a día de hoy.

```
// url (required), options (optional)
fetch('https://davidwalsh.name/some/url', {
  method: 'get'
}).then(function(response) {

}).catch(function(err) {
  // Error :(
});
```

Fuente: <https://davidwalsh.name/fetch>

Se debe valorar la conveniencia de esta API por encima si cabe de la propuesta por **jQuery** al no requerir la importación y el uso de una librería adicional.

Nota

Los tres métodos son válidos para usar en fase de desarrollo, pero os recomiendo encarecidamente que uséis Fetch porque es el más moderno y extendido. En caso de necesitar dar retrocompatibilidad a navegadores antiguos que no tengan incorporada esta API, podéis añadir en vuestro proyecto el [polyfill de Fetch](#) sobre el objeto *window*.

3. Gestión de la asincronía

3.6. Control de flujo en el código

3.6.1. Introducción

Estas tres maneras de realizar peticiones incorporan la gestión de la asincronía de formas diferentes:

1. `onload()` para `XMLHttpRequest`

2. `.done()` para `ajax`

3. `.then()` y `.catch()` para `Fetch`

En los dos primeros casos hablamos de **callbacks**; en el segundo, de la resolución de **promises**.

3. Gestión de la asincronía

3.6. Control de flujo en el código

3.6.2. *Callbacks*

Un *callback* es simplemente una función que se debe ejecutar una vez que se cumpla una situación o circunstancia.

En el caso de las peticiones asíncronas, el *callback* se ejecuta cuando los datos llegan devueltos desde el servidor.

Nota

En el siguiente módulo también revisaremos este concepto para los eventListeners sobre los elementos del DOM.

Cuando hablamos de *callbacks* pensamos en una función normal pero que será invocada en un momento concreto que no podemos controlar directamente; solo podemos indicar que, cuando esa situación se produzca, se ejecute dicha función.

Callback ejemplo de XMLHttpRequest:

```
xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    alert(`Received ${event.loaded} of ${event.total} bytes`);
  } else {
    alert(`Received ${event.loaded} bytes`); // no Content-Length
  }
};
```

Callback ejemplo de jQuery.ajax:

```
.done(function( msg ) {
  alert( "Data Saved: " + msg );
});
```

3. Gestión de la asincronía

3.6. Control de flujo en el código

3.6.3. Promises

Las promesas son una manera más sofisticada de implementar la gestión de la asincronía. La lógica detrás de las promesas en cuanto a especificación es altamente compleja.

Nota

Os recomiendo indagar en su conceptualización hasta donde os dé vuestro interés y capacidad en los enlaces que os dejo al final del documento, pero, por suerte o por desgracia, no necesitamos entender su implementación para poder usarlos.

Según la [definición de MDN de promise](#):

«Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca».

Esta definición es tan abstracta, genérica y sencilla que cuesta hacerse a la idea de la complejidad que lleva implícito su desarrollo en cuanto a especificación.

Aprovechando los textos encontrados en [MDN](#), podemos entender una promesa como uno de los tipos básicos de JavaScript (como Number, Array u Object), pero que almacena el estado de una petición, que puede ser:

1. **Pendiente** (*pending*): estado inicial, no cumplida o rechazada.
2. **Cumplida** (*fulfilled*): significa que la operación se completó satisfactoriamente.
3. **Rechazada** (*rejected*): significa que la operación falló.

Una promesa pendiente puede ser cumplida con un valor, o rechazada con una razón (error). Cuando una de estas dos posibilidades sucede, el método **then** o **catch** es ejecutado.

Las promesas fueron diseñadas de este modo porque el lanzar una petición no se debe bloquear el hilo de ejecución hasta que sus datos (o error) sean recibidos. Una promesa permite mantener una estructura de código y ejecución síncrona y secuencial aunque haya procesos asíncronos en marcha. Estos conceptos son los que hacen complicada la implementación desde el punto de vista del lenguaje, pero fácil su uso, ya que toda la complejidad cognitiva de lo que sucede a nivel asíncrono (y que nos resulta tan complicado de conceptualizar a los seres humanos) queda escondida detrás de la API **Fetch**.

Revisando el ejemplo anterior:

```
// url (required), options (optional)
fetch('https://davidwalsh.name/some/url', {
  method: 'get'
}).then(function(response) {

}).catch(function(err) {
  // Error :(
});
```

La gestión de la asincronía se realiza de un modo conceptualmente muy sencillo:

- al ser ejecutada la primera línea, Fetch crea una promesa en estado **pendiente**,
- cuando se resuelva esta, podrá pasar a un estado **completado** o **rechazado**,

- si se **rechaza**, se ejecutará el método catch con la función definida como *callback*, que recibirá como parámetro el error (err) encontrado,
- si se **completa**, se ejecutará el método then con la función definida como *callback*, que recibirá como parámetro la respuesta de la petición.

3. Gestión de la asincronía

3.6. Control de flujo en el código

3.6.4. Async/await

Async y await conforman un método alternativo de representar el trabajo con promesas. Hace exactamente lo mismo esto:

```
function loadJson(url) {  
    return fetch(url)  
        .then(response => {  
            if (response.status == 200) {  
                return response.json();  
            } else {  
                throw new Error(response.status);  
            }  
        })  
}
```

Que esto:

```
async function loadJson(url) { // (1)  
    let response = await fetch(url); // (2)  
  
    if (response.status == 200) {  
        let json = await response.json(); // (3)>  
        return json;  
    }  
  
    throw new Error(response.status);  
}
```

Ejemplo original de: <https://javascript.info/async-await>

Otro ejemplo para hacer más comprensible lo que sucede:

```
async function foo() {  
    console.log("hi");  
    return 1;  
}  
  
async function bar() {  
    const result = await foo();  
    console.log(result);  
}  
  
bar();  
console.log("lo");
```

Código original: <https://stackoverflow.com/a/42833744>

El proceso será:

1. creación de la promesa al ejecutar `const result = await foo();`,
2. ejecución del código secuencia del `foo()` `console.log('hi')`,
3. espera para la resolución de la promesa (aunque no hay petición asíncrona),
4. salto al hilo principal para invocar `console.log('lo')`,
5. resolución de la promesa, `result` obtiene el resultado y se invoca `console.log(result)`.

Otro ejemplo más:

```
function getUser(name) {
  fetch(`https://api.github.com/users/${name}`)
    .then(function(response) {
      return response.json();
    })
    .then(function(json) {
      console.log(json);
    });
}

//get user data
getUser('yourUsernameHere');
```

Se convierte en esto:

```
async function getUserAsync(name)
{
  let response = await fetch(`https://api.github.com/users/${name}`);
  let data = await response.json()
  return data;
}

getUserAsync('yourUsernameHere')
  .then(data => console.log(data));
```

Código original: <https://dev.to/shoupn/javascript-fetch-api-and-using-asyncawait-47mp>

Es posible que encontréis más fácil un método que otro, así que no dudéis en usarlo a conveniencia.

3. Gestión de la asincronía

3.7. Notas sobre asincronía en JavaScript

Todos los métodos que hemos visto han gestionado la asincronía con *callbacks* y promesas. Esta es la manera adecuada porque por la naturaleza de JavaScript solo existe un hilo de ejecución y todo lo que parezca simultáneo será en realidad concurrente ([más info](#)).

Puesto que solo existe un hilo de ejecución y este condiciona la presentación de datos al usuario, bloquear el hilo para realizar una petición bloquearía la interacción del usuario con nuestro sitio.

Como vimos en el primer apartado, cargar un fichero de **JavaScript** en el <head> del fichero html puede bloquear totalmente la carga de la página. Lo mismo puede suceder si una vez cargada la página realizamos una petición que bloquea el hilo principal: la página estaría renderizada pero toda interacción quedaría pendiente a que se resuelva nuestra petición.

3. Gestió de l'asincronia

3.8. Bibliografia

A continuació, us deixo la llista de recursos que heu de revisar per a la **PEC2**:

1. [YDKJS] Async & Performance

- [Introducció](#)
- [Conceptes bàsics](#)
- [Callbacks](#)
- [Promeses](#)

2. [Eloquent JS] [HTTP and Forms](#) (solament el que fa referència al protocol)

3. [Eloquent JS] [Asynchronous Programming](#)

4. [javascript.info] [The JavaScript language: Promises, async/await](#)

Vídeos:

1. [Revisiting Async – Kyle Simpson](#)

2. [Advanced Async and Concurrency Patterns in JavaScript](#)

Nota

Kyle Simpson té una opinió molt forta sobre els temes que tracta i sol estar enfrontat contra algunes decisions preses sobre l'especificació de JavaScript. Intenteu abstreure-us de tot el que sigui opinió i centreu-vos únicament en el que siguin fets i teoria objectiva.

4. Manipulación del DOM

4.1. Planteamiento

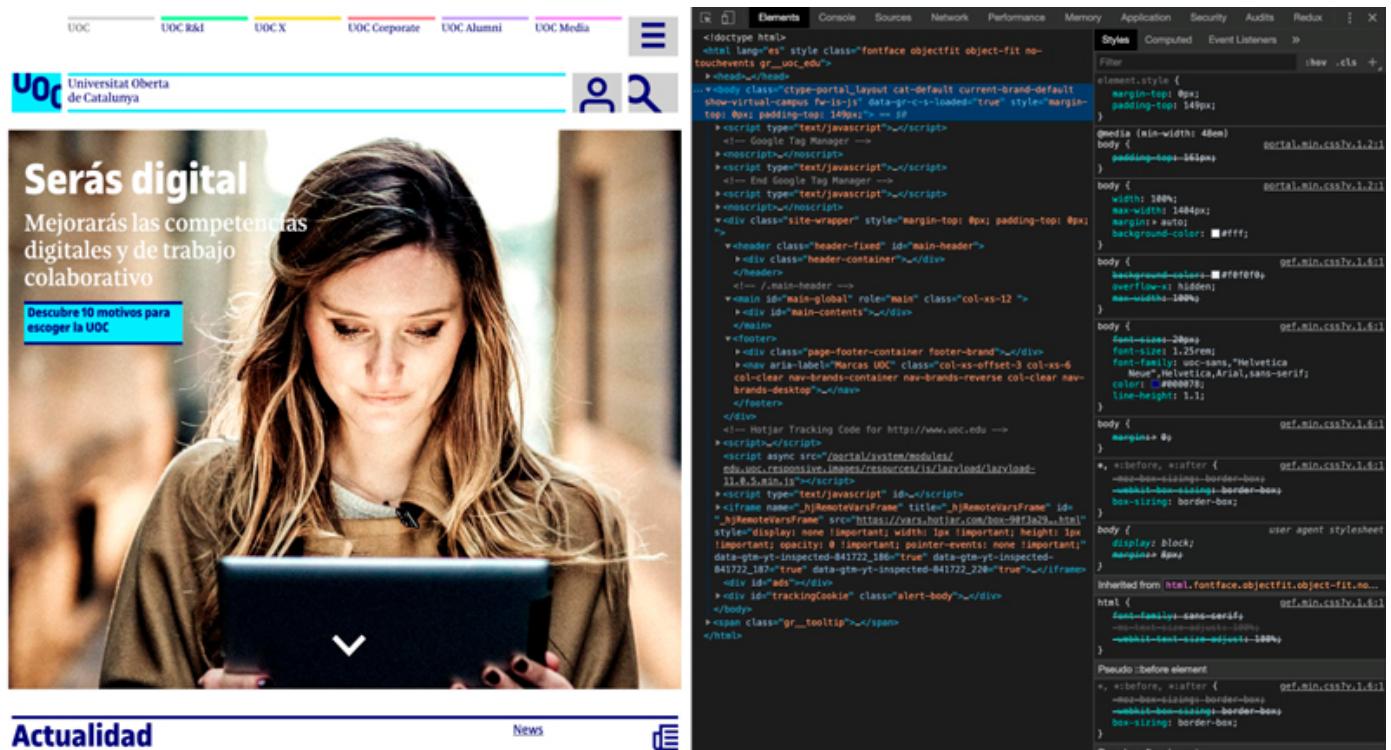
Una vez que hemos obtenido y manipulado la información que queríamos acorde a nuestro modelo de datos, llegó el momento de presentarlo de forma visual al usuario en el navegador mediante la manipulación del **DOM**.

4. Manipulación del DOM

4.2. Introducción

Como posiblemente sabéis y como habréis visto en otras asignaturas del máster, el DOM es la representación en forma de árbol del documento que renderizamos en el navegador. **DOM** son las siglas de document object model y, como decía, tiene una correlación exacta con el documento html que cargamos en el navegador.

Con una estructura clásica para una página web (html, css y js), el html define nuestro DOM. Podemos inspeccionar el aspecto que tiene nuestro DOM en las **developer tools** bajo la pestaña de **Elements**:



The screenshot shows a web browser window with the UOC website loaded. The top navigation bar includes links for UOC, UOC R&E, UOC X, UOC Corporate, UOC Alumni, and UOC Media. Below the header, there's a search bar and a user icon. The main content features a large image of a woman looking at a tablet, with text overlaying it: "Serás digital" and "Mejorarás las competencias digitales y de trabajo colaborativo". A blue button at the bottom left says "Descubre 10 motivos para escoger la UOC". On the right side of the browser window, the developer tools' Elements tab is open, displaying the HTML code for the page. The code includes standard HTML tags like <html>, <body>, <header>, <main>, <nav>, <script>, and . It also shows CSS styles and JavaScript snippets, such as Google Tag Manager scripts and a Hotjar tracking script. The code is color-coded by language (blue for HTML, red for CSS, green for JS).

Actualidad

News



Esto es ligeramente diferente con los *frameworks* y librerías que permiten tener un diseño de página tipo **SPA (single page application)**, ya que toda la generación de nodos del contenido suele hacerse en JavaScript y solo la estructura básica de documento se presenta en el html.

Para todas las interacciones con el DOM vamos a utilizar la API Document, que nos proporciona métodos y funcionalidades para realizar estas tareas.

4. Manipulación del DOM

4.3. Consultar el DOM: peticiones

Consultar el DOM implica lanzar una petición desde JavaScript a través de la API Document con los parámetros del objeto que estamos buscando. Como sabemos, hay tres características básicas que identifican un elemento html: elemento, id y clase.

```
<p id="mainHeading" class="heading">  
    This is a heading (not a header)  
</p>
```

Para obtener este elemento usaremos `document.querySelector()`, de una de estas tres maneras:

```
const el = document.querySelector("p");
```

Esta opción es la más genérica y, aunque válida, solo sería realmente útil en el caso de tener un único **elemento** del tipo párrafo en todo el documento. `querySelector()` devuelve únicamente la primera instancia recorriendo el DOM desde el elemento padre inicial, solo la primera. Esta opción es poco precisa y precisión es justo lo que necesitamos.

```
const el = document.querySelector(".header");
```

Buscar por **clase** no es incorrecto, pero de nuevo requiere tener un control extra sobre la construcción del fichero html y el DOM. Normalmente una clase se utiliza para asignar unos estilos en más de un elemento. Una clase es un tipo de estilo y su potencia radica en su reusabilidad. Precisamente por este concepto y por el hecho de poder tener elementos con la misma clase en distintos lugares del DOM no podemos ser precisos con nuestra selección de los nodos. Hablaremos de un caso de uso para selectores de clase más adelante.

```
const el = document.querySelector("#mainHeading");
```

Seleccionar por identificador es la opción deseable en todo caso y, sin duda, la más precisa. Requiere que desde el inicio del diseño tengamos claro cuáles serán nuestros contenedores o plantillas sobre las que vamos a trabajar, pero con esa planificación correctamente implementada las transiciones y manipulaciones del DOM serán altamente efectivas.

`document.querySelector()` es un método genérico que aglutina las funcionalidades de selección de varios selectores. En caso de tener claro querer seleccionar por identificador, es perfectamente válido utilizar el método `document.getElementById()`.

Un ejemplo de esto podemos encontrarlo en la [librería React](#), que requiere como paso inicial definir un contenedor en el html sobre el que pueda realizar su magia:

```
const name = 'Josh Perez';  
const element = <h1>Hello, {name}</h1>;  
  
ReactDOM.render(  
    element,  
    document.getElementById('root')  
) ;
```

Nota

No tenéis que entender todo el código, sino ver cómo el método render requiere como uno de sus parámetros el resultado de una consulta al DOM para recuperar el elemento cuya id es “root”.

También es posible realizar peticiones combinadas, pero usar este tipo de prácticas hace pensar en que la planificación del código no ha sido correcta y podría considerarse un parche para evitar atajar un problema más grave de base.

```
const el = document.querySelector("p.header#mainHeading");
```

Existen métodos adicionales dentro de la API de Document que sirven para recuperar en una única consulta todos los elementos del DOM que cumplan con el selector requerido:

1. [querySelectorAll](#)
2. [getElementsByClassName](#)
3. [getElementsByName](#)
4. [getElementsByTagName](#)
5. [getElementsByTagNameNS](#)

Estos métodos no se suelen usar por lo que os comentaba de su falta de precisión. Querer apuntar a todos los elementos de un tipo o por su clase solo puede deberse a una falta de planificación o a un uso no deseable, como puede ser cambiar los estilos de todos los elementos de compartan una clase. Todo lo referido a estilos debe solucionarse en el css y con JS lo que podemos hacer en este tipo de casos es cambiar o añadir la clase del padre (un único) elemento y que el diseño en cascada de las hojas d estilo transmita los cambios.

Desde mi punto de vista, aunque estas consultas son realizables y hay métodos de API en Document que lo permiten, eso no significa que debamos abusar de ellos.

4. Manipulación del DOM

4.4. Element, Node y NodeList

Cuando recuperamos un elemento del DOM, el tipo que asignamos es diferente a los que hemos visto hasta ahora manipulando datos en la PEC2. Al recuperar un único elemento obtendremos un [Element](#). Podéis pensar en un Element como una referencia viva al DOM (un [Node](#) del árbol del DOM) con métodos y propiedades a las que podemos acceder, que podemos modificar y que tendrán un reflejo en el documento renderizado en el navegador.

En caso de recuperar más de un nodo con los métodos de consulta múltiple, lo que obtendremos es una [NodeList](#), que conceptualmente es muy similar a una matriz de Nodes.

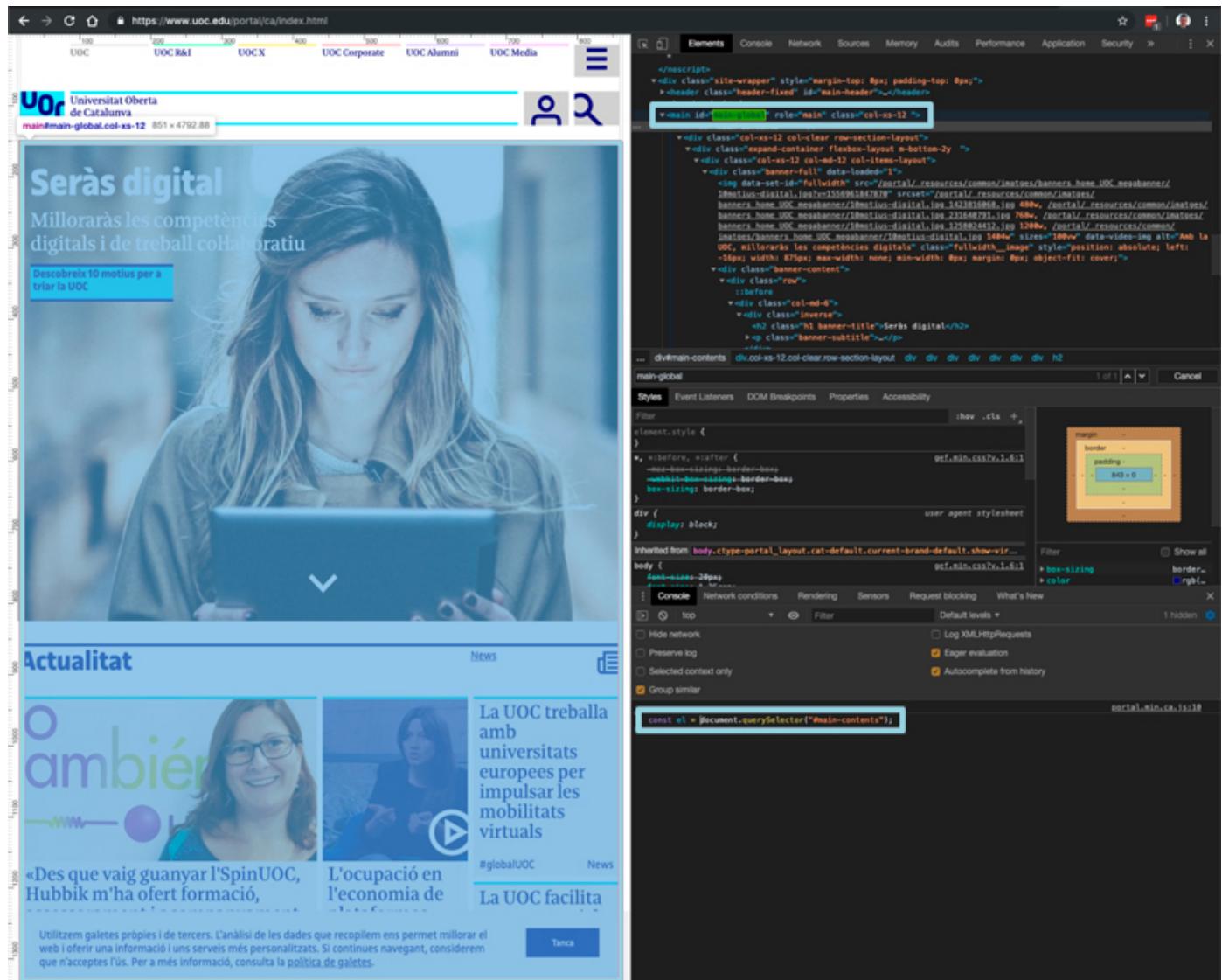
Nota

En caso de realizar una consulta al DOM con cualquiera de los métodos de la API Document y que ningún elemento cumpla con el selector, lo que obtendremos es un **null**, lo que nos servirá para realizar control de errores si así lo queremos.

4. Manipulación del DOM

4.5. Modificar el DOM: acciones

Cuando hemos recuperado un nodo podemos revisar sus propiedades o usar sus métodos.



Vamos a realizar un ejercicio guiado de ejemplo llevando a cabo las siguientes acciones simulando nuestro código JavaScript desde la consola del navegador:

1. Navegad hasta la página de la [UOC](#).
2. Abrir las *dev tools* y buscad algún elemento con id (por ejemplo, “main-contents”). Desde la consola: `const el = document.querySelector('#main-contents');`.
3. Ahora tenemos una referencia al nodo y si escribimos podremos ver un acceso rápido a todos los métodos y propiedades del elemento ([lista completa](#)).

Vamos a realizar una búsqueda sobre este nodo inicial para encontrar el título sobre la imagen «Seràs digital»; para ello, buscamos ese elemento dentro de la pestaña Elements de las *dev tools*:

The screenshot shows the UOC website's main banner. The banner features a woman looking down at a device. On the left, there is a blue header with the text 'Seràs digital'. A blue arrow points from this text to the 'h2.h1.banner-title' element highlighted in the developer tools' element inspector.

Vemos que el elemento que rodea el texto es:

```
<h2 class="h1 banner-title">Seràs digital</h2>
```

Si el diseñador de esta página hubiéramos sido nosotros y la hubiéramos implementado en función de nuestras necesidades, en esta cabecera habría un id. Como no es el caso, vamos a intentar hacer las consultas al nodo padre *el* que hemos obtenido antes para buscar este elemento.

Las *dev tools* nos ofrecen toda la ruta desde el html (cuadrado inferior derecha), lo cual facilita nuestros esfuerzos por establecer un selector correcto:

```
html body div #main-global #main-contents div div div div div.col-md-6 div.inverse h2.h1.banner-title
```

Nuestro nodo recuperado es el `#main-contents` y, viendo la estructura indicada, la opción más segura es la de anidar una selección sobre el primero y seleccionar por el elemento `h2` cuyas clases son `h1` y `banner: h2.h1.banner`.

Vemos que, incluso antes de ejecutar el comando, las *dev tools* ya nos avisan de que han encontrado ese elemento:

```
> el.querySelector("h2.h1.banner-title");
< h2.h1.banner-title
```

En este momento podríamos hacer dos cosas:

1. Asignarla a una variable como en el primer caso.
2. Realizar acciones sobre él sin asignarlo.

El primer caso es el adecuado si esperamos realizar acciones sobre este nodo más adelante. La segunda es la adecuada si solo queremos consultarla y modificarlo una vez.

Nota

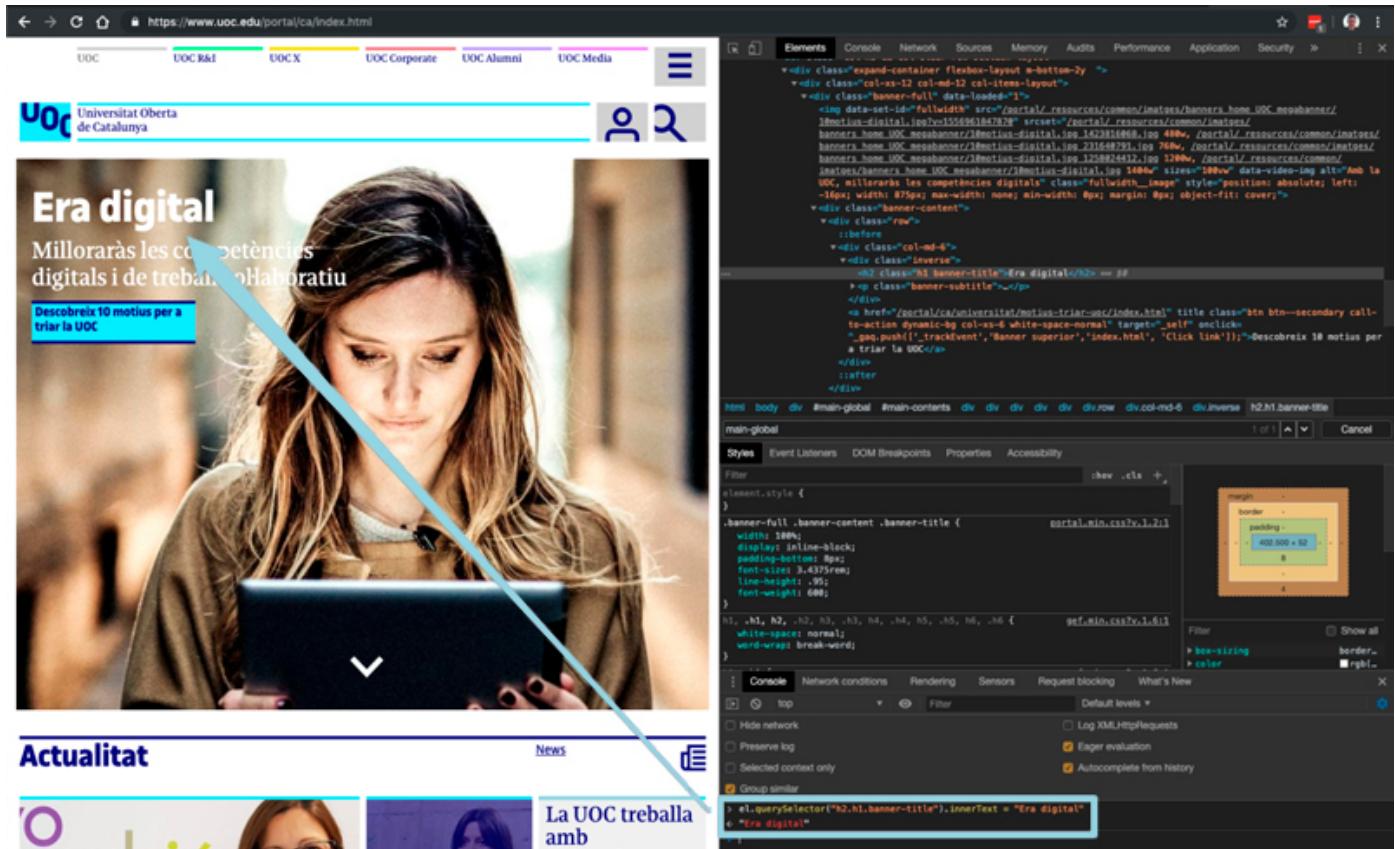
De haber querido acceder directamente a este nodo, podríamos haberlo obtenido con esta consulta:

```
const el = document.querySelector("#main-contents h1.h1.banner-title");
```

Lo más obvio en este caso es intentar modificar el contenido del texto con `innerText`:

```
> el.querySelector("h2.h1.banner-title").innerText
< "Seràs digital"
```

En este caso estamos consultando. Para redefinir el texto debemos hacer:



The screenshot shows a web browser displaying the UOC website at <https://www.uoc.edu/portal/ca/index.html>. A large banner at the top features the text "Era digital" and "Milloraràs les competències digitals i de treball a l'hàbitat". Below the banner, there is a call-to-action button with the text "Descobreix 10 motius per a triar la UOC". The browser's developer tools are open, specifically the Elements tab, which shows the HTML structure of the banner. An arrow points from the "Era digital" text in the banner to the corresponding line of code in the developer tools, which is highlighted with a green background. The code snippet is as follows:

```
> el.querySelector("h2.h1.banner-title").innerText = "Era digital"
< "Era digital"
```

Nota

Las operaciones sobre el DOM suelen ser caras a nivel de rendimiento. Modificar el DOM, su estructura y su contenido implica efectos colaterales de repintado (si no se modifican las dimensiones de los elementos, como cambio de colores) y de re-renderizado si las dimensiones de algún elemento han cambiado (*width*, *height*, cambio de texto...). Esto no implica que no deban hacerse, pero sí que deben hacerse siempre los mínimos posibles desde un punto de vista de rendimiento web.

En lo referente a nodos existentes, las acciones más habituales son las de consulta de las clases de un elemento y añadir o quitar clases en consecuencia de lo primero, o las de modificar textos.

Cabe también la posibilidad de que queramos generar nuevos elementos no existentes en el html. Para ello, utilizaremos un nuevo método de la API Document: [createElement](#). Este método nos permite replicar la construcción de elementos en JavaScript para ser insertados en el DOM una vez listos:

```
function addElement () {
  var newDiv = document.createElement("div");
  var newContent = document.createTextNode("Hola! ¿Qué tal?");
  newDiv.appendChild(newContent);

  var currentDiv = document.getElementById("div1");
  document.body.insertBefore(newDiv, currentDiv);
}
```

Fuente: [MDN](#)

1. [createElement](#): crea un nodo del tipo indicado.

2. `createTextNode`: crea un nodo de texto con el texto indicado.
3. `appendChild(newContent)`: añade el nodo `newContent` al nodo `newDiv`.
4. `getElementById(«div1»)`: obtiene del DOM el nodo con id “div1”.
5. `body.insertBefore(newDiv, currentDiv)`: añade el creado antes de “div1”.

Hay más métodos para trabajar en esta dirección con el DOM:

- `removeChild`
- `replaceChild`

En páginas tradicionales (no SPA) es más habitual modificar contenido existente. Puesto que diseñamos nuestro html como primer paso de nuestro desarrollo, parece complicado que algo que quisiéramos mostrar no esté ya en pantalla. El dinamismo del contenido se consigue con consultas asíncronas a API, interacción de usuario, modificación del contenido y aspecto de nodos, como hemos visto hasta ahora.

En el caso de páginas SPA (*angular, react, vue*) suele ser habitual que, excepto el contenedor base, todos los demás nodos se creen en JavaScript. Estas librerías y frameworks proveen un sistema de trabajo sobre la API Document como JSX en el caso de React para facilitar la creación y modificación de los nodos.

4. Manipulación del DOM

4.6. Añadir «escuchadores» de eventos

Gracias a los escuchadores de eventos (`eventListeners`) podemos adjuntar acciones y comportamiento a elementos del DOM para permitir al usuario manipular y obtener los datos que deseé.

La sintaxis de creación de un eventListener es:

```
const el = document.getElementById("main-contents");
el.addEventListener("click", miFuncion, false)
```

1. Obtenemos el nodo sobre el que queremos adjuntar el escuchador.

2. Añadimos el evento definiendo el tipo (`click`), la función que se debe ejecutar cuando el evento suceda (`miFuncion`) y el `useCapture` (`flag` opcional para indicar cómo se debe capturar el evento en el DOM).

Veamos un ejemplo:

<https://codepen.io/davidlampon/pen/EzYdgN>

```
const mainNode = document.querySelector('#main');
const clickMeButton = document.querySelector('#clickMeButton');

function changeBackgroundColor() {
  mainNode.classList.toggle('clicked');
}

clickMeButton.addEventListener('click', changeBackgroundColor)
```

Revisando solo el JavaScript:

1. Obtenemos el nodo con id `main`.
2. Obtenemos el nodo con id `clickMeButton`.
3. Definimos una función que revisa las clases del elemento `main` y si tiene la clase `clicked`, se la quita y si no, se la pone.
4. Añadimos en `eventListener` a `clickMeButton` para que ejecute la función `changeBackgroundColor` cada vez que se haga clic sobre él.

De este modo podemos hacer que nuestro usuario modifique el comportamiento de la página web que esté visitando. En este ejemplo ha sido un cambio visual, pero lo más habitual es un cambio de visibilidad o solicitud de nuevos contenidos.

Notas

1. Los eventos se adjuntan a elementos del DOM existentes. En caso de intentar añadir un escuchador a un nodo no existente (o que se cree posteriormente), el comportamiento no existirá.
2. En caso de eliminar un elemento del DOM, es adecuado eliminar el `eventListener` antes de eliminar el nodo.
3. Es importante entender cómo se propagan y cómo se detienen los eventos en el navegador (podéis consultar los recursos al final de este documento).

4. Manipulación del DOM

4.7. Bibliografía

A continuación os dejo la lista de recursos que debéis revisar para la **PEC3**:

1. [MDN] [Propagación de eventos](#)
2. [MDN] [API Document](#)
3. [MDN] [API Element](#)
4. [MDN] [Tipos de eventos para eventListeners](#)
5. [Eloquent JS]
 - [JavaScript and the Browser](#)
 - [The Document Object Model](#)
 - [Handling events](#)

5. Diseño de API en Node

5.1. Presentación

En esta práctica veremos conceptos teóricos y una implementación a alto nivel de una API en Node. Para ello, seguiremos el siguiente ejercicio desarrollado por [Scott Moss](#), CEO de [Tipe.io](#).

5. Diseño de API en Node

5.2. Introducción

5.2.1. Qué es una API

Una API (*application programming interface*) es un servidor que crea un interfaz HTTP para conectar aplicaciones externas u otros servidores con algún tipo de información a los que tiene acceso de manera directa, normalmente en forma de base de datos. Se puede considerar como esa parte del engranaje que hace posible que nuestras aplicaciones de navegador consigan datos sobre los que trabajar.

Las operaciones básicas (en relación con los datos) que definen el comportamiento de una API son las conocidas como CRUD:

1. *create* (crear)
2. *read* (leer)
3. *update* (actualizar)
4. *delete* (borrar)

Entre las diversas implementaciones de estas cuatro funcionalidades/verbos/operaciones, la más habitual y extendida es la que se conoce como REST (*representational state transfer*). En este diseño se combinan recursos de bases de datos, rutas y verbos HTTP (CRUD) para permitir que las aplicaciones puedan describir qué acción quieren realizar.

La adopción de REST como patrón de diseño se hizo popular cuando las empresas con servicios tipo SaaS empezaron a abrir sus fuentes de información para el uso de terceros, puesto que es fácil de usar y optimizar para estructuras básicas de datos.

Se trata del patrón de diseño más popular para diseñar API, aunque no por ello sea perfecto. En lo que se refiere a diseño de API, todas las soluciones tienen ventajas y desventajas, por lo que es un tema situacional y específico de cada servicio y estructura de datos establecer cuál es la solución más adecuada para cada caso.

5. Diseño de API en Node

5.2. Introducción

5.2.2. Qué es Node

Como ya hemos visto a lo largo del curso, Node es lo que resumidamente conocemos como JavaScript en el servidor. Node permite trasladar la potencia, las virtudes (y los defectos) del lenguaje más allá de del navegador y cliente.

Debido a las características intrínsecas de JavaScript:

- hilo único de ejecución
- asíncrono
- *event driven*

Resulta una buena elección cuando se diseñan servicios:

1. de alta concurrencia
2. no exigentes en cuanto a capacidad computacional (ni ML ni IA)

Además, gracias a la popularidad de JavaScript, cuenta con una gran comunidad, aceptación e infinidad de herramientas *open source* para facilitar la implementación de API, una de las cuales es Express, que usaremos en este ejercicio.

5. Diseño de API en Node

5.2. Introducción

5.2.3. Qué es Express

Express es el *framework de facto* para crear API en Node. Lo que Express facilita es la integración con la capa de sistema e infraestructura. Todo lo que hace podemos implementarlo nosotros mismos, pero se trata de un cúmulo de tareas que podrían considerarse de otra capa de desarrollo diferente, como por ejemplo:

- abrir y cerrar *sockets* y puertos
- comprobación de rutas
- gestión de errores

Y que no es necesario crear y definir de nuevo en cada nueva API. Trabajar con Express significa usar una herramienta *open source* con un gran apoyo por parte de la comunidad. No parece factible que llegue un competidor que vaya a desterrarlo, por lo que, a día hoy, es la elección obvia para desarrollar y trabajar la infraestructura de la API.

5. Diseño de API en Node

5.2. Introducción

5.2.4. Qué es MongoDB

MongoDB es la base de datos no relacional por defecto que, por su naturaleza, funciona genial con Node (JavaScript). Se trata de una store de almacenaje de documentos no relacionales (similar a JSON) que es fácil de instalar; además, es sencillo empezar a trabajar con ella y, por si fuera poco, escala bien junto con el resto de la aplicación y el número de usuarios.

De nuevo se trata de un sistema de gestión *open source* y está respaldado por una gran empresa (la propia MongoDB Inc.). Existen multitud de soluciones de hosting que ofrecen la posibilidad de proveer MongoDB.

El puente entre la base de datos no relacional MongoDB y nuestro código de API en Node (JavaScript) sobre Express se tiende mediante una herramienta conocida con el nombre de Mongoose. Se trata de un ODM (*object data modeling*), un envoltorio alrededor de los datos almacenados, que nos permite interactuar más fácilmente con la información.

5. Diseño de API en Node

5.3. Metodología del ejercicio

5.3.1. Introducción

Antes de empezar a trabajar, recomiendo hacer un fork del proyecto y trabajar sobre él, en lugar de clonar directamente el repositorio original. Esto nos permite subir cambios en caso de así quererlo y hacerlos permanentes en nuestro repositorio personal o incluso proponer *pull requests*, aunque eso está lejos del propósito de nuestro ejercicio.

Nota

Este es el modo de trabajar en *open source* y de presentar solución a incidencias (*issues*) o nuevas funcionalidades (*features*) a repositorios de herramientas o productos reales.

El propósito general de este ejercicio es implementar toda una app, una aplicación que dé forma a una lista de cosas pendientes de hacer. Las entidades en las que deberemos pensar durante el planteamiento y la implementación son las siguientes:

1. item: cada una de las «cosas pendientes de hacer»
2. lista: compendio de items
3. usuario: cada uno de los propietarios de una lista

5. Diseño de API en Node

5.3. Metodología del ejercicio

5.3.2. Requisitos

Los requisitos de nuestro sistema son (seguid los enlaces para instalar):

1. [Node v6.0 o superior](#) (comprobar con node -v)
2. [yarn](#) (alternativa a npm)
3. [MongoDB](#)
4. [Nodemon](#) (utilidad para reiniciar el servidor automáticamente)
5. [Insomnia](#) (cliente REST para trabajar contra la API)

Tras asegurarnos de que todas estas herramientas están instaladas en nuestro equipo, procederemos a clonar el repositorio en local, en nuestra carpeta de proyectos. En mi caso, después de haber hecho el *fork* del original, el comando que usaré es:

```
git clone git@github.com:davidlampon/api-design-node-v3.git
```

5. Diseño de API en Node

5.3. Metodología del ejercicio

5.3.3. Dependencias del proyecto

Una vez que el proyecto ya está en nuestra máquina, abrimos la carpeta de proyecto en nuestro editor y repasamos la estructura de las dependencias en el **package.json**:

```
"dependencies": {  
    "bcrypt": "^3.0.2",  
    "body-parser": "^1.18.3",  
    "cors": "^2.8.5",  
    "cuid": "^2.1.4",  
    "dotenv": "^6.1.0",  
    "express": "^4.16.4",  
    "jsonwebtoken": "^8.4.0",  
    "lodash": "^4.17.11",  
    "mongoose": "^5.3.13",  
    "morgan": "^1.9.1",  
    "validator": "^10.9.0"  
},
```

El desempeño de cada uno de ellos (seguid el enlace para obtener la descripción) es:

1. [bcrypt](#) > *hashing* de contraseñas
2. [body-parser](#) > *middleware* para parsear el *body* de la *request*
3. [cors](#) > *middleware* para las políticas de comunicación del servidor
4. [cuid](#) > generador de id únicas
5. [dotenv](#) > carga de variables de ficheros .nv
6. [express](#) > el *framework* de creación de API
7. [jsonwebtoken](#) > implementación del protocolo JWT
8. [lodash](#) > librería de utilidades
9. [mongoose](#) > *wrapper* de datos para MongoDB
10. [morgan](#) > herramienta de *logging* para el servidor
11. [validator](#) > validador de *strings*

5. Diseño de API en Node

5.3. Metodología del ejercicio

5.3.4. Comandos

Del mismo modo repasamos los comandos que utilizaremos durante el desarrollo:

```
"scripts": {  
  "build": "babel src --out-dir dist",  
  "test": "NODE_ENV=testing jest --forceExit --detectOpenHandles --silent",  
  "test-routes": "yarn test -t router",  
  "test-models": "yarn test -t model",  
  "test-controllers": "yarn test -t controllers",  
  "test-auth": "yarn test -t Authentication:",  
  "dev": "nodemon --exec yarn restart",  
  "restart": "rimraf dist && yarn build && yarn start",  
  "start": "node dist/index.js"  
},
```

1. build > transpila de ES6 a ES5 en la carpeta /dist
2. test > ejecuta el resto de los comandos de test
3. test-routes > ejecuta los test de las rutas
4. test-models > ejecuta los test de los modelos
5. test-controllers > ejecuta los test de los controladores
6. test-auth > ejecuta los test de la autenticación
7. dev > arranca nodemon para reiniciar el servidor en cada modificación de código
8. restart > el comando usado para arrancar de nuevo en dev
9. start > el comando usado en restart para arrancar el servidor

5. Diseño de API en Node

5.3. Metodología del ejercicio

5.3.5. Elementos adicionales

Hay unas configuraciones existentes de **eslint** y **prettier** que, con los *plugins* adecuados instalados en vuestro editor, permiten el autoestilado del código.

Dentro de la carpeta **/src** podemos encontrar:

1. **/config** > los secretos y las configuraciones del servidor
2. **/resources** > nuestra carpeta de trabajo
3. **/utils** > funciones auxiliares
4. **index.js** > punto de entrada de la aplicación
5. **server.js** > arranque y configuración del servidor

Una vez que tenemos una vista global de la estructura y el funcionamiento a alto nivel de la aplicación, ya podemos pasar a instalar las dependencias con:

```
npm install
```

5. Diseño de API en Node

5.4. Express

5.4.1. Introducción

Rama asociada: [link](#)

```
git checkout lesson-1
```

En este ejercicio:

- Crearemos una API sencilla basada en Express.
- Crearemos una ruta que devuelve un json.
- Crearemos una ruta que acepte json y lo enseñe por consola.
- Arrancaremos el servidor.

Arrancamos con este código en el fichero **server.js**:

```
import express from 'express'
import { json, urlencoded } from 'body-parser'
import morgan from 'morgan'
import cors from 'cors'

export const app = express()

app.disable('x-powered-by')

app.use(cors())
app.use(json())
app.use(urlencoded({ extended: true }))
app.use(morgan('dev'))

export const start = () => {}
```

Nuestra variable de trabajo es la que definimos como **app**, que podríais nombrar como **server** si eso os da una idea más clara de lo que sucede. En ella tenemos una instancia de Express que pasamos a configurar con los *middlewares* **cors**, **json**, **urlencoded** y **morgan**. Estos *middlewares* se ejecutarán de manera ordenada del primero al último procesando la petición recibida y pasando modificaciones al siguiente en caso de ser necesario, pero en ningún caso serán los encargados de implementar y enviar la respuesta al usuario.

Al final del fichero vemos que exportamos la función **start**, que será importada en **index.js** y será a la que llamaremos para arrancar el servidor. Sin embargo, ahora mismo esa función está vacía y debemos rellenarla con el código necesario:

En la cabecera:

```
import { connect } from './utils/db'
```

Con esto importamos la función para conectar con la base de datos local de MongoDB.

En la función start:

```
export const start = async () => {
  try {
    await connect()
    app.listen(config.port, () => {
      console.log(`REST API on http://localhost:${config.port}/api`)
    })
  } catch (e) {
    console.error(e)
  }
}
```

Conectamos con la base de datos y abrimos el servidor con **app.listen** en el puerto configurado en el fichero **config.js**, en nuestro caso: **3000**.

Recordad que para que vuestro servidor de MongoDB sea accesible lo debéis arrancar mediante comando o como servicio.

Si ahora ejecutamos:

```
npm run dev
```

Veremos este mensaje en consola:

```
> api-design-v3@1.0.0 dev /Users/davidlampon/Projects/api-design-node-v3
> nodemon --exec yarn restart

[nodemon] 1.18.9
[nodemon] reading config ./nodemon.json
[nodemon] to restart at any time, enter `rs`
[nodemon] or send SIGHUP to 99250 to restart
[nodemon] ignoring: ./git/**/* node_modules/**/node_modules ./dist/**/*
[nodemon] watching: src/**/*.js src/**/*.graphql src/**/*.gql
[nodemon] watching extensions: js,json,graphql
[nodemon] starting `yarn restart`
[nodemon] spawning
[nodemon] child pid: 99252
[nodemon] watching 24 files
yarn run v1.16.0
$ rimraf dist && yarn build && yarn start
$ babel src --out-dir dist
Successfully compiled 24 files with Babel.
$ node dist/index.js
(node:99261) DeprecationWarning: collection.ensureIndex is deprecated. Use createIndexes instead.
REST API on http://localhost:3000/api
```

Lo que nos indica que la API ya es accesible en <http://localhost:3000/api>. Sin embargo, puesto que no hemos definido ningún tipo de verbo para ninguna ruta, si intentamos acceder desde el navegador a esa dirección recibiremos un error 401 y en consola veremos:

```
GET /api 401 1.990 ms - -
```

5. Diseño de API en Node

5.4. Express

5.4.2. Routing

Rama asociada: [link](#).

En este ejercicio:

- Crearemos un *router* para el recurso item.
- Crearemos todas las rutas CRUD y las plantillas de los controladores.
- Montaremos el *router* en el servidor.
- Pasaremos todos los test mediante el comando `npm run test-routes`.

Vamos a implementar el primer par verbo-ruta. Añadimos este código después de los *middlewares* y antes de la definición de la función start:

```
app.get('/data', (req, res) => {
  res.json({ message: 'hello' })
})
```

Con esto le decimos a nuestra instancia de Express, **app**, cómo actuar cuando reciba una petición **get** (las más habituales) en la ruta **/data** definiendo el *callback*. En este *callback* Express nos facilitará dos parámetros por defecto:

1. **req**: los datos de la petición recibida (*request*)

2. **res**: el objeto plantilla de respuesta (*response*)

Si ahora accedemos a <http://localhost:3000/api>, seguiremos teniendo un error, pero si en su lugar vamos a la ruta que hemos definido en nuestro **server.js** <http://localhost:3000/data>, obtendremos en el navegador el json:

```
{ message: 'hello' }
```

Y por consola del servidor el mensaje:

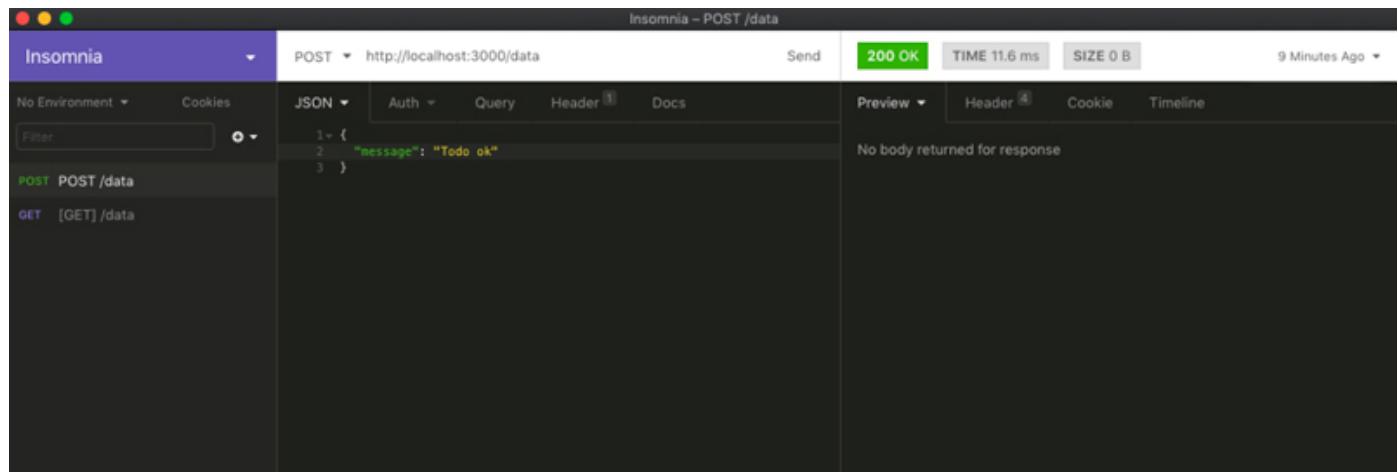
```
GET /data 200 5.932 ms - 19
```

Realizamos una nueva combinación de verbo ruta con el siguiente código (añadir justo después del *get* previo):

```
app.post('/data', (req, res) => {
  console.log(req.body)
  res.status(200).end()
})
```

Este caso es similar al anterior, solo que cambiamos **get** por **post**, que significa que recibimos datos en lugar de proveerlos y devolvemos un estado 200 como respuesta, que indica que todo ha ido bien.

Probar peticiones post es complicado en el navegador, por lo que utilizaremos la aplicación [Insomnia](#) para verificar que todo funciona bien. Elegimos **New request** y rellenamos los campos con la descripción que creamos conveniente. Rellenamos la url de nuestro servidor local y definimos el verbo (haremos uno para get y otro para post) del siguiente modo:



The screenshot shows the Insomnia REST client interface. On the left, there's a sidebar with environment dropdowns, a filter input, and two requests listed: "POST POST /data" and "GET [GET] /data". The main area has tabs for "JSON", "Auth", "Query", "Header", and "Docs". A code editor shows the JSON payload: "1: { 2: \"message\": \"Todo ok\" 3: }". The top right shows the response details: "200 OK", "TIME 11.6 ms", "SIZE 0 B", and "9 Minutes Ago". Below the status, it says "No body returned for response".

Y al ejecutar esta consulta post con nuestro objeto podemos ver en consola lo siguiente:

```
{ message: 'ok' }
POST /data 200 1.937 ms - -
```

Tanto el **app.get** como el **app.post** que hemos implementado se conocen con el nombre de controladores. Un controlador es exactamente lo mismo que un *middleware*, tanto en uso como en implementación; la única diferencia reside en que el *middleware* recibe un parámetro extra llamado `next` que debe ejecutarse para pasar al siguiente *middleware/controlador* y los controladores son el último paso de proceso de una petición, por lo que no tienen `next` y en su lugar emiten una respuesta de vuelta al usuario.

En nuestro caso, los *middleware* se ejecutan para todas las rutas sin excepción, pero podría limitarse su uso de igual modo que hemos hecho con los controladores para rutas concretas.

Podemos ver un ejemplo de *middleware* en una función que simplemente emita un mensaje por la consola del servidor en cada petición recibida.

```
const log = (req, res, next) => {
  console.log('logging');
  next();
}
```

Y lo añadiremos al final de la lista de controladores si queremos que se ejecute en todas las rutas del siguiente modo:

```
app.use(log());
```

O si queremos que solo se ejecute en rutas específicas del siguiente:

```
app.get('/data', log, (req, res) => {
  res.json({ message: 'hello' })
})
```

De las diversas maneras de diseñar rutas a partir de los verbos CRUD, las más usadas son las de *matching exacto* de ruta (las que hemos hecho hasta ahora) y las de *matching* de parámetro. Estas son las que definen el modelo REST y sobre las que se diseña Express, por lo que tiene todo lo que necesitamos para implementar nuestra API.

| Inicial | Significado | Verbo HTTP | Método Express |
|---------|-------------|------------|----------------|
| C | Create | POST | app.post() |
| R | Read | GET | app.get() |
| U | Update | PUT | app.put() |
| D | Delete | DELETE | app.delete() |

Como se mencionó anteriormente, el orden en el que se definen los *middleware* en el código es importante, ya que la petición pasará de uno a otro en serie, no en paralelo. Del mismo modo, el orden de definición de las rutas es importante, ya que la primera que coincide será la que se ejecutará, por lo que las más concretas deberán definirse antes que las más genéricas.

En la definición que hemos hecho de las dos rutas anteriormente las hemos añadido sobre el objeto **app** directamente. Express ofrece la posibilidad de definir un objeto **router** como abstracción y aplicarlo sobre la instancia de Express (app).

```
const router = express.Router();

router.get('/data', (req, res) => {
  res.send({ message: 'hello' });
}

app.use('/api', router);
```

Nota

En este caso las rutas que aceptaríamos no son directamente /data sino /api/data, puesto que la construcción de la ruta en app y router así la define.

Los casos en los que querríamos hacer routers específicos se limitan a aquellas aplicaciones o partes de ellas que se comportan de manera muy diferente al resto y las reglas globales no aplican.

Supongamos una API que sirve *gatos* (*cats*), las combinaciones de rutas y verbos de CRUD/REST serían:

| Verbo | Ruta | Significado |
|--------|----------|---------------------------------------|
| GET | /cat | Ler varios gatos (ruta) |
| GET | /cat/:id | Ler un gato (ruta + parámetro) |
| POST | /cat | Crear un gato (ruta) |
| PUT | /cat/:id | Actualizar un gato (ruta + parámetro) |
| DELETE | /cat/:id | Borrar un gato (ruta + parámetro) |

Y esto que acabamos de ver es prácticamente igual en cualquier API que se haya diseñado bajo el patrón REST. Gracias a Express, podemos hacer simplificaciones de código y en lugar de escribir cada par verbo-ruta podemos hacer las definiciones para los *match* de ruta y los *match* de parámetro del siguiente modo:

```

const router = express.Router();

router
  .route('/cat')
  .get(controller)
  .post(controller);

router
  .route('/cat/:id')
  .get(controller)
  .put(controller)
  .delete(controller);

```

Nota

A partir de este punto de la práctica trabajaremos el código en la dirección de hacer cumplir los test a modo de comprobación de que los conceptos y la implementación son correctos. Sin embargo, en cualquier momento, se puede volver a usar la aplicación **Insomnia** para valorar cuál es el resultado de enviar una consulta a nuestra API.

Por último, vamos a definir una serie de funciones plantilla como controladores. En este preciso momento son solo funciones que al ejecutarlas no devuelven nada, pero nos servirán para establecer la base de código sobre la que trabajaremos en las siguientes partes del ejercicio. En el fichero item.controllers.js añadimos el siguiente código:

```

export const getOne = model => async (req, res) => {}

export const getMany = model => async (req, res) => {}

export const createOne = model => async (req, res) => {}

export const updateOne = model => async (req, res) => {}

export const removeOne = model => async (req, res) => {}

export const crudControllers = model => ({
  removeOne: removeOne(model),
  updateOne: updateOne(model),
  getMany: getMany(model),
  getOne: getOne(model),
  createOne: createOne(model)
})

```

Ahora, al ejecutar los test con `npm run test-routes` obtendremos el siguiente mensaje por consola:

```

PASS src/resources/item/__tests__/item.router.spec.js
  item router
    ✓ has crud routes (243ms)

Test Suites: 1 passed, 1 total

```

```
Tests:      1 passed, 1 total
```

```
Snapshots: 0 total
```

```
Time:      1.613s
```

5. Diseño de API en Node

5.5. Modelo de datos con MongoDB, schemas

Rama asociada: [link](#).

En este ejercicio:

- Crearemos un *router* para el recurso ítem.
- Crearemos todas las rutas CRUD y las plantillas de los controladores.
- Montaremos el *router* en el servidor.
- Pasaremos todos los test mediante el comando `npm run test-models`.

A pesar de que MongoDB sea una base de datos no relacional, es muy recomendable utilizar algún tipo de estructura de datos, los **schemas**. MongoDB proporciona herramientas para definirlos, pero **Mongoose** como herramienta es más adecuada y más ampliamente aceptada. Esta validación es necesaria antes de que ningún dato entre en la base de datos y asegurar así que pueda ser correctamente consultada y tratada posteriormente.

Podemos crear los modelos para cada recurso REST que queramos exponer mediante la API, que se crean a partir de las instrucciones definidas en el schema mediante Mongoose.

El proceso de implementación es como sigue:

1. Los datos se diseñan sobre schemas.
2. Los schemas se convierten en modelos.
3. Los controladores usan los modelos.
4. Los controladores son activan por los recursos.

Nota

Los modelos de lista y usuario ya están definidos, por lo que no deberemos preocuparnos de su implementación; sin embargo, revisarlos nos puede dar pistas sobre nuestra tarea para este ejercicio.

Empezaremos ejecutando los test relativos a los modelos:

```
npm run test-models
```

Veremos que todos ellos fallan porque aún no hemos implementado nada:

```
FAIL src/resources/item/__tests__/item.model.spec.js
  Item model
    schema
      X name (262ms)
      X status (26ms)
      X notes (28ms)
      X due (20ms)
      X createdBy (23ms)
      X list (28ms)
```

Mirando las respuestas de los test podemos ver exactamente cuál es la definición esperada para los modelos. Por ejemplo, para el campo **name** obtenemos:

```
Item model > schema > name

expect(received).toEqual(expected)

Expected value to equal:
  {"maxlength": 50, "required": true, "trim": true, "type": [Function String]}
Received:
  undefined
```

Encontramos mensajes similares para status, notes, due, createdBy y list.

Sabiendo esto debemos ir a **resources > item > item.model.js** e implementar el modelo.

Esto es lo que nos encontramos en el fichero como plantilla:

```
import mongoose from 'mongoose'

const itemSchema = new mongoose.Schema({}, { timestamps: true })
export const Item = mongoose.model('item', itemSchema)
```

Y deberemos recrear la estructura que nos solicita el test:

```
const itemSchema = new mongoose.Schema(
{
  name: {
    type: String,
    required: true,
    trim: true,
    maxlength: 50
  },
  status: {
    type: String,
    required: true,
    enum: ['active', 'complete', 'pastdue'],
  },
  notes: String,
  due: Date,
  createdBy: {
    type: mongoose.SchemaTypes.ObjectId,
    ref: 'user',
    required: true
  },
  list: {
    type: mongoose.SchemaTypes.ObjectId,
    ref: 'list',
    required: true
  }
})
```

```
        }
    },
    { timestamps: true }
)
```

Adicionalmente para asegurarnos de que los índices en la base de datos son correctos, deberemos añadir esta línea antes del `export`. Esto nos permite mantener la coherencia de datos dentro de una base no relacional:

```
itemSchema.index({ list: 1, name: 1 }, { unique: true })
```

Los valores 1 de `list` y `name` se refieren al tipo de ordenación y con ello evitamos que se pueda añadir en una misma lista un nombre duplicado, aunque permitimos que existan nombres duplicados siempre que existan en diferentes listas.

Con este código hemos definido la estructura de datos que deberá tener cualquier información recibida para poder ser incluida en la base de datos. Con esto aseguramos que ninguna estructura que no conocemos de antemano y se ajuste a nuestra plantilla llegará a ser parte de esta. Este punto resulta básico no solo para insertar datos, sino para poder tratarlos, manipularlos y servirlos posteriormente.

Si volvemos a ejecutar los test relativos a los modelos:

```
npm run test-models
```

Obtendremos el siguiente mensaje por consola:

```
PASS src/resources/item/__tests__/item.model.spec.js
  Item model
    schema
      ✓ name (285ms)
      ✓ status (41ms)
      ✓ notes (40ms)
      ✓ due (22ms)
      ✓ createdBy (21ms)
      ✓ list (23ms)

  Test Suites: 1 passed, 1 total
  Tests:       6 passed, 6 total
  Snapshots:   0 total
  Time:        1.283s, estimated 2s
```

5. Diseño de API en Node

5.6. Controladores

Rama asociada: [link](#).

En este ejercicio:

- Crearemos los *resolvers* CRUD en **utils/crud.js**.
- Crearemos los controladores para el recurso **item** usando los *resolvers*.
- Pasaremos todos los test mediante el comando `npm run test-controllers`.

Los controladores implementan la lógica que interactúa con nuestro modelo de la base de datos desde una ruta y mediante un verbo.

Si pensamos en ellos como un *middleware* más, son el último de la cadena, ya que no ejecutan la función `next()` (tercer parámetro del *callback*), por lo que no continúan hacia otro *middleware* y son los responsables de enviar una respuesta de vuelta.

Los formatos de respuesta en un controlador pueden adoptar varias formas:

```
res.status(404).end()
```

En este caso fijamos el código de respuesta como un error 404 mediante `.status(404)` y damos por finalizada la manipulación de la petición mediante el método `.end()`. En este caso no estamos definiendo el *body* de la respuesta, por lo que, por defecto, se devolverá lo mismo que se envió en la petición `req.body`. Si, por contra, queremos devolver un mensaje concreto, podríamos hacerlo de este modo:

```
res.status(404).json({ message: 'not found' }).end()
```

Con el método `.json()` nos aseguramos de que el *body* de la respuesta se envíe en este formato aunque podríamos usar `.send()` si configuramos por defecto que la respuesta sea en formato JSON.

En ambos casos se puede añadir un *return* al inicio del comando, pero, aunque no se incluya, al hacer un `.end()` el código no se ejecutaría tras esta sentencia. La respuesta se cierra y se envía de vuelta.

Los controladores plantilla que creamos en los apartados anteriores funcionarán independientemente del modelo. La potencia radica en que la acción es la misma y solo el modelo de datos cambia.

Mongoose proporciona métodos sobre los modelos:

| Verbo | Métodos |
|-------|--|
| C | <code>model.create()</code> |
| R | <code>model.find()</code> , <code>model.findOne()</code> , <code>model.findById()</code> |
| U | <code>model.update()</code> , <code>model.findByIdAndUpdate()</code> , <code>model.findOneAndUpdate()</code> |
| D | <code>model.remove()</code> , <code>model.findByIdAndRemove()</code> , <code>model.findOneAndRemove()</code> |

En nuestro fichero **item.controllers.js** añadiremos este código:

```

import { crudControllers } from '../utils/crud'
import { Item } from './item.model'

export default crudControllers(Item)

```

El modelo lo hemos definido en el apartado anterior y lo que queremos es implementar una serie de controladores CRUD genéricos sobre los cuales aplicar cualquier modelo, en nuestro caso item, y que su funcionamiento sea completamente agnóstico de todo lo demás.

Abrimos el fichero **utils > crud.js** y tendremos esta plantilla de código:

```

export const getOne = model => async (req, res) => {}

export const getMany = model => async (req, res) => {}

export const createOne = model => async (req, res) => {}

export const updateOne = model => async (req, res) => {}

export const removeOne = model => async (req, res) => {}

export const crudControllers = model => ({
  removeOne: removeOne(model),
  updateOne: updateOne(model),
  getMany: getMany(model),
  getOne: getOne(model),
  createOne: createOne(model)
})

```

Si ejecutamos `npm run test-controllers` veremos este resultado en consola:

```

FAIL src/utils/__tests__/crud.spec.js
  crud controllers
    getOne
      X finds by authenticated user and id (535ms)
      X 404 if no doc was found (78ms)
    getMany
      X finds array of docs by authenticated user (69ms)
    createOne
      X creates a new doc (32ms)
      X createdBy should be the authenticated user (15ms)
    updateOne
      X finds doc by authenticated user and id to update (60ms)
      X 400 if no doc (15ms)
    removeOne
      X first doc by authenticated user and id to remove (48ms)
      X 400 if no doc (13ms)

```

Debemos implementar cada uno de los métodos para poder pasar cada uno de los test.

Nota

Nótese cómo los controladores no tienen en cuenta nada relativo al modelo y solo basan su comportamiento en la petición (`req`) y la respuesta (`res`).

Empezaremos por el código de `getOne`:

```
export const getOne = model => async (req, res) => {
  try {
    const doc = await model
      .findOne({ createdBy: req.user._id, _id: req.params.id })
      .lean()
      .exec()

    if (!doc) {
      return res.status(400).end()
    }

    res.status(200).json({ data: doc })
  } catch (e) {
    console.error(e)
    res.status(400).end()
  }
}
```

Hay una gran parte de *boilerplate* que se repetirá para los demás comandos. La parte interesante que debemos analizar es:

```
const doc = await model
  .findOne({ createdBy: req.user._id, _id: req.params.id })
  .lean()
  .exec()
```

1. El `await` es necesario porque es un proceso asíncrono, la conexión a base de datos.
2. Sobre el `model` ejecutamos el método `findOne()`.
3. Buscamos `createdBy: req.user._id` > el id del usuario que hizo la petición.
4. Buscamos `_id: req.params.id` > el id pasado en el parámetro de la ruta.
5. `lean()` mejora el rendimiento de la petición.
6. `exec()` ejecuta la consulta.

Teniendo claro este primer ejemplo, los demás son extremadamente similares.

Para `getMany` (todos los items `createdBy: req.user._id`):

```

export const getMany = model => async (req, res) => {
  try {
    const docs = await model
      .find({ createdBy: req.user._id })
      .lean()
      .exec()

    res.status(200).json({ data: docs })
  } catch (e) {
    console.error(e)
    res.status(400).end()
  }
}

```

Para **createOne**:

```

export const createOne = model => async (req, res) => {
  const createdBy = req.user._id
  try {
    const doc = await model.create({ ...req.body, createdBy })
    res.status(201).json({ data: doc })
  } catch (e) {
    console.error(e)
    res.status(400).end()
  }
}

```

Para **updateOne**:

```

export const updateOne = model => async (req, res) => {
  try {
    const updatedDoc = await model
      .findOneAndUpdate(
        {
          createdBy: req.user._id,
          _id: req.params.id
        },
        req.body,
        { new: true }
      )
      .lean()
      .exec()

    if (!updatedDoc) {
      return res.status(400).end()
    }

    res.status(200).json({ data: updatedDoc })
  } catch (e) {
    console.error(e)
  }
}

```

```
        res.status(400).end()
    }
}
```

Para **removeOne**:

```
export const removeOne = model => async (req, res) => {
  try {
    const removed = await model.findOneAndRemove({
      createdBy: req.user._id,
      _id: req.params.id
    })

    if (!removed) {
      return res.status(400).end()
    }

    return res.status(200).json({ data: removed })
  } catch (e) {
    console.error(e)
    res.status(400).end()
  }
}
```

Ahora podemos ejecutar los test para estos controladores con `npm run test-controllers` y obtendremos esto por consola:

```
PASS src/resources/item/__tests__/item.controllers.spec.js
  item controllers     ✓ has crud controllers (379ms)
```

```
PASS src/utils/__tests__/crud.spec.js  crud controllers
  getOne
    ✓ finds by authenticated user and id (495ms)
    ✓ 404 if no doc was found (47ms)
  getMany
    ✓ finds array of docs by authenticated user (80ms)      createOne      ✓ creates a new doc
    ✓ createdBy should be the authenticated user (53ms)
  updateOne
    ✓ finds doc by authenticated user and id to update (68ms)
    ✓ 400 if no doc (29ms)
  removeOne
    ✓ first doc by authenticated user and id to remove (62ms)
    ✓ 400 if no doc (15ms)
```

5. Diseño de API en Node

5.7. Autorización

Esta parte presenta un nivel adicional de complejidad que escapa por completo al temario de la asignatura.

Existen paquetes que nos facilitan la implementación de la autorización e identificación de usuarios. Todo lo que se trabaja e implementa en este punto del ejercicio es solo una parte de lo que obtenemos si instalamos el paquete [express-jwt](#) en nuestro proyecto y lo incluimos como *middleware*.

La tecnología que se usa en esta propuesta, que no es la única ni necesariamente la mejor, es la de [Json Web Tokens \(JWT\)](#), cuya ventaja más evidente es que permite que el servidor no tenga estado de sesión, ya que todo el proceso de verificación se hace con la entrega de un *token* con parte de los secretos del servidor al usuario y la validación de las peticiones de este sobre la base de la codificación de la petición con esos *tokens*.

5. Diseño de API en Node

5.8. Bibliografía

- *Framework* más opinionado construido sobre Express: [Hapi](#)
- Otros tipos de base de datos [PostgreSQL](#)
- Lenguaje de consultas [GraphQL](#)