Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

GREATIVE

Desarrollo de aplicación Web SPA en Angular para simular el portafolio anotado del diseñador

Miguel Alonso Hernández

Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA EN DISEÑO INDUSTRIAL Y DESARROLLO DE PRODUCTO

Greative: Desarrollo de aplicación Web SPA en Angular para simular el portafolio anotado del diseñador

Autor:

Alonso Hernández, Miguel

Tutor:

Escudero Mancebo, David
Departamento de Informática

Valladolid, julio de 2021.

DEDICATORIAS

A mis padres, Jesús y Marimar, ya que nada habría sido posible sin ellos.

A mis hermanos, Javi y Marta, que desde que soy un niño han sido mi principal inspiración y modelo a seguir.

A Ángela, por su apoyo y cariño incondicional desde el principio. Te amo.

A mis amigos, y en especial, a Iván, que siempre ha tenido tiempo para animarme y prestarme su ayuda cuando lo he necesitado.

A Dani, que, aunque no pueda celebrar esto con él, seguro que está orgulloso allá donde esté. Te echamos de menos.

RESUMEN

Hoy en día, la mayoría de empresas, organizaciones e instituciones de todo el mundo, cuentan con su propia aplicación o web para presentarse o dirigir su negocio. Ante esta informatización progresiva, mediante el presente proyecto, he querido llevar este contexto al diseño industrial, creando una aplicación para facilitar el trabajo de un diseñador. Para ello, he optado por una SPA (Single-Page Application) en Angular, uno de los framework más conocidos actualmente, cuyo dominio es cada vez más demandado por las empresas punteras. Previamente, he realizado un estudio sobre las principales tecnologías de programación. Posteriormente, he profundizado en los fundamentos de Angular. Por último, utilizando los conceptos aprendidos, he creado 'Greative', un cuaderno virtual de apuntes para un diseñador, donde incluir sus ideas y almacenar aquellos modelos que le inspiren. El proyecto realizado documenta el estudio previamente mencionado, y sirve como guía básica para el desarrollo de aplicaciones en Angular.

Palabras clave: Angular, Aplicación, Desarrollo, Diseño, Programación.

0 ÍNDICE

ÍNDICE

ÍNDICE DE FIGURAS

1

INTRODUCCIÓN

1. INTRODUCCIÓN

Hoy en día, la mayoría de las empresas y organizaciones cuentan con su propio sitio o aplicación web, debido a que el mundo en el que vivimos está cada vez más informatizado, e Internet cuenta con miles de millones de usuarios. Tanto es así, que es el medio de comunicación más utilizado del mundo.

En Europa, según "Eurostat", ha habido un incremento porcentual del 10% en los últimos 5 años en cuanto a la población de entre 16 y 74 años que usó Internet en los últimos tres meses, pasando del 78% al 88% [9]. Concretamente en España, hemos pasado de no tener fibra a ser de los países más avanzados en apenas cinco años. Y es que, según las "Encuestas sobre Equipamiento y Uso de Tecnologías de la Información y Comunicación en los Hogares" realizadas por el "Instituto Nacional de Estadística" en los últimos años, España ha pasado de un 78,7% de población entre 16 y 74 años que utilizó Internet en los últimos tres meses en 2015, a un 93,2% en 2020 [23], lo que supone un aumento de casi el 15%. Ante este evidente crecimiento, el desarrollo web adquiere gran importancia.

Este proyecto consiste en un estudio sobre el desarrollo de aplicaciones web, desde los conceptos y lenguajes más básicos, hasta alcanzar el objetivo principal del trabajo: el desarrollo de una aplicación de una sola página (*SPA*[1]) en Angular, uno de los *framework*[2] más populares actualmente, relacionada con el diseño industrial.

La aplicación desarrollada es 'Greative'. Se trata de un *Annotated Portfolio* virtual para diseñadores. Es decir, una plataforma donde un diseñador puede almacenar aquellos productos que le inspiren, para facilitar la etapa inicial del proceso de diseño, que es la investigación.

En el punto 2 hablaré de los fundamentos de las tecnologías básicas para el desarrollo de aplicaciones web. En el 3 explicaré el funcionamiento de Angular, así como sus estructuras y el flujo de ejecución de las aplicaciones creadas en este *framework*. Además, comentaré las diferencias más importantes entre TypeScript (lenguaje que utiliza Angular) y JavaScript. En los puntos 4 y 5, explicaré en detalle la aplicación 'Greative' y los procesos de instalación necesarios, así como el despliegue. Y, por último, en los puntos 6 y 7 comentaré las conclusiones que he sacado trabajando en este proyecto, y la bibliografía consultada.

[1] *Single-Page Application. Aplicación de una sola página.*
[2] *Marco o entorno de trabajo. Esquema, estructura o patrón para el desarrollo de aplicaciones.*

2

DESARROLLO DE APLICACIONES WEB. TECNOLOGÍAS BÁSICAS

2. DESARROLLO DE APLICACIONES WEB. TECNOLOGÍAS BÁSICAS

2.1. HTML

Se explicará HTML y sus fundamentos siguiendo el documento de MDN Web Docs especificado en la bibliografía [25].

HTML (HyperText Markup Language) no es en sentido estricto un lenguaje de programación, sino un lenguaje de marcado que despliega y estructura una página o aplicación web, pero sin darle formato, ni estilo, ni funcionalidad más allá de hiperenlaces. Por ello, actualmente en el diseño de aplicaciones y páginas web nunca se utiliza solamente HTML, sino que se utiliza junto con CSS (para describir la apariencia) y JavaScript (para aportar mayor funcionalidad y comportamiento).

HyperText hace referencia a los enlaces que conectan entre sí a las páginas web, ya sea en un mismo sitio web o entre sitios web.

HTML utiliza etiquetas para mostrar texto, imágenes y demás contenido en un navegador web. Por ejemplo <title>, para crear un título: <title>HOLA MUNDO<title>

ANATOMÍA DE UN ELEMENTO HTML

Las partes principales de un elemento son:

1. Etiqueta de apertura. Se escribe entre "<" y ">" el tipo de elemento. Establece dónde comienza el elemento. En el caso de la figura 1, dónde empieza el párrafo.
2. Etiqueta de cierre. Igual que la de apertura, pero se utiliza "</" y ">". Establece dónde termina el elemento.
3. Contenido. En el caso de la figura 1, sólo texto.
4. Elemento. El conjunto formado por la etiqueta de apertura, la etiqueta de cierre y el contenido, forman el elemento.

Figura 1. Anatomía de un elemento HTML.

Un elemento también puede tener atributos, que se escriben dentro de la etiqueta de apertura. Aportan información adicional al elemento, que no queremos que aparezca en el contenido. Por ejemplo:

Figura 2. Especificación de atributos de un elemento HTML.

En el ejemplo de la figura 2, *class* es el nombre del atributo, y *editor-note* es el valor del atributo. Este tipo de atributo le da un nombre al elemento para identificarlo, por ejemplo, a la hora de agregarle estilo.

Se escribe el nombre del atributo, seguido de un signo igual (=), y después el valor del atributo entre comillas.

ANIDAMIENTO

Se pueden colocar elementos dentro de otros. Si escribimos:

Mi gato es muy gruñón.

Habremos creado un elemento párrafo ( ), con la palabra “muy” resaltada.

ELEMENTOS VACÍOS

Son aquellos elementos que no poseen contenido. Por ejemplo, el elemento <img> se utiliza así:

<img src="deportes.jpg" alt="Un balón de fútbol sobre el césped de un estadio.">

No poseen etiqueta de cierre ni contenido, ya que un elemento *imagen* sólo necesita el atributo *src* que hace referencia a la dirección de la imagen que queremos representar. Explicaré el atributo *alt* más adelante.

ANATOMÍA DE UN DOCUMENTO HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mi página de prueba</title>
  <head>
  <body>
    . . .
```

```
  </body>
</html>
```

- <!DOCTYPE html>. El tipo de documento. Es un preámbulo requerido. Anteriormente, actuaba como un vínculo a un conjunto de reglas que el código HTML de la página debía seguir para ser considerado bueno, lo que podía significar la verificación automática de errores y algunas otras cosas de utilidad. Sin embargo, hoy en día es simplemente algo que se debe incluir para que todo funcione correctamente.

- <html></html>. Este elemento encierra todo el contenido del documento HTML.

- <head></head>. Este elemento contiene todo aquello no visible por los usuarios, que se quiere incluir en la página HTML. Por ejemplo, palabras clave (keywords), una descripción de la página que quieres que aparezca en resultados de búsquedas, código CSS para dar estilo al contenido, declaraciones del juego de caracteres (como utf-8), scripts creados con JavaScript, etc.

- <meta charset="utf-8">. Este elemento establece que el juego de caracteres que el documento usará es utf-8. Incluye casi todos los caracteres de todos los idiomas. Básicamente, puede manejar cualquier contenido de texto. Es recomendable utilizarlo siempre, ya que puede evitarnos problemas.

- <title></title>. Define el título de la página. Es el título que aparece en la pestaña o en la barra de título del navegador cuando la página es cargada.

- <body></body>. Encierra todo el contenido que se desea mostrar a los usuarios que visiten la página, ya sea texto, imágenes, vídeos, juegos, pistas de audio reproducibles, y demás.

HOLA MUNDO

```
...
  <body>
    <h1>¡HOLA MUNDO!</h1>
  </body>
...
```

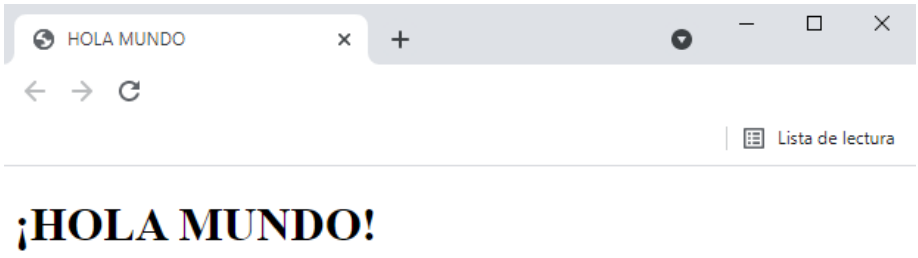

Figura 3. Página HTML de ejemplo "HOLA MUNDO".

IMÁGENES

Vamos a considerar el siguiente elemento *img* de ejemplo:

<img src="deportes.jpg" alt="Un balón de fútbol sobre el césped de un estadio.">

Incrusta la imagen deportes.jpg en el lugar donde ubiquemos esta etiqueta. El atributo *src* (source) contiene el *path* (ruta o ubicación) donde se encuentra el archivo de imagen que queremos representar. En este caso, la ruta es local, es decir, la imagen se encuentra en el dispositivo, concretamente, en la misma carpeta donde tenemos el propio archivo *html*. Sin embargo, si queremos representar una imagen de internet, podemos utilizar la URL correspondiente.

Además, tenemos el atributo *alt* (alternative) el cual contiene un texto que describe la imagen, que recibirán aquellos usuarios que no la puedan ver, por causas como:

1. Ceguera o deficiencias visuales. Los usuarios con impedimentos visuales usualmente utilizan 'Lectores de Pantalla', los cuales les leen el texto contenido en el atributo *alt* de las imágenes.

2. Error en el código. Es posible que la ruta de la imagen esté mal especificada, y por ello la página no pueda recibir y plasmar la imagen.

El valor del atributo *alt* debe proporcionarle al lector la suficiente información como para que este tenga una buena idea de qué muestra la imagen.

MARCADO DE TEXTO

Html ofrece etiquetas diferentes para textos según sean encabezados, subencabezados, párrafos, listas, etc.

Para encabezados, tenemos desde <h1> hasta <h6>, aunque normalmente se usa hasta el <h4>. Para crear párrafos, se utiliza la etiqueta <p>.

Otro tipo de marcado de texto, son las listas, que se utilizan mucho. Por ello *HTML* ofrece elementos específicos para ellas. Hay dos tipos de listas, las ordenadas y las desordenadas, cuya diferencia es evidente.

1. Las listas desordenadas se utilizan cuando el orden de los ítems no es relevante (por ejemplo, la lista de la compra). Se encierran con la etiqueta <ul> (unordered list).
2. Las listas ordenadas se utilizan cuando el orden sí es relevante (por ejemplo, una receta). Se encierran con la etiqueta <ol> (ordered list).

Cada ítem de la lista se coloca dentro de un elemento <li> (list item).

Un ejemplo de lista podría ser:

```
<p>'Los Cuatro de Glasgow' son:</p>
<ul>
  <li>Charles Rennie Mackintosh</li>
  <li>Frances Macdonald</li>
  <li>Margaret Macdonald</li>
  <li>Herbert McNair</li>
</ul>
```

Otro elemento muy utilizado es . Se utiliza cuando se quiere dar un formato específico a alguna parte de un texto.

Y, por último, hay que mencionar la etiqueta más utilizada, y más genérica. El elemento se utiliza para definir una división o sección del documento. Además, se utiliza como contenedor del resto de los elementos HTML.

VÍNCULOS

Los vínculos o enlaces se implementan con la etiqueta <a> (anchor), y se añade la dirección web deseada como valor del atributo *href*. Por ejemplo:

```
<a href="https://www.google.es/">Google.</a>
```

Así quedaría una página de ejemplo utilizando los conceptos explicados.

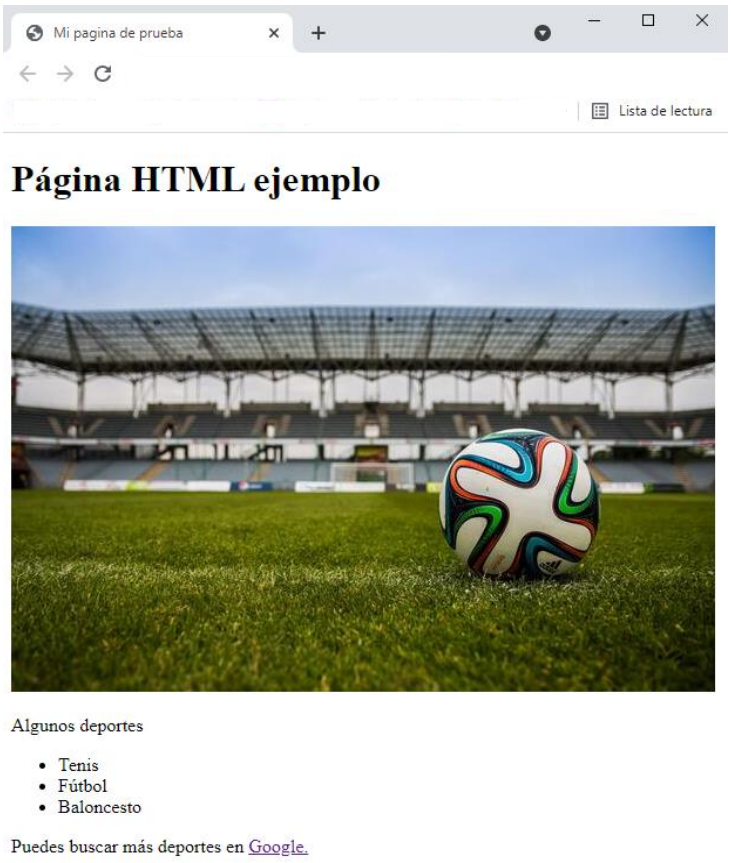

Figura 4. Página HTML de ejemplo con imagen.

2.2. CSS

De nuevo, me basaré en los documentos de MDN Web Docs [26].

CSS (Cascading Style Sheets, Hojas de Estilo en Cascada), tampoco es un lenguaje de programación exactamente. Es un lenguaje de hojas de estilo que nos sirve para estilizar una página web, configurando detalles como el color del texto, la posición de ciertos contenidos, fuentes, etc., de manera selectiva a elementos en documentos HTML.

Por ejemplo, si quiero que todos los párrafos (etiqueta <p> en HTML) sean de color azul, escribiría lo siguiente en mi hoja de estilos:

```
p {
     color: blue;
}
```

Guardamos el archivo en la misma ubicación que index.html, con el nombre "style.css". Para aplicar la hoja de estilos a un documento HTML, debemos incluir la siguiente etiqueta dentro del <head> del archivo index.html:

```
<link href="style.css" rel="stylesheet" type="text/css">
```

ANATOMÍA DE UNA REGLA CSS

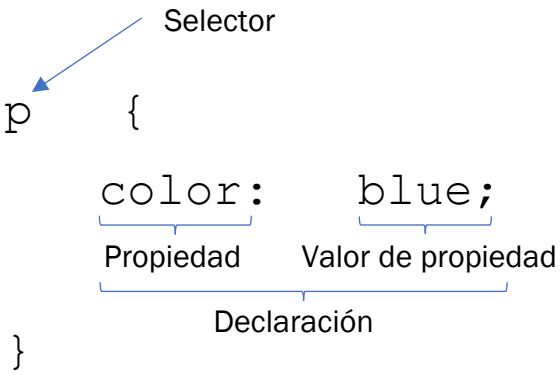

Figura 5. Anatomía de una regla CSS.

- Selector. Es el elemento HTML al que se va a aplicar la propiedad.
- Declaración. Es la propia especificación de la propiedad.
- Propiedad. Características que podemos definir para dar estilo a un elemento HTML.
- Valor de la propiedad. Se escribe a la derecha de la propiedad, después de los dos puntos, para elegir la apariencia que queremos darle a la propiedad en cuestión.

Como se ve en el esquema, cada regla debe estar encapsulada entre llaves. Dentro de cada declaración se deben usar dos puntos entre la propiedad y el valor de la propiedad. Dentro de cada regla, se debe usar un punto y coma para separar una declaración de la siguiente, y normalmente, se hace cada declaración en una línea. Por ejemplo, así:

```
p {
     color: blue;
     width: 500px;
     border: 1px solid black;
}
```

Además, se pueden aplicar las mismas declaraciones a varios elementos a la vez, de esta manera:

```
p, li, h1 {
     color: blue;
}
```

Podemos hacer referencia al elemento de varias formas. Haciendo referencia al elemento directamente, a la identificación *id* (si id='ejemplo', el selector sería #ejemplo), a la clase (si class='ejemplo', el selector sería .ejemplo), a todos los elementos del documento (utilizando * como selector), o a la pseudoclase (elemento en algún estado especificado, por ejemplo, *selector*:hover hace referencia al

elemento referenciado con el selector cuando el usuario pase el puntero por encima de ese elemento).

Se pueden definir muchas propiedades en CSS, todas ellas consultables en páginas como la referenciada en la bibliografía en [38].

Figura 6. Página HTML de ejemplo con texto y estilo CSS.

2.3. JAVASCRIPT

Explicaré este punto basándome en las referencias [27] y [40] de la bibliografía.

JavaScript (JS) es un lenguaje de programación que permite implementar funciones complejas en una página web. Se utiliza junto con HTML para crear interactividad dinámica en los sitios web, y con CSS para el estilo.

Aporta a la página web mayor funcionalidad y un comportamiento más complejo. Además, este lenguaje está en constante evolución ya que los desarrolladores lo actualizan añadiendo nuevas herramientas, desbloqueando funcionalidades adicionales.

COMENTARIOS

En JavaScript, como en cualquier lenguaje de programación, podemos escribir comentarios, los cuáles el navegador no va a ejecutar. Sirven básicamente para

organizar un poco el código y facilitar su depuración o reparación en caso de ser necesario.

Para crear comentarios de línea se utiliza "//", y para comentarios de varias líneas se puede utilizar "/*" para indicar el principio de comentario y "*/" para el final.

VARIABLES

Las variables son, básicamente, contenedores de datos. Para declarar una variable se utiliza la palabra clave *let*. La palabra clave *var* que también sirve para declarar variables, pero desde que existe *let,* ya no se usa, ya que en algunos casos puede generarse un código confuso. Más adelante entraré en detalles.

Se declaran variables de esta manera:

```
let nombreDeLaVariable;
```

A la hora de elegir un nombre para las variables, existen ciertas restricciones y recomendaciones. Se recomienda el estilo camelCase, que consiste en poner la primera palabra en minúsculas y las siguientes con la primera letra en mayúscula.

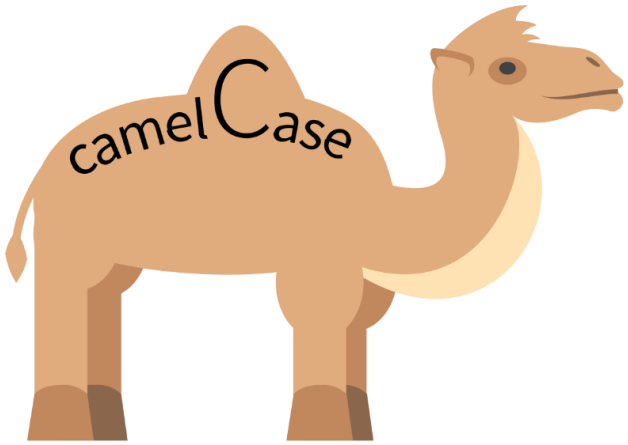

Figura 7. Metáfora visual del estilo camelCase.

A mayores, tenemos la palabra clave *const*, la cual sirve para declarar una constante, es decir, un valor que vamos a utilizar a lo largo del código y que no queremos que cambie. Si intentásemos cambiar una constante, nos daría un error.

Una vez declarada la variable, se le debe asignar un valor (que puede ser de varios tipos, como veremos más adelante).

```
nombreDeLaVariable = 'valor de ejemplo';
```

Podemos declarar la variable y asignarla un valor en una misma línea de código, de esta manera:

```
let nombreDeLaVariable = 'valor de ejemplo';
```

Si queremos cambiar el valor de una variable ya declarada, sería:

```
nombreDeLaVariable = 'valor cambiado';
```

Los tipos de datos que una variable puede almacenar son.

- String (cadena). Es una secuencia de texto. Recibe ese nombre ya que es una "cadena de caracteres". Para que JavaScript entienda que queremos un valor de tipo *string*, debemos escribir el valor entre comillas.

- Number. Un número. Se escribe sin comillas.

- Boolean. Solo toman valor verdadero o falso, es decir *true* o *false*, y se escribe sin comillas ya que si no sería de tipo *string*.

- Array. Es una estructura que nos permite almacenar varios valores en una sola variable. Hay de tipo vector, y tipo matriz. Un vector almacena una fila de datos. Una matriz almacena varias filas de datos.

- Object. En JavaScript, cualquier cosa es un objeto, y puede ser almacenado en una variable.

- null. Un valor nulo.

- undefined. Un valor no definido.

Volviendo a la diferencia entre *var* y *let*. Si declaramos una variable con *var* y le asignamos un valor, y líneas después declaramos por error una variable con *var* con el mismo nombre que la primera, no nos daría error, sino que sobrescribiría la primera, y quizá nuestro programa finalmente no funcione como queremos. Al no aparecer ningún error, resultaría tedioso encontrar y resolver el problema. Sin embargo, haciendo esto mismo con *let,* al final nos daría un error porque se ha declarado una variable con el mismo nombre que una que ya estaba declarada. Este es el funcionamiento más deseable.

Una variable declarada, pero sin un valor asignado, tiene el valor "undefined". A continuación, un ejemplo de cómo actúan *let* y *var*.

```
var a;

console.log('El valor de a es ' + a);

//Se mostrará en consola: El valor de a es undefined

console.log('El valor de b es ' + b);

//Se mostrará en consola: El valor de b es undefined

var b;

console.log('El valor de c es ' + c);

//Uncaught ReferenceError: c is not defined
```

Vemos que a pesar de que “b” esté declarada al final del código, al utilizar “var”, el navegador encuentra “b” antes de mostrarla en consola. La variable “c” no está declarada en ningún momento, por lo tanto, sale un error de referencia. Sin embargo, utilizando “let”:

```
let x;

console.log('El valor de x es ' + x);

//Se mostrará en consola: El valor de x es undefined

console.log('El valor de y es ' + y);

//Uncaught ReferenceError: y is not defined

let y;
```

Utilizando *let*, el navegador no entiende que la variable ‘y’ exista hasta que no lee ‘*let y’*. Por eso, lee ‘console.log’ antes de que la variable exista, y se muestra un error de referencia.

Una variable se dice que es global si está declarada fuera de una función y, por consiguiente, está disponible para cualquier parte del código del documento. Por otro lado, se dice que una variable es local si está declarada dentro de una función y, por tanto, sólo está disponible dentro de esa función. Utilizando *var* para declarar una variable dentro de una declaración de bloque, se está declarando de forma local, pero se va a comportar de forma global. Es decir, si ejecutamos el siguiente código:

```
if (true) {
     var x = 5;
}
console.log(x);
```

Aparecerá en consola el valor 5. Sin embargo, si ejecutamos este otro código:

```
if (true) {
     let y = 5;
}
console.log(y);
```

Aparecerá en consola un error de referencia porque al utilizar *let*, la variable ‘y’ sólo existe dentro del bloque.

Otro ejemplo similar, si utilizamos un bucle for.

```
var i = 2;

document.write("Durante el bucle: ");

for (var i = 0; i < 4; i++) {
```

```
  document.write(i + " ");
}
document.write("<br>Después del bucle: " + i);
```

En este caso utilizando “var”, aparecerá en pantalla:

Durante el bucle: 0 1 2 3

Después del bucle: 4

Sin embargo, si utilizamos “let”:

```
let i = 2;
document.write("Durante el bucle: ");
for (let i = 0; i < 4; i++) {
  document.write(i + " ");
}
document.write("<br>Después del bucle: " + i);
```

En este caso utilizando “let”, aparecerá en pantalla:

Durante el bucle: 0 1 2 3

Después del bucle: 2

STRINGS

Para asignar un valor de tipo *string* (cadena de caracteres) se utilizan comillas (dobles “, o simples ‘). Por ejemplo:

“Esto es un string”;

‘Esto es otro string’;

Algunas propiedades y métodos de *strings*:

- length. Se usa para obtener la longitud de la cadena, es decir, el número de caracteres. Por ejemplo.
  ```
  let texto = 'Hola Mundo';
  texto.length;
  ```
  Esto devolverá 10, porque tiene 10 caracteres (incluido el espacio).
- indexOf(“*cadena a buscar*”). Devuelve la posición de la primera coincidencia de la cadena buscada dentro del texto.

- lastIndexOf("*cadena a buscar*"). Devuelve la posición de la última coincidencia de la cadena buscada dentro del texto. indexOf y lastIndexOf devuelven -1 si no se encuentra la cadena.

- slice(*inicio , fin*). Extrae parte de una cadena, y devuelve esa parte en una nueva cadena.

- substring(*inicio , fin*). Es similar a slice, pero no admite índices negativos. Si no se especifica '*fin*', se entiende que es hasta el final de la cadena.

- substr(*inicio , tamaño*). Es similar a slice, pero en vez de especificarse '*fin*', se especifica el tamaño de la cadena a extraer desde '*inicio*'.

- replace("*texto actual*", "*texto sustituto*"). Devuelve la misma cadena, pero cambiando "*texto actual*" por "*texto sustituto*". Es importante saber que este método no modifica la cadena original, si no que crea una nueva. Además:

    - por defecto, se distingue entre mayúsculas y minúsculas. Si queremos que sustituya una parte tanto si coincide las mayúsculas y minúsculas como si no, debemos escribir */TEXTO ACTUAL/i*, en lugar de "*texto actual*".

    - por defecto sólo se modifica la primera coincidencia. Si queremos cambiar todas las coincidencias debemos escribir */texto actual/g*, en lugar de "*texto actual*". Se puede utilizar conjuntamente /ig para sustituir todas las coincidencias sin tener en cuenta mayúsculas y minúsculas.

- toLowerCase( ) / toUpperCase ( ). Estas propiedades devuelven la misma cadena, pero con todas las letras en minúscula (toLowerCase) o en mayúscula (toUpperCase). Por ejemplo, si utilizamos la variable 'texto' de antes:
  `texto.toLowerCase( );`
  Esto devolverá: 'hola mundo'.
  `texto.toUpperCase( );`
  Esto devolverá: 'HOLA MUNDO'.

- concat("*cadena 1*", "*cadena 2*"). Sirve para unir dos o más cadenas. Se puede utilizar en lugar de concat( ), el operador '+'.

- trim( ). Elimina los espacios blancos a cada lado de una cadena.

Podemos acceder a un caracter concreto de una cadena especificando la posición deseada entre corchetes [ ].

```
texto[0];   // Devuelve el primer carácter 'H'
texto[1];   // Devuelve el segundo carácter 'o';
```

Incluso podemos utilizar la propiedad length para acceder a posiciones desde el final.

```
texto [ texto.length – 2 ];   // Devuelve 'd', porque es la segunda posición desde el final.
```

Otro tipo de string son los *template strings.* Se utilizan cuando parte de la cadena queremos que sea el valor de una variable previamente declarada. Se utilizan *backticks* `` en lugar de comillas. Para representar el valor de una variable dentro de una *template string* escribimos ${ *nombreVariable* }. Por ejemplo:

```
let origen = 'Irlanda';

let informacion = `Eileen Gray nació en ${ origen } en 1878.`;
```

Si, por ejemplo, quisiéramos añadir más información, podríamos usar varias líneas.

```
let ciudad = 'Enniscorthy';

let informacionDetalle = `Eileen Gray nació en ${ origen }
                  en 1878, en la ciudad de ${ ciudad } `;
```

OPERADORES

Son símbolos matemáticos que actúan sobre dos valores y producen un resultado. Tenemos diferentes operadores:

- Operadores aritméticos:
  - Suma +. Se utiliza para sumar dos números (*number*), o unir dos cadenas de caracteres (*string*), es decir, para concatenar.
  - Resta -, multiplicación * y división /. Se utilizan con números y funcionan igual que en matemáticas.
  - Resto de división %. Por ejemplo, el resto de 21 entre 5, es 1 y se obtiene como 21%5.
  - Potencia **. Por ejemplo, 2 elevado a 4, sería 2**4.
  - Incremento ++, y decremento - -. Sirve para aumentar o disminuir 1 unidad.

- Operadores de asignación:
  - Asignación básica, =. Sirve para poder asignar un valor a alguna variable.
  - Aumento, +=. Por ejemplo, a += b, es lo mismo que a = a + b. Sirve también para concatenar cadenas.
  - Disminución, -=. Por ejemplo, a -= b, es lo mismo que a = a - b.
  - *=. Por ejemplo, a *= b, es lo mismo que a = a * b.
  - /=. Por ejemplo, a /= b, es lo mismo que a = a / b.
  - %=. Por ejemplo, a %= b, es lo mismo que a = a % b.
  - **=. Por ejemplo, a ** b, es lo mismo que a = a**b.

- Operadores de comparación:
  - Igualdad básica, ==. Sirve para comprobar si dos valores son iguales.
  - Igualdad estricta, ===. Sirve para comprobar si dos valores son iguales y del mismo tipo.
  - Desigualdad, !=. Sirve para comprobar si dos valores son diferentes.
  - Desigualdad estricta, !==. Sirve para comprobar si dos valores son diferentes o son de diferente tipo.
  - Mayor, >, y mayor o igual >=.
  - Menor, <. Y menor o igual <=.

- Operadores lógicos:
  - &&. Sirve para añadir condiciones, las cuáles todas deben cumplirse (corresponde con la conjunción 'y').
  - ||. Sirve para añadir condiciones, de tal forma que es suficiente con que se cumpla una de ellas (corresponde con la conjunción 'o').
  - !. Sirve para cambiar *true* por *false* y viceversa (corresponde con "no").

Aunque hay algunos más, estos son los básicos.

CONDICIONALES

Son estructuras de código que se utilizan para ejecutar una acción u otra, según se cumpla o no la condición especificada.

If... else

La estructura condicional más utilizada es *if... else:*

```
let color = 'azul';

if (color === 'azul') {

     alert ('Este color es AZUL.');

} else {

     alert ('Este color NO ES AZUL.');

}
```

Después de *if* se escribe entre paréntesis la condición. Si se cumple esa condición se ejecuta el código que está dentro de las primeras llaves. Si no se cumple, se ejecuta el código que hay entre llaves inmediatamente después de *else*.

Si hay varias posibilidades, y queremos que se realice una acción en función de cada una de ellas, podemos añadir *else if*, y al final *else* para especificar la acción que se debe realizar en caso de que ninguna de las opciones anteriores se cumpla. Por ejemplo:

```
let color = 'azul';
```

```
if (color === 'azul') {
     alert ('Este color es AZUL.');
} else if (color === 'rojo'){
     alert ('Este color es ROJO.');
}else {
     alert ('Este color NO ES NI AZUL NI ROJO.');
}
```

Switch

Otra estructura condicional a tener en cuenta es *switch*. Esta estructura considera un dato y realiza una acción según el valor que tome. Si no toma ningún valor de los especificados, se realiza la acción establecida por defecto. Por ejemplo:

```
let dia;
switch (new Date().getDay()) {
  case 5:
    dia = "Viernes";
    break;
  case 6:
    dia = "Sábado";
    break;
  case 0:
    dia = "Domingo";
    break;
  default:
    dia = "Esperando al fin de semana";
}
```

El comando *new Date().getDay(),* obtiene el día actual en número, siendo el 0 domingo, el 1 el lunes... hasta el 6 que es sábado. La palabra *break* hace que se deje de leer la estructura, es decir, hace que se salga de ella. Por ejemplo, si hoy es sábado, se entra en la estructura *switch*, pasamos por el caso 5, viernes, y se sigue leyendo. Después llegamos al caso 6, que es sábado. Se accede a este caso, se almacena "Sábado" en la variable 'dia', y llegamos al *break*, por lo que se deja de leer y se sale de *switch*.

En el caso de que se cumplan varios casos a la vez, se selecciona el primero de ellos.

BUCLES

Los bucles se utilizan cuando queremos ejecutar acciones cierto número de veces, cada vez con un valor diferente. Los bucles son:

- for. Ejecuta un bloque de código un número de veces.
- for/in. Recorre las propiedades de un objeto.
- for/of. Recorre los valores de un objeto.
- while. Recorre un bloque de código mientras la condición especificada se cumpla.
- do/while. Recorre un bloque de código mientras la condición especificada se cumpla.

For

El bucle *for* se utiliza para ejecutar un bloque de código un número de veces.

```
for ( sentencia 1; sentencia 2; sentencia 3 ) {
     // Bloque de código a ejecutar
}
```

- Sentencia 1. Declaración e inicialización de una variable. Esto se hace una sola vez, y antes de ejecutar el bloque de código. Se pueden declarar más de una variable. Esta sentencia es opcional.
- Sentencia 2. Definición de una condición que debe cumplirse para ejecutar el bloque de código. Normalmente, es una condición con la variable de la sentencia 1. Esta sentencia también es opcional, pero si se omite, se debe incluir un *break* dentro del bloque de código. Si no el bucle no terminará nunca, y el navegador colapsará.
- Sentencia 3. Acción que se ejecuta cada vez que termina de ejecutarse el bloque de código. Normalmente, es un incremento o decremento de la variable de la sentencia 1.

Por ejemplo:

```
let texto = "";
for (let i = 0; i < 3; i++) {
  texto += "El número es " + i + "<br>";
}
```

Cuando este bucle termine de leerse, se habrá almacenado en 'texto' lo siguiente:

El número es 0

El número es 1

El número es 2

For in

El bucle *for in* se utiliza para recorrer las propiedades de un objeto, y ejecutar un bloque de código por cada iteración. Tiene la siguiente estructura:

```
for (variable in objeto) {
     // Bloque de código a ejecutar
}
```

La variable especificada, almacena una propiedad del objeto a cada ejecución. Veamos un ejemplo:

```
const persona = {nombre:"Marcel", apellido:"Breuer", edad:26};
let texto = "";
for (let x in persona) {
  texto += persona[x] + " ";
}
document.write(texto);
```

Aparecerá por pantalla:

Marcel Breuer 26.

Básicamente, este bucle *for in* recorre el objeto 'persona'. A cada iteración, la variable 'x' almacena una propiedad del objeto (las propiedades son nombre, apellido y edad), y se utiliza persona[x] para acceder al valor de cada propiedad.

Se puede utilizar también para acceder a cada posición de una matriz, aunque no es recomendable, ya que es posible que no se acceda a los valores de la matriz en el orden esperado. Para esto es mejor usar un bucle *for*, *for of* o el método forEach( ) de matrices.

For of

El bucle *for of* se utiliza para recorrer los valores de un objeto, y ejecutar bloque de código por cada iteración. Tiene la siguiente estructura:

```
for (variable of objeto) {
     // Bloque de código a ejecutar
}
```

La variable especificada, almacena el valor de una propiedad del objeto a cada ejecución. Veamos un ejemplo:

```
const persona = ["Marcel", "Breuer", 26];
let texto = "";
for (let x of persona) {
```

```
  texto += x + " ";
}
document.write(texto);
```

Aparecerá por pantalla:

Marcel Breuer 26.

Como podemos ver, en este ejemplo la constante 'persona' almacena una matriz. Este tipo de bucle no sirve para recorrer objetos como el del ejemplo que hemos utilizado para explicar el bucle *for in*, sino para recorrer objetos como matrices, o strings. Realmente una matriz y un objeto sirven para lo mismo, pero se diferencian en la manera de acceder a los elementos.

Para acceder a los elementos de una matriz se utilizan números según la posición del elemento deseado; para acceder a los elementos de un objeto, se utilizan los nombres de las propiedades. Se puede ver la diferencia en el ejemplo de *for in* y de *for of.*

While

El bucle *while* ejecuta el bloque de código de manera reiterada mientras que la condición especificada sea *true*. Tiene la siguiente estructura:

```
while (condición) {
  // Bloque de código a ejecutar
}
```

El siguiente bucle *while* se repetiría una y otra vez mientras la variable 'i' fuese menor que 10.

```
let texto;
while (i < 10) {
  texto += "El número es " + i;
  i++;
}
```

Realmente funciona igual que el bucle *for* omitiendo las sentencias 1 y 3. Por ejemplo si quisiéramos recorrer una matriz con un bucle:

```
const coches = ["BMW", "Volvo", "Saab", "Ford"];
let i = 0;
let texto = "";
for (;coches[i];) {
  texto += coches[i];
```

```
  i++;
}
```

De esta manera resolveríamos el problema mediante un bucle *for*, pero podríamos utilizar indistintamente un bucle *while*.

```
const coches = ["BMW", "Volvo", "Saab", "Ford"];
let i = 0;
let texto = "";
while (coches[i]) {
  texto += coches[i];
  i++;
}
```

Do while

Funciona igual que el bucle while, con la diferencia de que ejecuta el bloque de código al menos una vez, antes de tener en cuenta la condición. La sintaxis es:

```
let texto;
do{
  texto += "El número es " + i;
  i++;
} while (i < 10);
```

FUNCIONES

Las funciones sirven para crear líneas de código que se necesitan reutilizar. En vez de tener que escribir las mismas líneas cada vez que se quiera ejecutar la misma acción, basta con escribirlas una vez dentro de una función, y cuando se quiera realizar esa acción de nuevo, se llama a la función. Un ejemplo:

```
function multiplica(num1,num2) {
     let resultado = num1 * num2;
     return resultado;
}
```

Con estas cuatro líneas creamos una función con el nombre “multiplica” que recibe como parámetros dos valores, que se almacenan en las variables locales “num1” y “num2”. Después, crea una variable local llamada “resultado”, donde se almacena

el resultado de multiplicar el valor de “num1” por el valor de “num2”. Y, por último, devuelve el resultado.

Una vez creada esta función, podemos utilizarla así:

```
let operacion1 = multiplica(2,5);
```

De esta manera, creamos una variable llamada “operacion1”, donde se almacena el resultado de multiplicar 2 por 5.

EVENTOS

Para crear interacciones se necesitan los eventos. Se utilizan para ejecutar determinadas acciones, definidas en una función, cuando el usuario realiza alguna interacción, por ejemplo, hacer clic. Podemos ver un ejemplo rápido escribiendo lo siguiente en la consola del navegador:

```
document.querySelector('html').onclick = function() {
     alert('Has hecho click en la página.');
}
```

Este pequeño programa va a hacer que salte una ventana emergente con el mensaje ‘Has hecho click en la página’ cada vez que hagamos click sobre la página.

Además de onclick, tenemos más eventos disponibles, que son:

- onchange. Cuando cambia un elemento.
- onclick. Cuando el usuario hace click en un elemento.
- onmouseover. Cuando pasa el puntero sobre un elemento.
- onmouseout. Cuando saca el puntero de un elemento.
- onkeydown. Cuando pulsa una tecla.
- onload. Cuando el navegador carga la página.

JAVASCRIPT HTML DOM

Cuando un navegador carga una página web, crea un DOM, *Document Object Model*, de la página. El DOM es una plataforma y una interfaz de lenguaje neutral que permite a los programas y a los scripts acceder dinámicamente a un documento y modificar su contenido, estructura y estilo.

Un DOM básico se puede representar mediante el siguiente diagrama de árbol.

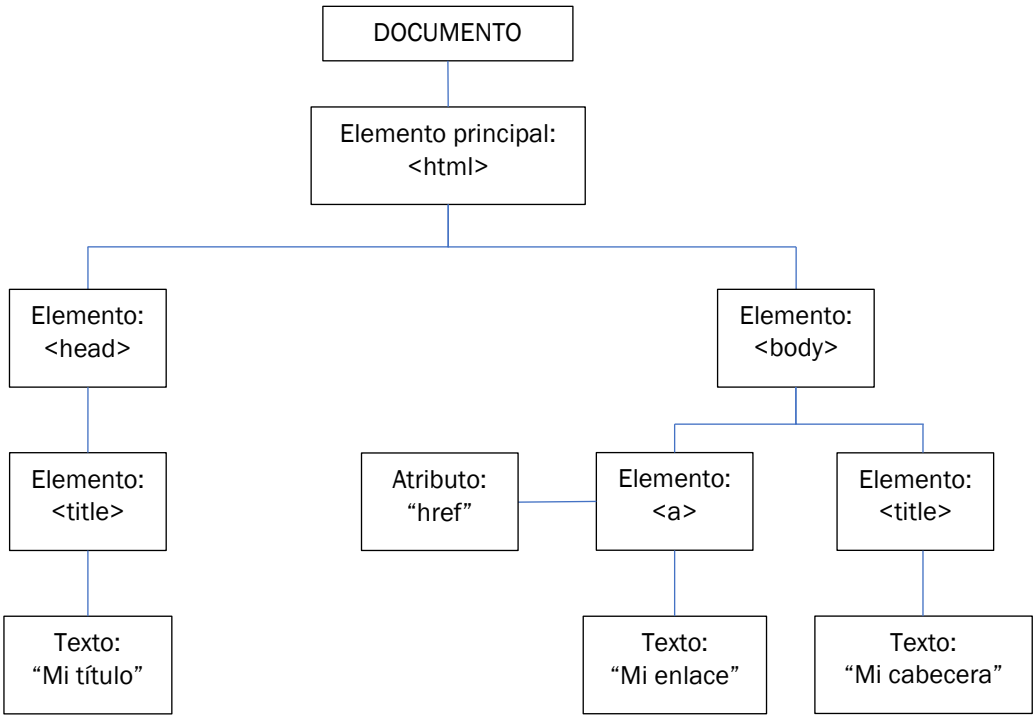

Figura 8. Estructura de un DOM básico.

JavaScript tiene acceso al DOM, y nos permite crear HTML dinámico, pudiendo:

- Cambiar todos los elementos HTML de la página.
- Cambiar todos los atributos HTML.
- Cambiar todos los estilos CSS.
- Eliminar elementos y atributos HTML.
- Añadir nuevos elementos y atributos HTML.
- Reaccionar a todos los eventos HTML.
- Crear nuevos eventos HTML.

Para el DOM, todos los elementos HTML son objetos, y cada uno de los diferentes tipos tiene sus propiedades y sus métodos. Una propiedad es un valor que podemos establecer o modificar (como el contenido de un elemento HTML). Un método es una acción que podemos realizar (como añadir o eliminar un elemento HTML).

Por ejemplo, vamos a utilizar el método *getElementById* y la propiedad *innerHTML* del elemento <p> con *id="prueba"*.

```
<html>
<body>
  <p id="prueba"></p>
  <script>
```

```
    document.getElementById("prueba").innerHTML = "Hola
Mundo";
  </script>
</body>
</html>
```

Mediante *getElementById,* se "selecciona" un elemento en función de su *id*. Utilizando *innerHTML*, accedemos al contenido del elemento, pudiendo establecerlo, obtenerlo o cambiarlo. Estos son el método y la propiedad para buscar y cambiar un elemento HTML más típicos, pero hay más.

- Para buscar o seleccionar elementos:
  - document.getElementById("*id*"). Busca un elemento según su id.
  - document.getElementsByTagName("*etiqueta*"). Busca un elemento según la etiqueta.
  - document.getElementByClassName("*clase*"). Busca un elemento según su clase.
- Para cambiar elementos:
  - *elemento*.innerHTML = "*nuevo contenido*". Cambia el contenido de un elemento HTML.
  - *elemento.atributo* = "*nuevo valor*". Cambia el valor de un atributo de un elemento HTML. En *atributo,* escribimos el atributo que deseamos definir. Por ejemplo, *elemento*.className = 'ejemplo'.
  - *elemento*.style.property = "*nuevo estilo*". Cambia el estilo de un elemento HTML.
  - *elemento*.setAttribute("*atributo*", "*valor*"). Cambia el valor de un atributo de un elemento HTML.

  Donde pone *elemento*, habrá alguno de los métodos para seleccionar elementos.
- Para añadir y eliminar elementos:
  - document.createElement("*elemento*"). Crea un elemento HTML.
  - document.removeChild("*elemento*"). Elimina un elemento HTML.
  - document.appendChild("*elemento*"). Añade un elemento HTML.
  - document.replaceChild("*elemento*"). Sustituye un elemento HTML.
  - document.writte("*texto*"). Escribe directamente en el flujo de salida HTML.

Además, podemos añadir eventos a estas sentencias. Por ejemplo:

```
document.getElementById("id").onclick = function( ) {
instrucción };
```

Define una instrucción al efectuar un evento sobre un elemento HTML, en este caso al hacer click sobre él. Más adelante, se explicará en detalle.

Si queremos acceder a un elemento en función de su etiqueta, se utiliza la propiedad getElementsByTagName. Podría interesarnos si queremos acceder a varios elementos del mismo tipo a la vez. Por ejemplo:

```
<html>
<body>
<p>Este es el PRIMER elemento p.</p>
<p>Este es el SEGUNDO elemento p.</p>
<p id="prueba"></p>
<script>
const elemento = document.getElementsByTagName("p");
document.getElementById("prueba").innerHTML = 'El texto contenido en el primer <p> es: ' + elemento[0].innerHTML;
</script>
</body>
</html>
```

Para seleccionar un elemento en función de su clase:

```
<html>
<body>
<p class="texto">Hola Mundo</p>
<p class="texto">Esto es un ejemplo de uso del método <b>getElementsByClassName</b>.</p>
<p id="prueba"></p>
<script>
const x = document.getElementsByClassName("texto");
document.getElementById("prueba").innerHTML = 'El texto contenido en el primer <p> con clase =”texto” es: ' + x[0].innerHTML;
</script>
</body>
</html>
```

Hemos visto que podemos acceder al contenido de los elementos HTML. Para modificarles se utiliza directamente la propiedad *innerHTML*. La estructura sería:

```
document.getElementById('id').innerHTML = nuevo contenido;
```

Como podrá deducirse, en este caso se ha seleccionado el elemento mediante la *id*, pero se podría haber seleccionado mediante cualquier forma explicada antes.

Si se quiere modificar el valor de un atributo de un elemento, se especifica el atributo en cuestión a continuación del *elemento*. La estructura sería:

```
document.getElementById(id).atributo = 'valor';
```

Si se quiere modificar el estilo de un elemento, se usa la propiedad *style.propiedad*. En *propiedad* se especifica la propiedad CSS a modificar. La estructura sería:

```
document.getElementById(id).style.propiedad = 'valor';
```

Por ejemplo:

```
<html>
  <body>
    <p id="p1">Hola!</p>
    <p id="p2">¿Qué tal?</p>
    <script>
      document.getElementById("p2").style.color = "blue";
      document.getElementById("p2").style.fontFamily =
"Arial";
      document.getElementById("p2").style.fontSize =
"larger";
    </script>
  </body>
</html>
```

De esta manera se cambia el color, la tipografía y el tamaño de letra del texto con *id="p2"*.

Podemos establecer estas modificaciones como consecuencia de alguna acción que realice el usuario, mediante los eventos. Por ejemplo:

```
<html>
<body>
<h1 id="titulo">Título 1</h1>
<button type="button"
onclick="document.getElementById('titulo').style.color =
'blue'">
Haz click</button>
</body>
</html>
```

Aparecerá el texto 'Título 1' en negro, y un botón. Si pulsamos el botón veremos que el texto se vuelve de color azul.

Otro ejemplo:

```
<html>
<body>
<h1 id="titulo">Título 1</h1>
<button type="button"
onclick="document.getElementById('titulo').innerHTML =
'Título modificado'">
Haz click</button>
</body>
</html>
```

En este caso, al pulsar el botón se modificaría el contenido del elemento 'titulo'. Incluso, podemos omitir el botón y que ocurra al pulsar en el propio texto, eliminando el elemento <button>, y modificando la etiqueta h1.

```
<h1 id="titulo" onclick="this.innerHTML = 'Título
modificado'">Título 1</h1>
```

Podemos también pedir que se ejecute una función. Por ejemplo:

```
<html>

<body>

<h1 id="titulo" onclick="modificarTitulo(this)">Título 1</h1>

<script>

function modificarTitulo(elemento) {

  elemento.innerHTML = "Título modificado";

}

</script>

</body>

</html>
```

La palabra clave *this* hace referencia al elemento en cuestión, en este caso, al elemento <h1>. De esta manera, la función *modificarTitulo* recibe como parámetro el elemento <h1>, para poder acceder a su contenido mediante *innerHTML*. Hemos visto el evento *onclick*, pero se pueden utilizar todos los descritos en el apartado eventos, y más (sólo están explicados los más comunes).

Para terminar JavaScript HTML DOM, vamos a ver el método *addEventListener( )*. Sirve para agregar un controlador de eventos (event handler) a un elemento. La estructura sería:

```
elemento.addEventListener(evento, función, true/false);
```

Mediante el primer parámetro *evento*, se especifica el tipo de evento que queremos utilizar, pero sin el prefijo 'on'. Por ejemplo, *click* en vez de *onclick*.

El segundo parámetro es la función que queremos que se ejecute cuando ocurra el evento.

El tercer parámetro es un valor *booleano*, es decir, sólo toma valor *true* o *false*. No es necesario especificar este parámetro, por defecto toma el valor *false*. Sirve para especificar si queremos utilizar Event Bubbling (si el parámetro es *false*) o Event Capturing (si el parámetro es *true)*. Para definir estos conceptos, me basaré en una publicación de Amit Diwan [8].

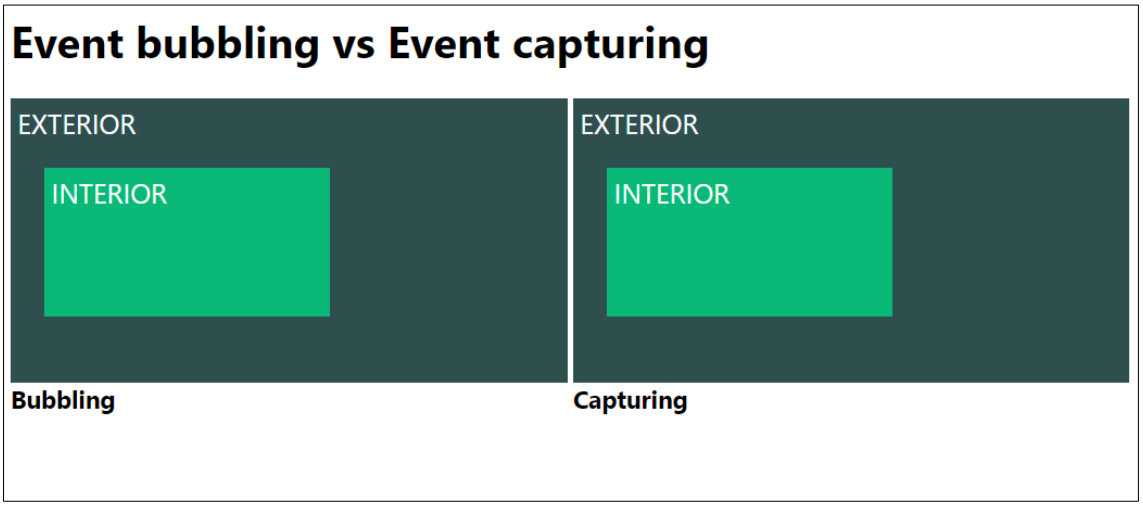

Figura 9. Ejemplo de funcionamiento de Event Bubbling y Event Capturing.

La figura muestra un ejemplo representativo de lo que es Event Bubbling y Event Capturing. El código implementado hace que, si hacemos click en el recuadro verde claro "INTERIOR", se muestre debajo "Has hecho click sobre 'INTERIOR'". Y si hacemos click sobre el verde oscuro "EXTERIOR", se muestra debajo "Has hecho click sobre 'Exterior'". El problema está en que al pulsar sobre el recuadro "INTERIOR", también estamos haciendo click en el "EXTERIOR", porque el pequeño está contenido en el grande. Ahora sí, veamos qué pasa si hacemos click en el recuadro pequeño de la izquierda, es decir, utilizando Event Bubbling.

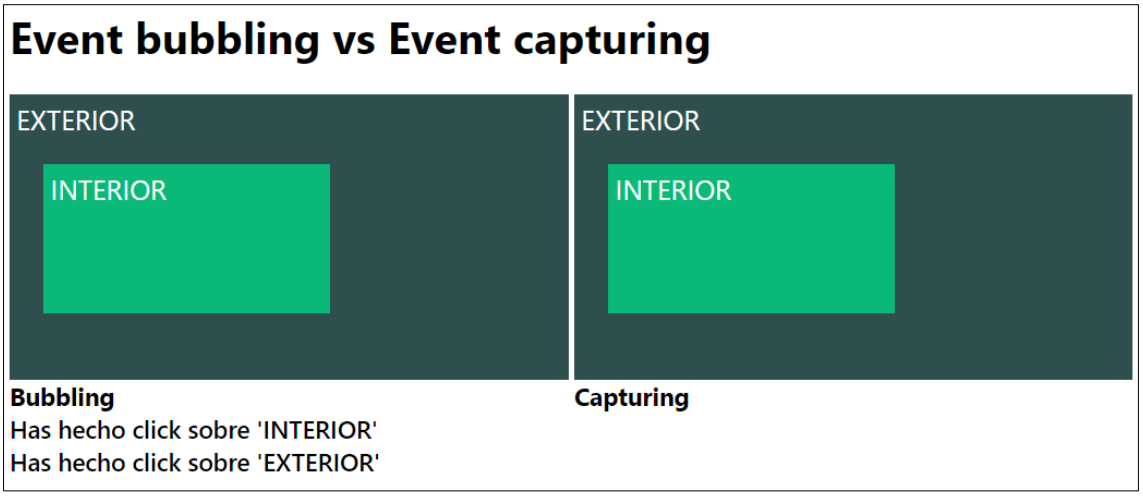

Figura 10. Ejemplo representativo del funcionamiento de Event Bubbling.

Y ahora veamos que ocurre si hacemos click en el recuadro pequeño de la derecha.

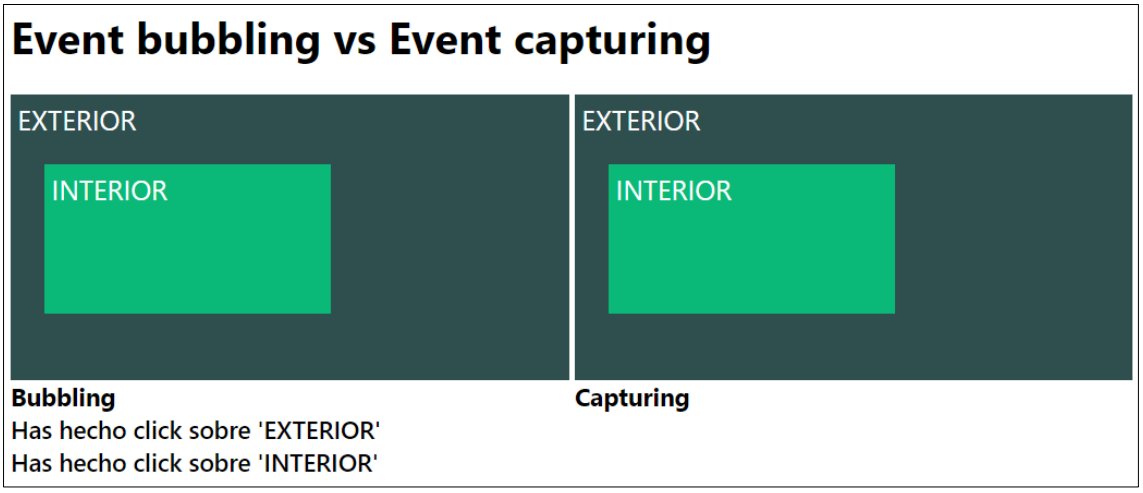

Figura 11. Ejemplo representativo del funcionamiento de Event Capturing.

Event Bubbling se utiliza si queremos que el evento suceda primero sobre el elemento en cuestión, y después sobre sus "antecesores". Event Capturing se utiliza si queremos lo contrario, es decir, que el evento ocurra en orden descendiente, primero sobre el "antecesor" y después sobre los "descendientes".

HOLA MUNDO (JAVASCRIPT)

Creamos el siguiente archivo index.html:

```
<!DOCTYPE html>
```

```
<html>
  <head>
    <meta charset="utf-8">
    <title>HOLA MUNDO</title>
  </head>
  <body>
    <h1></h1>
    <script src="scripts/main.js"></script>
  </body>
</html>
```

La etiqueta <h1> contendrá el texto “HOLA MUNDO”, que incluiremos mediante un script. En la ubicación donde tenemos el archivo index.html, creamos una carpeta llamada “scripts”, y dentro, un archivo “main.js” con el siguiente código:

```
const miTitulo = document.querySelector('h1');

miTitulo.textContent = 'HOLA MUNDO!';
```

Con este código, creamos una constante llamada “miTitulo” que apunta al contenido de la etiqueta <h1> del archivo index.html, y después editamos ese contenido para que aparezca “HOLA MUNDO!”. Para que esto funcione, tenemos que “llamar” desde el archivo html al script incluyendo la línea <script src="scripts/main.js"></script> justo antes del cierre </body>.

Esta línea debe ir al final del “body” porque el navegador va a cargar el archivo html en orden, y debe cargarse antes la etiqueta <h1> que el script. Si se cargase primero el script, no reconocería ninguna etiqueta <h1> y por lo tanto, no aparecería el texto.

Así, obtendremos el siguiente resultado:

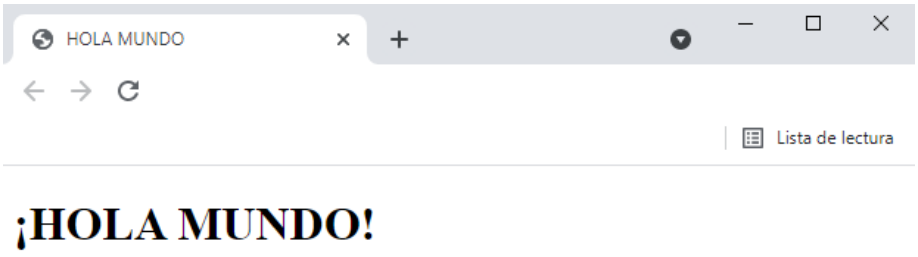

Figura 12. Página HTML con JavaScript de ejemplo “HOLA MUNDO”.

2.4. SQL

SQL (Structured Query Language), es un lenguaje diseñado para almacenar, manipular y recuperar datos de bases de datos relacionales. Para explicar esta tecnología, me basaré en el tutorial de w3schools de SQL [41]. Algunas aplicaciones de este lenguaje son:

- Ejecutar consultas en una base de datos.
- Recuperar datos de una base de datos.
- Insertar y actualizar registros en una base de datos.
- Eliminar registros de una base datos.
- Crear nuevas bases de datos.
- Crear nuevas tablas en una base de datos.
- Crear procedimientos almacenados en una base de datos.
- Crear vistas en una base de datos.
- Establecer permisos en tablas, procedimientos y vistas.

RDBMS (Relational Database Management System), es la base de SQL y de todos los sistemas de bases de datos modernos como MS SQL Server o MySQL. Los datos en RDBMS se almacenan en objetos de las bases de datos, llamados tablas.

2.4.1. TABLAS

Son objetos de las bases de datos donde se almacenan los datos en un RDBMS. Está formada por filas y columnas que almacenan entradas de datos relacionados.

Cada tabla se divide en campos. Por ejemplo, en la tabla creada para almacenar la información de los modelos en 'Greative', los campos son ModeloId, ModeloNombre, ModeloProducto, ModeloFabricante... Son las columnas, diseñadas para almacenar datos específicos.

Un registro es cada entrada individual de una tabla. En la tabla de modelos de 'Greative', los registros son los modelos almacenados, cada uno con su Id, nombre, fabricante... Son las filas.

Una base de datos puede contener una o más tablas, en función de los datos que se quieren almacenar.

ModId	ModNombre	ModProducto	ModFabricante	ModFecha	ModImagen
1	Egg Chair	Silla	Arne Jacobsen	16/07/1958	eggChair.jpg
2	LC4	Silla	Le Corbusier	12/10/1928	lc4.jpg
3	R8	Coche	Audi	...	...

Figura 13. Tabla SQL de ejemplo.

Este es un ejemplo de tabla. Cada columna (ModId, ModNombre...) son los campos, y cada fila son los registros.

DECLARACIONES SQL

Para llevar a cabo las diferentes acciones descritas anteriormente, se utilizan sentencias SQL. Estas sentencias no distinguen entre mayúsculas y minúsculas. Las más importantes son:

- SELECT. Extrae datos de una base de datos.
- UPDATE. Actualiza datos.
- DELETE. Elimina datos.
- INSERT INTO. Inserta datos nuevos en una base de datos.
- CREATE DATABASE. Crea una nueva base de datos.
- ALTER DATABASE. Modifica una base de datos.
- CREATE TABLE. Crea una nueva tabla.
- ALTER TABLE. Modifica una tabla.
- DROP TABLE. Elimina una tabla.

Create Database

Para crear una nueva base de datos:

```
CREATE DATABASE ProductosDB;
```

Drop Database

Para eliminar una base de datos existente.

```
DROP DATABASE ProductosDB;
```

Create Database

Para crear una nueva tabla:

```
CREATE TABLE dbo.Modelos (
    ModeloId int,
    ModeloNombre string,
    ModeloProducto string,
    ModeloFabricante string,
    ModeloFecha date,
    ModeloImagen string,
```

```
    ModeloVent string,
    ModeloInco string
)
```

Entre paréntesis, se especifican los campos con su nombre y el tipo de datos que van a almacenar.

Drop Table

Para eliminar una tabla existente.

```
DROP TABLE dbo.Modelos;
```

Alter Table

Para modificar una tabla existente. Dentro de este comando tenemos varias opciones, que son:

- ADD COLUMN. Para añadir un nuevo campo.
- ALTER COLUMN. Para modificar el tipo de datos de un campo.
- DROP COLUMN. Para eliminar un campo.

Por ejemplo:

```
ALTER TABLE dbo.Modelos
ADD COLUMN CampoEjemplo int;


ALTER TABLE dbo.Modelos
ALTER COLUMN CampoEjemplo date;


ALTER TABLE dbo.Modelos
DROP COLUMN CampoEjemplo;
```

Select

Selecciona datos de una base de datos y los devuelve en forma de tabla. Por ejemplo:

```
SELECT ModNombre, ModFabricante,
FROM dbo.Modelo;
```

Devolverá una tabla compuesta por los campos ModNombre y ModFabricante de la tabla dbo.Modelo.

Si quisiéramos seleccionar todos los campos de la tabla, escribiríamos SELECT *.

Además, se puede utilizar la sentencia SELECT DISTINCT si queremos sólo valores distintos (no repetidos).

Where

Se utiliza para extraer solo los registros que cumplen una condición específica. Por ejemplo:

```
SELECT ModNombre,

FROM dbo.Modelo

WHERE ModProducto = "Silla";
```

Devolverá sólo aquellos modelos cuyo campo ModProducto contenga 'Silla'.

Se pueden utilizar con los operadores:

- AND, para especificar varias condiciones que deben cumplirse a la vez.
- OR, para especificar varias condiciones, pero basta con que se cumpla una de ellas. Se puede utilizar el operador IN si vamos a especificar muchas condiciones.
- NOT, para especificar una condición que debe ser falsa para cumplirse.

Por ejemplo:

```
SELECT ModNombre,

FROM dbo.Modelo

WHERE ModProducto = "Silla" AND ModFabricante = "Arne Jacobsen";
```

Selecciona sólo aquellos modelos que sean sillas, y su fabricante sea Arne Jacobsen.

```
SELECT ModNombre,

FROM dbo.Modelo

WHERE ModFabricante = "Arne Jacobsen" OR ModFabricante = "Le Corbusier";
```

O, utilizando IN...

```
SELECT ModNombre,

FROM dbo.Modelo

WHERE ModFabricante IN ("Arne Jacobsen", "Le Corbusier");
```

Selecciona los modelos diseñados por Arne Jacobsen o Le Corbusier.

```
SELECT ModNombre,

FROM dbo.Modelo

WHERE NOT ModFabricante = "Arne Jacobsen";
```

Se extraen todos los modelos que no son de Arne Jacobsen.

Además, se pueden combinar estos operadores, por ejemplo:

```
SELECT ModNombre,

FROM dbo.Modelo

WHERE NOT ModProducto = "Silla" AND (ModFabricante = "Arne Jacobsen" OR ModFabricante = "Le Corbusier");
```

Se escogen los modelos que no son sillas, diseñados por Arne Jacobsen o por Le Corbusier.

Order By

Este comando se utiliza para ordenar el conjunto seleccionado. Se añade ASC o DESC al final según se quiera ordenar en sentido ascendente o descendente. Por ejemplo:

```
SELECT * FROM dbo.Producto

ORDER BY ProductoNombre ASC;
```

Se seleccionan todos los productos y se ordenan por su nombre alfabéticamente.

Insert Into

Se usa para insertar nuevos registros en una tabla. Hay dos formas de utilizarlo:

1. Especificando los nombres de las columnas y los datos que se insertarán.

   ```
   INSERT INTO nombre_de_tabla (columna1, columna2, columna3, …)
   VALUES (dato1, dato2, dato3, …);
   ```

   Por ejemplo, en mi tabla dbo.Modelo, para introducir un nuevo registro se debe especificar todas las columnas menos la de ID, que se genera automáticamente. Entonces sería:

   ```
   INSERT INTO dbo.Modelo (ModeloNombre, ModeloProducto, ModeloFabricante, ModeloFecha, ModeloImagen, ModeloVent, ModeloInco)
   VALUES ("Egg Chair", "Silla", "Arne Jacobsen", "1958-06-01", "eggchair.jpg", "Atractivo visual", "Muy caro");
   ```

2. Si se van a agregar valores en todos los campos (columnas) de la tabla, no hace falta especificar las columnas. A la hora de especificar los datos, es importante hacerlo en el mismo orden que el de las columnas de la tabla.

   ```
   INSERT INTO  nombre_de_tabla
   VALUES (dato1, dato2, dato3, …);
   ```

Update

La palabra clave UPDATE sirve para modificar registros. Por ejemplo:

```
UPDATE dbo.Modelo

SET ModeloInco=”Caro y pesado”

WHERE ModeloNombre=”Egg Chair”;
```

Así, cambiaríamos el inconveniente de “Egg Chair”, que previamente era “Muy caro”, por “Caro y pesado”. Es muy importante realizar esta acción con cuidado, ya que, si no se especifica la sentencia WHERE, el cambio afectará a todos los registros.

Delete

Sirve para eliminar registros. Por ejemplo:

```
DELETE FROM dbo.Modelo

WHERE ModeloFabricante=”Le Corbusier”
```

De este modo eliminamos todos los modelos cuyo fabricante sea Le Corbusier. Si no especificamos el comando WHERE, no se eliminará la tabla, pero sí todos los registros de esta.

RESTRICCIONES SQL

Se utilizan para especificar el tipo de datos que se va a almacenar en cada campo de una tabla, de tal manera que, si el dato no es del tipo especificado, no se almacena. Estas restricciones se pueden utilizar para cada columna, o para toda la tabla. Además de *int* (para datos numéricos enteros), *string* (cadenas de caracteres), *date* (para fechas), etc., se utilizan restricciones especiales, como las siguientes:

- NOT NULL. No se aceptan datos con valores “nulos”.
- UNIQUE. No se aceptan dos registros con valores iguales en un campo.
- PRIMARY KEY. Combinación de NOT NULL y UNIQUE, es decir, no se aceptan dos registros iguales en un campo, ni valores “nulos”.
- FOREIGN KEY. Se utiliza para que un campo apunte a algún campo que contiene PRIMARY KEY de otra tabla. Obliga a que el campo que contiene esta constante, almacene solamente valores que están en el campo con PRIMARY KEY de la otra tabla.
- CHECK. Se aceptan los datos si cumplen una condición.
- DEFAULT. Se establece un valor por defecto si no se especifica ninguno.
- CREATE INDEX. Para crear un índice para algún campo. Se utiliza para acelerar las búsquedas.

3 ANGULAR

3. ANGULAR

3.1. INTRODUCCIÓN

Angular es un framework de Google para diseño de aplicaciones de una sola página en el lado del cliente usando HTML y TypeScript. En una aplicación de una sola página (Single Page App (SPA)), el cliente o usuario hace una petición al servidor. El servidor la recibe y le responde enviándole el archivo html que se cargará en el navegador del cliente. Hasta aquí, el funcionamiento es igual que una aplicación web normal. Pero en una normal, cada vez que el usuario pulse en algún sitio o interactúe de alguna forma, se van enviando peticiones al servidor y el servidor responde con más archivos html. Sin embargo, en una SPA solo hay una petición inicial al servidor, una respuesta html del mismo, y ya no hay más archivos html respuesta. Se actualiza alguna parte. No se envían htmls por cada operación. Sobre el documento inicial, se muestran los nuevos elementos [33].

A continuación, en el punto 3.2., explicaré los conceptos básicos de Angular. Para ello, me basaré en los documentos de la página web oficial de Angular, referenciados en la bibliografía desde [11] hasta [22].

3.2. FUNDAMENTOS

3.2.1. MÓDULOS

Angular ofrece un sistema de módulos llamado NgModules. Estos módulos pueden contener componentes, proveedores de servicios u otros archivos. Además, pueden importar funcionalidades que otros módulos han exportado, y exportar sus funcionalidades para que otros módulos puedan utilizarlas.

Toda aplicación Angular tiene al menos un módulo, llamado AppModule y considerado el módulo raíz. Está contenido en un archivo llamado app.module.ts.

Metadatos de NgModule

Un NgModule se define mediante una clase con el decorador @NgModule(), una función que toma un único objeto de metadatos cuyas propiedades describen al módulo. Las más importantes son:

- imports. Aquí se mencionan aquellos módulos externos que necesitamos para los templates (plantillas html) del módulo que se está describiendo.
- providers. Los servicios que necesita el módulo y que estarán disponibles para toda la aplicación.
- declarations. Los componentes y directivas pertenecientes a este NgModule.

- exports. Aquí se mencionan aquellas declaraciones que queremos que sean accesibles para las plantillas de otros módulos.
- bootstrap. La vista principal de la aplicación. Esta propiedad sólo se define en el módulo raíz.

A continuación, una definición simple de módulo raíz, que es la que se establece por defecto al crear una aplicación mediante el CLI (Command-Line Interface, Interfaz de Línea de Comandos) de Angular.

```
// imports
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
// Decorador @NgModule con sus metadatos
@NgModule({
declarations: [AppComponent],
imports: [BrowserModule],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule {}
```

Con BrowserModule, se importa la infraestructura requerida para cualquier aplicación Angular. Se incluye por defecto en el módulo raíz cuando creamos una aplicación Angular con el CLI. AppComponent es el componente principal. Al crear un componente nuevo con el CLI de Angular, la clase de este componente se importa y se registra automáticamente en *declarations* del NgModule.

Los NgModules proporcionan un contexto de compilación para sus componentes. Un NgModule raíz (root) siempre tiene un componente principal que se crea durante el arranque de la aplicación, pero puede tener componentes adicionales. Todos ellos, comparten un contexto de compilación. A su vez, un componente puede contener una jerarquía de *vistas*, permitiendo definir áreas complejas que se pueden crear, modificar y eliminar como una unidad. Una jerarquía de *vistas* puede mezclar vistas definidas en componentes de NgModules diferentes.

Cuando se crea un componente, se le asocia una sola vista (*vista host*). Esta vista puede contener *vistas* incrustadas, que sean *vistas host* de otros componentes, pertenecientes al mismo NgModule o importados desde otros diferentes.

A continuación, un esquema de un ejemplo de jerarquía de vistas.

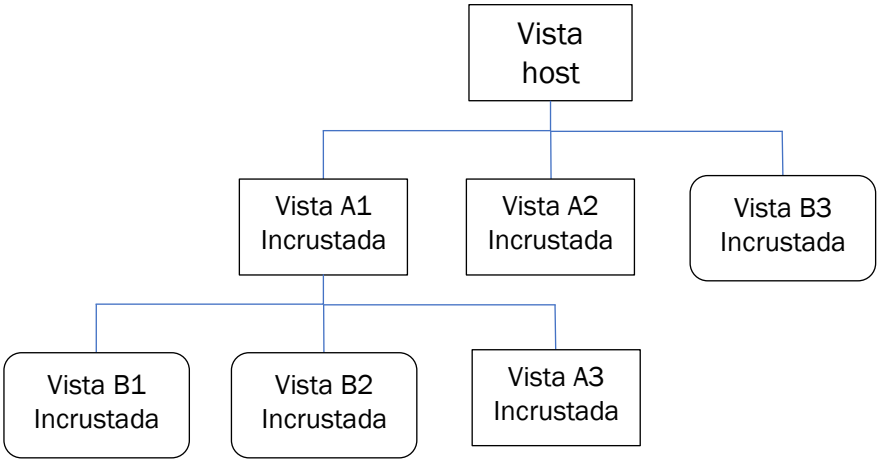

Figura 14. Jerarquía de vistas en Angular.

Librerías Angular

Como cualquier framework, Angular tiene una serie de librerías que podemos importar para facilitar el desarrollo del código, o para solucionar problemas concretos. Amplían las funcionalidades.

El *path* (ruta o ubicación) de las librerías Angular, empieza con el prefijo @angular. Para importar una de ellas, por ejemplo, la librería 'Component', se escribe lo siguiente en la parte superior de app.module.ts:

```
import { FormsModule } from '@angular/forms';
```

y se debe añadir en *imports*, de esta manera:

```
imports: [ FormsModule ],
```

Se puede consultar el funcionamiento y utilidad de cualquier librería en los documentos oficiales de la página web de Angular.

3.2.2. COMPONENTES

Un *componente* controla una parte de lo que el navegador muestra al ejecutar una SPA Angular. Es decir, controla una *vista*. El comportamiento del componente se define dentro de la *clase* definida en el archivo TypeScript, que interactúa con la *vista* para mostrar lo que se quiere mostrar. Angular crea, modifica y elimina componentes mientras el usuario utiliza la aplicación.

Metadatos de los componentes

El decorador @Component define los metadatos del componente, que son los que dicen a Angular dónde están los bloques principales para construir y presentar el

componente y su *vista*. Principalmente, asocia una *plantilla* con el componente, pero normalmente se referencia también, al menos, una hoja de estilos. Para ello:

```
@Component({
  selector:    '. . .',
  templateUrl: '. . .',
  styleUrls: ['. . .']
  . . .
})
export class nombreComponenteComponent implements OnInit {
/* . . . */
}
```

@Component, es el decorador, y contiene:

- El selector, es el nombre de la etiqueta HTML que se utilizará como instancia del componente en los *templates* de otros, de tal manera que, escribiendo la etiqueta del selector, aparecerá el contenido del componente al renderizar.
- templateUrl, es la ruta donde se encuentra la plantilla HTML que se va a mostrar.
- styleUrls, es la ruta donde se encuentra el archivo CSS de estilos, para definir la apariencia de los elementos de la plantilla.
- Se puede incluir más definiciones, como *animations*, pero estas tres anteriores son las más básicas.

templateUrl y styleUrls se pueden sustituir por *template* y por *styles*, si se quieren definir *inline,* es decir, en el decorador mismo. Nos podría interesar si contienen poco código. Se utilizarían así:

```
@Component({
  selector: 'app-prueba',
  template: `
    <h1>Template inline de prueba</h1>
    <p>Esto es un template definido de forma inline, incluido el estilo css.</p>
  `,
  styles: ['h1 { font-weight: normal; color: red; }']
})
```

Este modelo de componentes hace que Angular ofrezca una encapsulación sólida, además de una estructura bastante intuitiva. Los componentes facilitan el testeo de las aplicaciones, y la legibilidad del código.

Plantillas

Todo componente tiene una plantilla HTML donde se indica cómo se va a mostrar ese componente. Esta plantilla se puede definir, como hemos mencionado antes, dentro del decorador de forma *inline*, o haciendo referencia a un archivo HTML externo.

Para representar un componente, se utiliza su selector asociado. En el ejemplo del componente cuyo decorador @Component hemos visto antes, el selector es app-prueba. Si quisiéramos utilizar este componente, incluiríamos la etiqueta <app-prueba></app-prueba> en una plantilla HMTL de otro componente, allí donde queramos mostrarlo.

Figura 15. Ejemplo visual de cómo representar un componente en Angular.

Las plantillas de Angular son básicamente archivos HTML normales, pero pueden contener estructuras específicas de Angular para modificar el contenido según la lógica de la aplicación.

- Se puede usar *data binding* para enlazar la aplicación con los datos del DOM.
- Se pueden usar *pipes* para transformar datos antes de mostrarlos.
- Se pueden usar *directivas*, para agregar un comportamiento específico a lo que se muestra.

Por ejemplo:

```
<h2>Lista de la compra</h2>

<p><i>Elige un producto para ver los detalles.</i></p>

<ul>

  <li *ngFor="let producto of compra">

    {{producto.nombre}}

  </li>

</ul>
```

En este ejemplo se utiliza la directiva *ngFor para almacenar en una variable tipo *array* llamada 'producto' cada elemento con sus características, contenido en 'compra'. Después se utiliza *data binding* para representar en cada iteración el nombre de los productos.

Data Binding

Se usa para representar datos o definir propiedades de manera dinámica. Mediante *data binding*, podemos cambiar el DOM según las acciones que el usuario lleve a cabo. El siguiente diagrama muestra los cuatro tipos de *data binding* que hay.

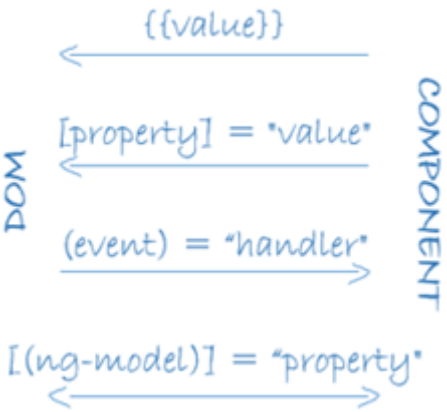

Figura 16. Esquema de los cuatro tipos de Data Binding.

Vamos a ver un ejemplo:

```
<li>{{producto.nombre}}</li>

<app-producto-detalles [producto]="productoElegido"></app-producto-detalles>

<li (click)="productoElegido(producto)"></li>
```

- {{producto.nombre}} es una interpolación. Muestra el valor de la propiedad *producto.nombre* especificada en la clase del componente actual.
- [producto]="productoElegido" es un ejemplo de *property binding*. Pasa el valor de *productoElegido* del componente actual, a la propiedad *producto* del componente 'descendiente' cuyo selector es app-producto-detalles.
- <li (click)="productoElegido(producto)"> es un ejemplo de *event binding*. Se llama al método *productoElegido( )* del componente actual cuando el usuario hace click sobre el elemento .

A continuación, un ejemplo de *binding bidireccional* o *two-way binding*.

```
<input type="text" id="producto-nombre"
[(ngModel)]="producto.nombre">
```

En este caso, el valor de la propiedad *producto.nombre* llega al *input* desde el componente (*property binding*). El usuario cambia el valor, llega de vuelta al componente, y se almacena ese valor en la propiedad (*event binding*).

Interpolación

En Angular, se pueden insertar valores dinámicos desde el componente. Al cambiar el valor del dato en la clase del componente, Angular actualiza ese dato en aquellas partes de la aplicación donde se ha llamado a ese valor dinámico.

Para poder crear una interpolación, se declara una propiedad en la clase del componente (dentro del archivo con extensión .ts del componente):

```
artista = 'René Magritte';
```

Y luego se "llama" a esa propiedad en el *template*.

```
<h3>El artista se llama: {{ artista }}</h3>
```

Al ejecutar la aplicación, la etiqueta h3 mostrará "El artista se llama: René Magritte".

Además, se pueden resolver expresiones dentro de la interpolación {{ }}. Por ejemplo {{ 1 + 1 }} muestra 2.

Property Binding

En la clase del componente añadimos propiedades y en el template, asociamos esas propiedades al objeto que queramos. *Property binding* es crear un vínculo entre las propiedades declaradas en la clase y el objeto incluido en el *template*. De tal forma que las propiedades del objeto pueden ser modificadas de forma dinámica, mientras la aplicación está en ejecución. Es un comportamiento básico de una aplicación Angular. Mientras el usuario va interactuando con la aplicación, pulsando en menús, introduciendo textos, pasando el ratón por encima de algún elemento, etc., podemos hacer que esas interacciones hagan que los objetos de la aplicación vayan cambiando. Por ejemplo:

```
<img [src]="imagen">
```

Mediante *property binding,* se está asociando el valor de la propiedad *src* del elemento *img*, con el valor de la propiedad 'imagen' declarada en la clase del componente actual.

Además, se puede utilizar entre componentes, como en el ejemplo mostrado antes, en el punto de *Data Binding*, en la línea:

```
<app-producto-detalles [producto]="productoElegido"></app-
producto-detalles>
```

Event Binding

Un evento es el desencadenante de una acción. Cuando trabajamos con Angular, creamos funciones dentro de la *clase* del componente, y son estas funciones las que se asocian a eventos dentro del *template*, para que se ejecuten una vez ocurra esos eventos.

Si queremos que aparezca una imagen en nuestra aplicación cuando el usuario haga click en un botón, el evento sería 'hacer click', el objeto que desencadena el evento

sería el botón, y la acción a ejecutar que estaría programada dentro de una función en la *clase* del componente sería mostrar la imagen.

Binding Bidireccional (Two-Way Binding)

El flujo de información irá tanto de la *clase* del componente al *template* como del *template* a la *clase* del componente. Permite que las aplicaciones se actualicen continuamente.

Directivas

Son instrucciones que vamos aplicando a los elementos *html* del *template*, que permiten añadir a esas etiquetas cierta funcionalidad, gracias a la cual modificamos la estructura del DOM. Una directiva es una *clase* con un decorador @Directive( ).

Un componente técnicamente es una directiva. Sin embargo, los componentes son protagonistas en una aplicación Angular, y por ello tienen su decorador propio @Component(), que extiende el decorador @Directive() con características del *template* asociado.

Existen también las *directivas estructurales* y las *directivas de atributos*. Angular dispone de una serie de directivas de estos dos tipos, y a mayores nos da la posibilidad de definir directivas propias mediante el decorador @Directive().

Las directivas estructurales, alteran el DOM añadiendo, eliminando o sustituyendo elementos. Por ejemplo:

```
<li *ngFor="let producto of compra"></li>

<app-producto-detalles *ngIf="productoElegido"></app-
producto-detalles>
```

La directiva *ngFor es iterativa. Se utiliza la directiva para almacenar en una variable llamada ‘producto’ cada elemento contenido en ‘compra’.

La directiva *ngIf es un condicional. Se utiliza cuando queremos mostrar o ocultar un elemento según el valor que tome una variable booleana. Si es *true*, se muestra. Si es *false*, se oculta. En este caso se muestra el componente asociado al selector <app-producto-detalles> si la variable productoElegido es *true*.

Las directivas de atributo, alteran la apariencia o el comportamiento de un elemento. Se utilizan como *atributos* de un elemento, de ahí el nombre.

La directiva ngModel se utiliza cuando necesitamos establecer un *binding bidireccional*. Modifica el comportamiento de un elemento existente definiendo el valor que se muestra, y variando en función de algún evento.

Pipes

Para desarrollar 'Greative', no he utilizado *pipes*, así que sólo explicaré el concepto. Básicamente, se utilizan para pasar un valor a una función antes de mostrarlo. Esta función se declara en un archivo aparte.

Se crean en el CLI de Angular, mediante el comando 'ng generate pipe *nombrePipe*'. Dentro del archivo, se crea la función deseada, con el correspondiente parámetro que va a recibir especificando el tipo. Por último, para utilizar este *pipe* en algún componente, se escribe {{ *valor* | *nombrePipe* }}.

3.2.3. SERVICIOS E INYECCIÓN DE DEPENDENCIAS

Los servicios son *clases* con un propósito limitado y bien definido. Deben hacer algo específico y hacerlo bien. Abarcan cualquier valor, característica o función que necesite una aplicación.

Angular diferencia entre componentes y servicios para potenciar la idea de modularidad y reutilización, y hacer que los componentes sean ágiles y eficientes. Para entender bien la idea de "inyección de dependencias", explicaré el concepto siguiendo una publicación de Enrique Oriol [29].

Los servicios realizan tareas como obtener datos de un servidor, validar *inputs* de usuarios... En el caso de la aplicación que he desarrollado, tuve que crear un servicio para obtener los datos de la base de datos. Definir estas tareas en una *clase de servicio inyectable*, nos permite disponer de ellas para cualquier componente.

El medio que tiene un componente para poderle indicar a Angular que necesita acceso a un servicio, es el constructor. Así, cuando Angular quiere instanciar un componente determinado, si en el constructor se pide por ejemplo "servicioA", Angular tiene que pasarle ese objeto para poder crear el componente.

Durante la fase de inicialización del proyecto, Angular crea un inyector para el módulo principal. Este inyector es un objeto que se encarga de crear y guardar las dependencias de clase de su módulo para inyectarlas cuando sea necesario. Instancia el servicio una vez, y luego comparte ese objeto con todos los componentes del módulo que lo necesiten. Pero el inyector necesita saber cómo obtener o crear la dependencia, y de esto se encarga el *provider*. Para cada dependencia que tengamos, necesitamos decirle a Angular cuál es su *provider*, para que el inyector pueda crear esas nuevas instancias. En los servicios, el provider suele ser la propia clase del servicio.

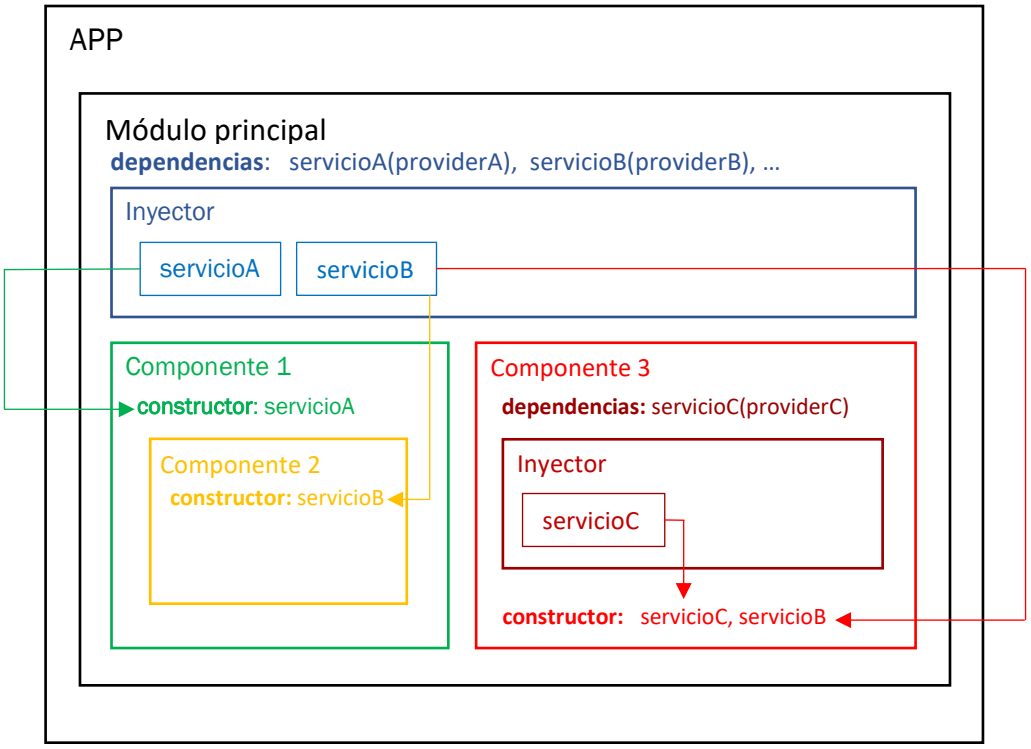

Figura 17. Ejemplo esquematizado del funcionamiento de la inyección de dependencias.

Veamos la aplicación representada en el esquema. Se construye en base a un módulo principal. Los metadatos del módulo importan dos *providers*, uno para el servicioA y otro para el servicioB. Angular crearía un inyector para todo el módulo, y dentro de ese inyector crearía una instancia para el servicioA y otra para el servicioB. Dentro del módulo principal tenemos el componente 1, que espera recibir en su constructor una instancia del servicioA. Esta instancia se la pasa el inyector del módulo principal. Esto es lo que se llama *inyección de dependencias* en Angular.

Dentro del Componente 1, tenemos el Componente 2, que espera recibir en su constructor una instancia del servicioB. De nuevo, es el inyector del módulo principal el que le pasa esta instancia. Además, tenemos el Componente 3 dentro del Componente 1. Este Componente 3, en este caso, también recibe un *provider* en sus metadatos, el del servicioC, que no está en el módulo raíz. Angular crea un inyector dentro del ámbito de ese componente y sus descendientes, e instancia nuevos objetos para sus dependencias, en este caso para el servicioC.

El Componente 3, solicita en su constructor el servicioB y el servicioC. La instancia del servicioC la recibirá de su propio inyector, mientras que la instancia del servicioB que recibirá es el del inyector del módulo principal. Es decir, la instancia del servicioB que recibirá es la misma instancia del servicioB que recibe el Componente 2.

Vamos a ver ahora un ejemplo real.

Creamos un archivo nuevo dentro del directorio app, y lo llamamos logger.service.ts. Dentro, escribimos lo siguiente:

```
export class LoggerService{
```

```
    log(msg:string){
        console.log(msg);
    }
    error(msg:string){
        console.error(msg);
    }
    warn(msg:string){
        console.warn(msg);
    }
}
```

Para usar este servicio en un componente, éste tiene que recibir una instancia del servicio por inyección de dependencias. Supongamos que vamos a utilizar el servicio en app.component. Abrimos app.component.ts, y en la parte superior escribimos lo siguiente para importar el servicio:

```
import { LoggerService } from ‘./logger.service’;
```

Para la inyección, pasamos la clase del servicio dentro del constructor, de esta manera:

```
export class AppComponent{
. . .
  constructor(private logger:LoggerService){
  }
. . .
}
```

Faltaría declarar el provider para que Angular sepa como crear este servicio. Para ello entramos en el módulo principal app.module.ts., importamos el servicio igual que lo hemos hecho en app.component.ts., y por último instanciamos el LoggerService dentro de *providers* en el NgModule, así:

```
@NgModule({
  . . .
  providers: [ LoggerService ],
  . . .
})
```

3.3. TYPESCRIPT

En este punto veremos el lenguaje TypeScript desde un punto de vista comparativo con JavaScript, siguiendo una publicación de César Alberca [1].

TypeScript es un lenguaje de programación basado en JavaScript, pero de tipado fuerte. Mediante JavaScript puedes utilizar tipos como *string* y *number*, pero no se comprueba que esos tipos no cambien.

Aunque JavaScript es un lenguaje de programación muy potente, para proyectos grandes tiene ciertas carencias.

- Se desconocen los tipos de datos que se están trabajando.
- Requiere una documentación exhaustiva del código para saber con qué estructuras de datos se está trabajando.
- Ofrece escasas ayudas.

TypeScript surge ante la necesidad de resolver los problemas que trae JavaScript.

En JavaScript podemos obtener ayudas que mejoran la experiencia durante su uso. Por ejemplo, si declaramos una constante (*const*) llamada 'prueba' que contiene "Esto es un string", por el hecho de usar las comillas, IntelliSense (la ayuda inteligente de Visual Studio) detecta que el dato es de tipo *string*. Si escribimos después 'prueba.', se despliegan todos los métodos que ofrecen las variables de tipo *string*.

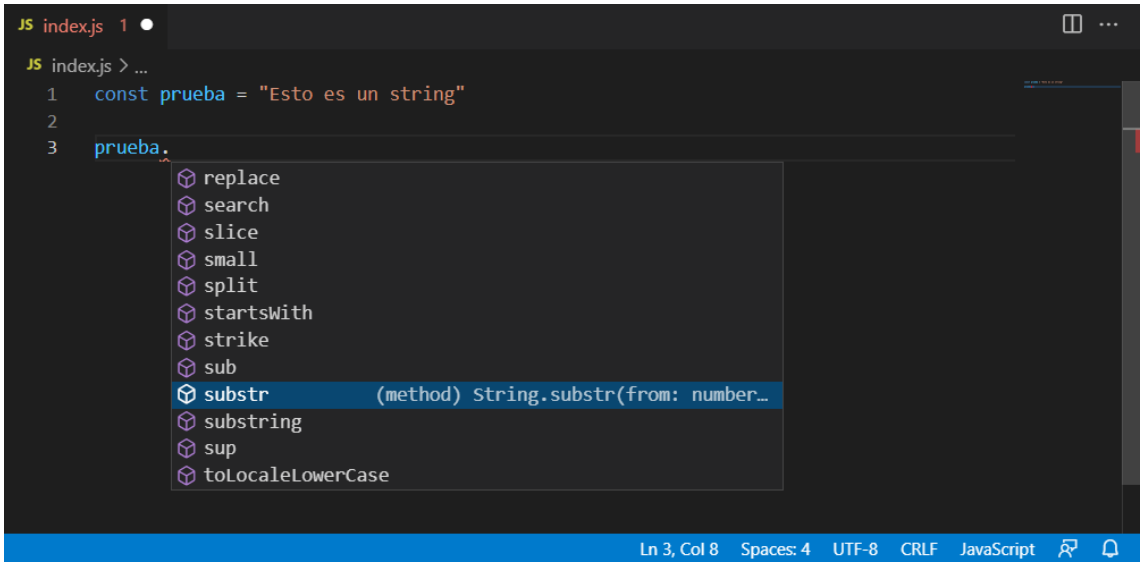

Figura 18. Demostración de IntelliSense en Visual Studio Code con JavaScript.

Si intentamos utilizar un método, por ejemplo, *substr*, con un atributo incorrecto, no nos va a informar de ningún error.

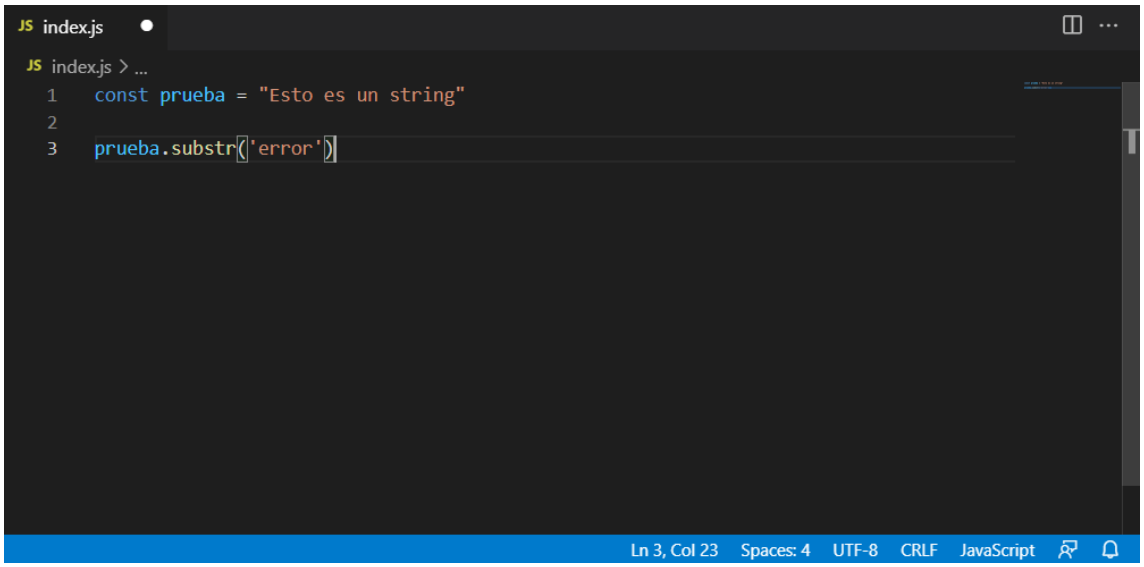

Figura 19. Demostración del tipado débil de JavaScript.

Esto se debe a que no sabe si en algún momento la variable *prueba* ha cambiado a otro tipo de dato, que cuente con el método *substr*. No nos informa de ningún error, pero dará un error en tiempo de ejecución.

Si realizamos exactamente el mismo experimento en TypeScript, veamos lo que pasa.

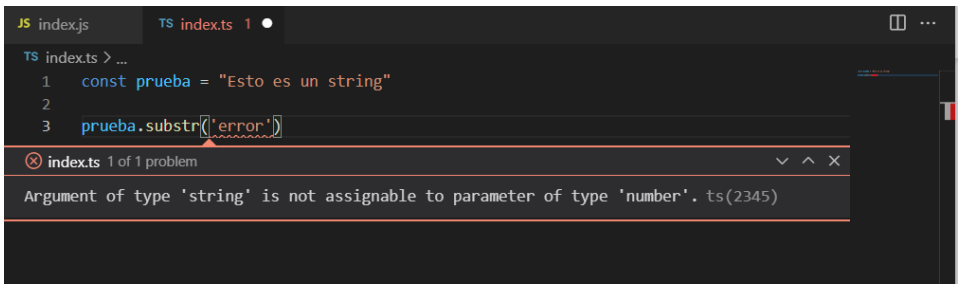

Figura 20. Demostración del tipado fuerte de TypeScript.

Aparece un error que nos informa de que no se puede utilizar un atributo de tipo *string* para ese método, sino que tiene que ser un dato de tipo *number*. Esto es porque TypeScript sabe que *prueba* no ha cambiado de tipo. Es deseable encontrarse estos errores a la hora de escribir el código en vez de en tiempo de ejecución, ya que de esta manera se puede solucionar el problema más rápida y eficazmente.

Esto es tan solo un ejemplo. A continuación, haré una comparación más exhaustiva.

En cuanto al tipado de datos al declarar variables.

Con JavaScript:

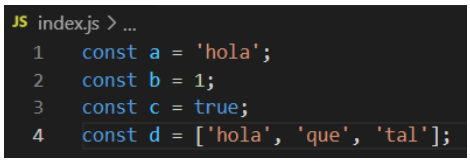

Figura 21. Declaración de variables en JavaScript.

Con TypeScript:

```ts
const a:string = 'hola';
const b:number = 1;
const c:boolean = true;
const d:string[] = ['hola', 'que', 'tal'];
```

Figura 22. Declaración de variables en TypeScript.

En cuanto al tipado de datos en funciones.

Con JavaScript:

```js
function suma(a, b = 1) {
    return a + b
}

const res = suma(2).toFixed(4)
console.log(res)
```

Figura 23. Funciones en JavaScript.

Con TypeScript:

```ts
function suma(a: number, b: number = 1): number {
    return a + b
}

const res = suma(2).toFixed(4)
console.log(res)
```

Figura 24. Funciones en TypeScript.

En cuanto a las interfaces.

Con JavaScript:

```js
const personas = [{
    nombre: 'Jose',
    edad: 25,
    trabajo: 'Diseñador'
    },
    {
    nombre: 'Alberto',
    edad: 32
    }
]
```

Figura 25. Interfaces en JavaScript.

Con TypeScript:

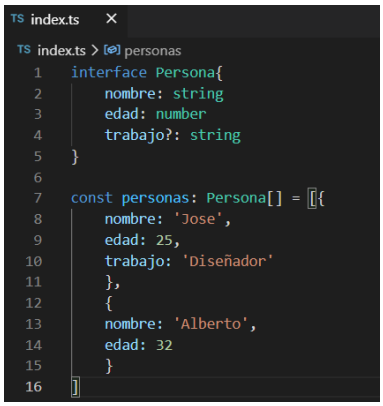

Figura 26. Interfaces en TypeScript.

Como vemos, en TypeScript se puede crear una “interface” donde especificar el tipo de datos que va a almacenar un array. En el campo “trabajo” de la “interface”, vemos que hemos añadido un “?”. Esto quiere decir que no es necesario especificar este campo, y nos va a permitir no recibir un error al no incluir el trabajo de ‘Alberto’.

En cuanto a las clases.

Con JavaScript:

Figura 27. Clases en JavaScript.

Con TypeScript:

Figura 28. Clases en TypeScript.

TypeScript además nos informará de un error si intentamos llamar a un método de la clase que no existe. La desventaja de TypeScript es que añade complejidad a JavaScript a la hora de configurar un proyecto. Pero realmente, TypeScript parte de JavaScript. Sólo añade el tipado fuerte.

3.4. CREACIÓN DE UNA APLICACIÓN EN ANGULAR

Una vez instalado Angular (proceso explicado en el punto 5.1.6.), podemos crear una aplicación siguiendo los siguientes pasos:

1. Accedemos a la ubicación donde queremos almacenar la aplicación.
2. En la barra de dirección del explorador de archivos escribimos “cmd” y pulsamos intro.
3. Tecleamos “ng new *nombreAplicación*” y pulsamos intro.
4. Nos pregunta si queremos agregar a la aplicación un chequeo más estricto, que va a permitir que sea más fácil de analizar estáticamente. Pulsamos la tecla “y” (yes) y pulsamos intro.
5. Después nos dice si queremos que agregue automáticamente las rutas de Angular. Pulsamos “y” y luego intro.
6. Por último, nos pregunta qué tipo de estilos queremos agregar a la aplicación, y sale por defecto CSS. Pulsamos “y” y seguido intro.

Se habrá creado una carpeta en el destino elegido, con el nombre de la aplicación. Dentro encontramos todos los archivos propios de una aplicación Angular.

3.4.1. CÓMO ARRANCAR Y APAGAR EL SERVIDOR

Abrimos la consola dentro de la carpeta de la aplicación. Escribimos 'ng serve –open', arranca el servidor y se abre automáticamente la aplicación. Si ahora tecleamos *ctrl + c*, nos pregunta: "¿Desea terminar el trabajo por lotes (S/N)?". Pulsamos "s", y se cierra el servidor. Si actualizamos la ventana del navegador donde está abierta la aplicación, vemos que no encuentra nada, ya que el servidor no está funcionando.

3.4.2. HOLA MUNDO (ANGULAR)

Podemos crear un "Hola Mundo" sencillo en Angular, mediante interpolación. Declaramos dentro de la clase TypeScript de app.component la siguiente variable.

```
let texto = '¡HOLA MUNDO!'
```

Y dentro de la plantilla HTML de este componente escribimos:

```
<h1>{{ texto }}</h1>
```

Para consultar cómo está definido el selector, podemos entrar en app.component.ts, y fijarnos en el decorador @Component. Si lo hacemos, veremos que por defecto, el selector del componente principal app.component es <app-root></app-root>. Por último, en el index.html, dentro del <body>, escribimos este selector.

```
...
  <body>
    <app-root></app-root>
  </body>
...
```

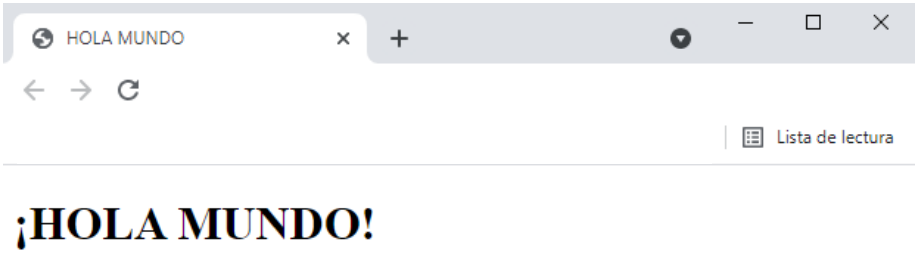

Figura 29. Página de ejemplo en Angular, 'HOLA MUNDO'.

4 GREATIVE

4. GREATIVE

Se trata de un cuaderno virtual de apuntes para un diseñador, donde incluir sus ideas y subir aquellos modelos que le aporten algún tipo de inspiración.

4.1. ANTECEDENTES

Uno de los objetivos principales de este proyecto, era estudiar a fondo el proceso de desarrollo de aplicaciones, y concretamente, en el entorno Angular. Para ello, busqué información al respecto, la cual derivó en el estudio de todo lo explicado anteriormente.

Esta aplicación está inspirada en otra, desarrollada y explicada en un vídeo de youtube, que consiste en una plataforma para registrar empleados en los diferentes departamentos de una empresa [2]. El vídeo se titula “Learn Angular 10, Web API & SQL Server by Creating a Web Application from Scratch”, y el creador es “Art of Engineer”. En él, se explica tanto la parte del *backend* como la parte del *frontend*, aunque lo que realmente me ha sido útil es la parte del *backend*.

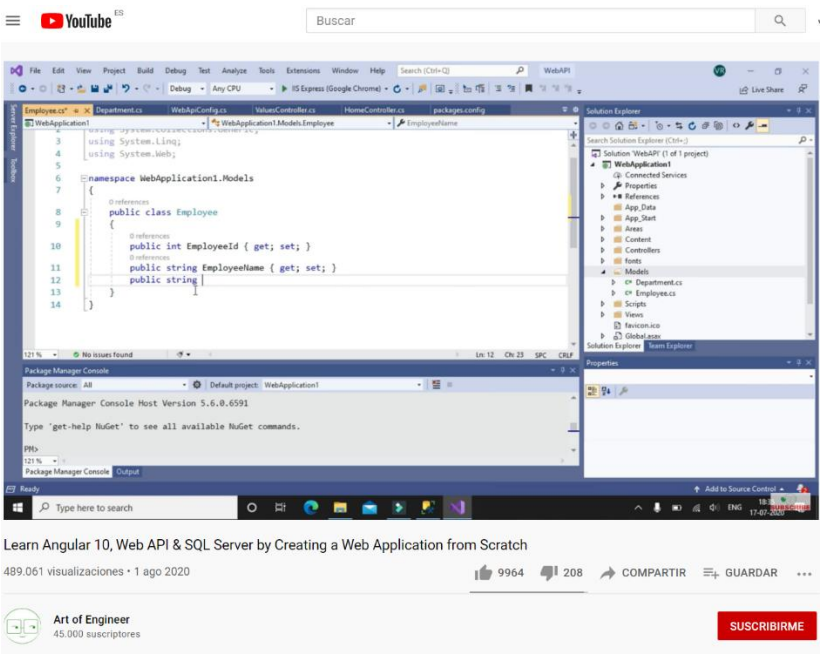

Figura 30. Vídeo de youtube en el que se desarrolla una aplicación básica.

4.2. REQUISITOS

‘Greative’ debe cumplir una serie de funcionalidades para resultar útil. El usuario de ‘Greative’ puede:

- Visualizar los productos que tiene registrados.

- Visualizar los modelos que tiene registrados.
- Añadir un producto nuevo del cuál quiere hacer una investigación. Los campos que debe rellenar se detallan en el manual de usuario.
- Modificar y eliminar productos existentes.
- Añadir un modelo, asociado a un producto previamente registrado, del cual quiera destacar alguna característica para un futuro diseño propio. Los datos que debe rellenar se detallan en el manual de usuario.
- Modificar y eliminar modelos existentes.
- Visualizar los modelos asociados a un producto específico.

4.3. DISEÑO

El logo hace referencia a la 'G' de 'Greative', que mezcla los términos 'great' y 'creative'. Literalmente, podría significar 'gran creativo', o 'genial creativo'.

La figura 30 muestra un esquema básico del flujo de peticiones entre el usuario, la aplicación, la base de datos, y el servidor.

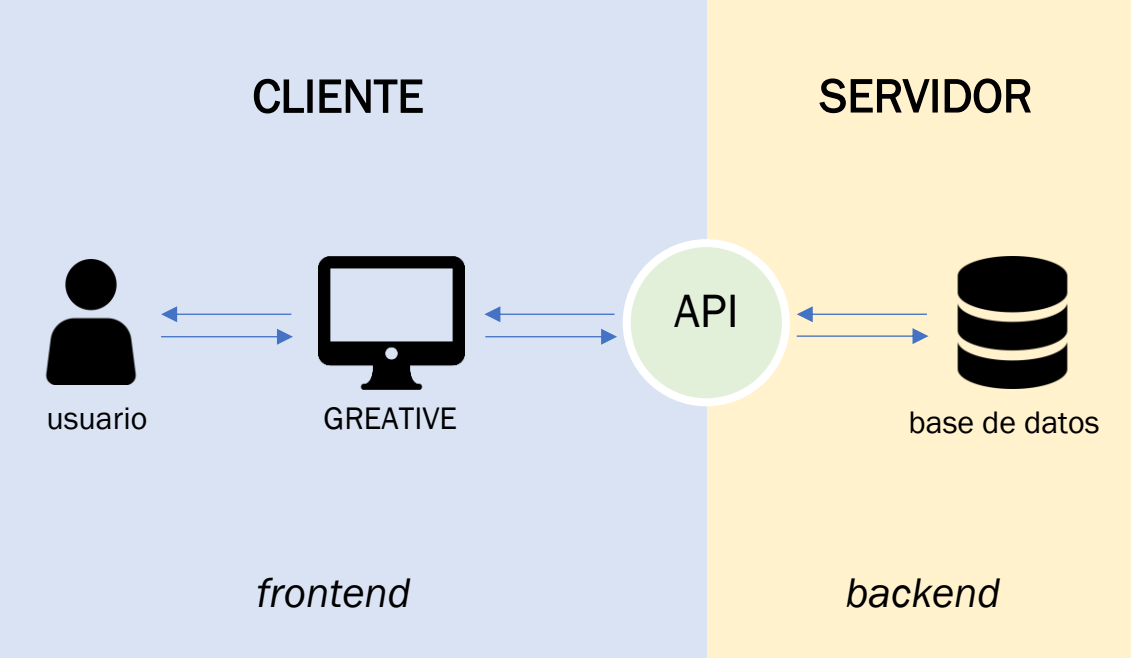

Figura 31. Diagrama de flujo de peticiones en una aplicación.

La figura 31 muestra un esquema con los componentes que forman el módulo principal (y único en este caso) de la aplicación. Cada uno de los componentes de este esquema debe entenderse como el conjunto de archivos que lo forman, es decir, la plantilla, la clase TypeScript y la hoja de estilos CSS. El SharedService mostrado, contiene las funciones creadas para cada método de la API que accede a la base de datos. Este SharedService se inyecta en los componentes indicados, que son los que van a necesitar consumir la API para realizar las peticiones.

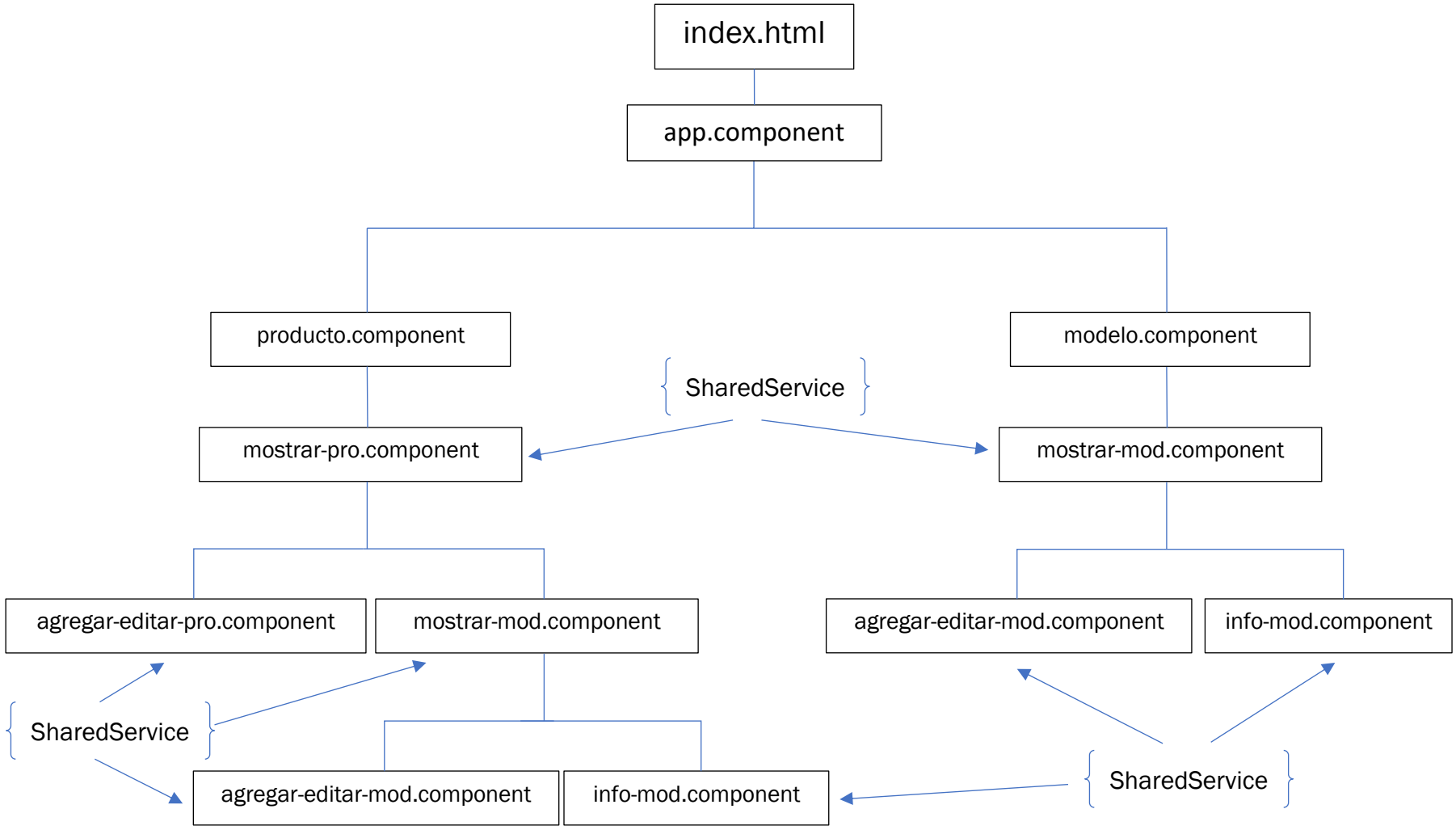

Figura 32. Esquema de componentes Angular de 'Greative'.

4.4. PROGRAMACIÓN

En cuanto a la parte del *backend,* una vez instalado el servidor para la base de datos en SQL Server Management Studio, se ha creado una base de datos que almacenará una tabla para los productos, y otra para los modelos.

La tabla de productos contiene los campos ProductoId, ProductoNombre y ProductoApuntes. ProductoId se completa automáticamente, ProductoNombre almacena el nombre del producto registrado y ProductoApuntes almacena los apuntes que se quieran hacer de ese producto, en el caso de que se quiera rellenar este campo en el formulario.

La tabla de modelos contiene los siguientes campos:

- ModeloId. De igual manera que en ProductosId, este campo se rellena automáticamente al efectuar el registro del modelo.
- ModeloNombre. Este campo almacena el nombre del modelo registrado.
- ModeloProducto. Registra el producto al que está asociado el modelo.
- ModeloFabricante. Contiene el fabricante o marca del modelo.
- ModeloFecha. Contiene la fecha del diseño del modelo. De manera opcional, se puede registrar la fecha en la que se registró el modelo en la aplicación, o incluso, no especificar ninguna.
- ModeloVent. Aquí se almacenan las características o detalles positivos del modelo.
- ModeloInco. Registra las características o detalles negativos del modelo.

Por otro lado, se ha utilizado ASP.NET Web Application de Microsoft Visual Studio para crear la API Web que accede a la base de datos. Se han establecido los métodos GET, POST, PUT y DELETE, para obtener, registrar, modificar y eliminar datos de la base de datos, respectivamente.

Por otro lado, en cuanto al *frontend*, 'Greative' está formada únicamente por un módulo, el principal, que se ha generado al crear la aplicación mediante el CLI. Este módulo está formado por un componente principal, app.component, que también se ha generado automáticamente al crear la aplicación. Además, app.component contiene dos componentes, producto.component y modelo.component. A su vez, producto.component contiene dos subcomponentes, mostrar-pro.component y agregar-editar-pro.component. Y modelo.component contiene otros tres subcomponentes, mostar-mod.component, agregar-editar-mod.component e info-mod.component. Esta estructura de componentes podemos observarla en la figura 31.

El componente principal app.component, alberga el código que define la barra superior de navegación. Los componentes producto.component y modelo.component, solamente contienen el selector del componente mostrar-pro.component y mostrar-mod.component respectivamente.

El componente mostrar-pro.component se utiliza cuando hacemos click en “Productos” en la barra de navegación. Contiene el código para obtener (gracias al servicio *SharedService* inyectado) los productos de la base de datos, y para mostrarles en pantalla de manera visualmente atractiva. Además, gracias a la hoja de estilos, los productos se distribuyen por la pantalla del dispositivo en función de la resolución de la misma. Dentro del componente mostrar-pro.component, se utilizan los selectores de los componentes agregar-editar-producto.component y mostrar-mod.component. El primero, para abrir la ventana modal con el formulario para agregar o editar un producto (según hagamos click en el botón de agregar o de editar un producto). El segundo, se utiliza cuando hacemos click en un determinado producto, para mostrarnos los modelos asociados a éste.

El componente mostrar-mod.component, se utiliza cuando hacemos click en “Modelos” en la barra de navegación, pero también cuando hacemos click en un producto, como he comentado antes. Si hacemos click en “Modelos” en la barra superior, se carga este componente, que obtendrá los modelos de la base de datos (gracias al servicio *Shared Service* inyectado) y los mostrará en pantalla, de una manera similar a cómo se mostraban los productos. En el código del componente mostrar-mod.component, se llama a otros dos componentes, agregar-editar-mod.component e info-mod.component. Si hacemos click en el botón de agregar un modelo nuevo, o en el de editar un modelo existente, se accede al componente agregar-editar-modelo.component, y se muestra el formulario con los campos necesarios para definir el modelo, detallados en el manual de usuario. Por otro lado, si hacemos click en la “bombilla” de uno de los modelos, se accede al componente info-mod.component, que abre una ventana con la información del modelo en cuestión, sin posibilidad de editar los campos.

Aparte, se ha creado un *servicio* Angular llamado *SharedService* (mencionado antes) para declarar las funciones que acceden a los métodos de la API, que es el que se inyecta en los componentes indicados en la figura 31.

4.5. PRUEBAS

Como en cualquier otro tipo de desarrollo que suponga escribir código, ‘Greative’ ha sido un largo proceso de ensayo y error. Bien es cierto que al principio me encontraba con más errores que al final, cuyos arreglos ahora me resultarían obvios. Por ejemplo, al principio me topaba con errores en Angular a la hora de declarar variables, El motivo era simple, Angular utiliza TypeScript, y esto requiere especificar el tipo de dato que va a almacenar la variable, pero en su momento, me costó entender el motivo.

Otro problema que tuve fue relacionado con la visualización de modelos de un producto específico. Al hacer click sobre, por ejemplo, el producto “Silla”, conseguí que se mostrasen los diferentes modelos de silla registrados. El problema era que para salir de esa “vista” solo podía hacer click en “Modelos”. Si pulsaba en

‘Productos’, no ocurría nada. Esto era porque la vista de estos modelos específicos se cargaba dentro de la ruta de ‘Productos’, ya que se cargaba mediante el subcomponente mostrar-mod, pero “llamado” dentro de mostrar-pro (explicados en el punto 4.3.). Finalmente se ha solucionado haciendo que desaparezcan “Productos” y “Modelos” de la barra de navegación, y que aparezca un botón para volver atrás, es decir, a “Productos”.

4.6. MANUAL DE USUARIO

La interfaz es bastante intuitiva. Básicamente está formada por dos rutas principales, que son “Productos” y “Modelos”, donde se presentan los productos y modelos subidos respectivamente.

En “Productos”, aparecen todos los productos que el diseñador ha creado, así como un botón para crear más. Además, encontramos dos botones para cada producto que nos permiten editar o eliminar el producto en cuestión.

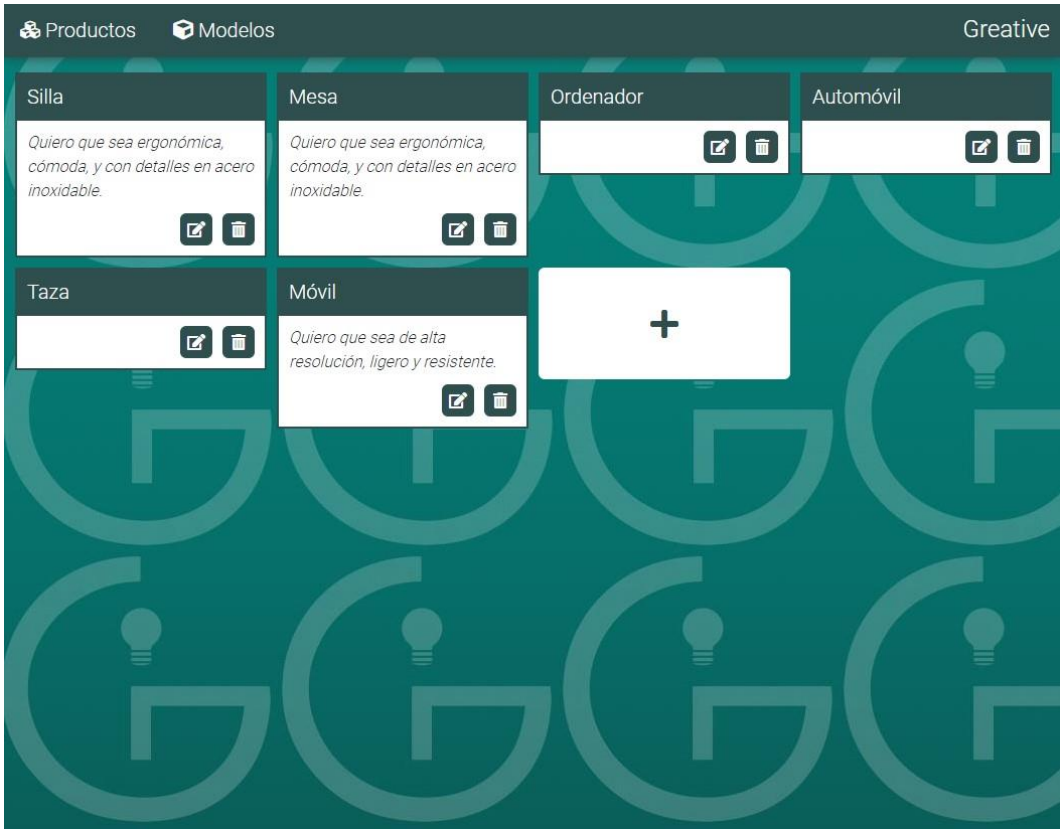

Figura 33. Apartado ‘Productos’, Greative.

Tanto para agregar como para modificar un producto, aparece una ventana emergente con el mismo formulario, en el que nos piden el nombre del producto (campo requerido, si no, salta un error), y unos apuntes opcionales.

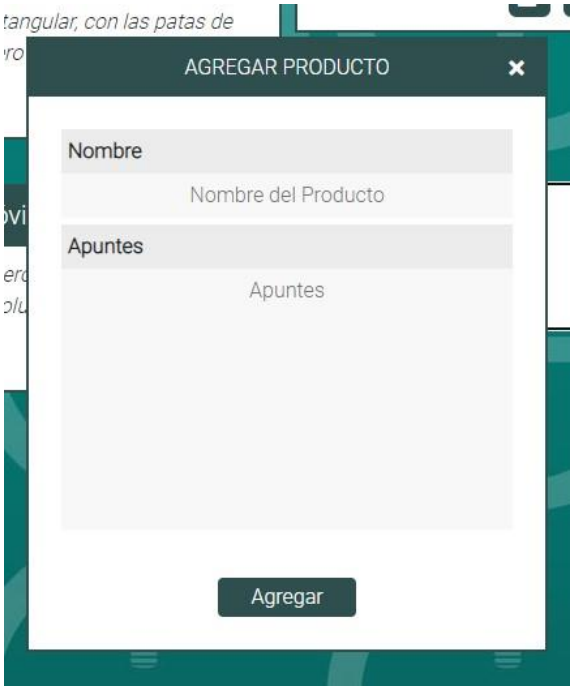

Figura 34. Ventana agregar producto, Greative.

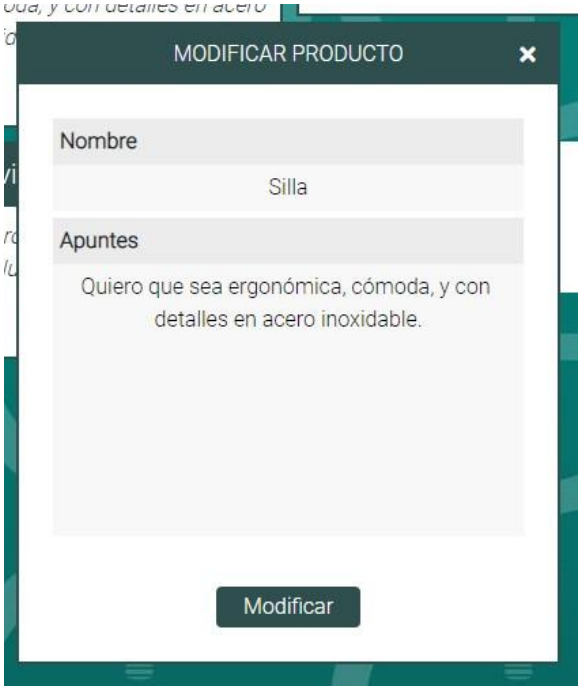

Figura 35. Ventana modificar producto, Greative.

En el apartado “Modelos”, aparecen todos los modelos, de todos los productos, incluido un botón para añadir más modelos.

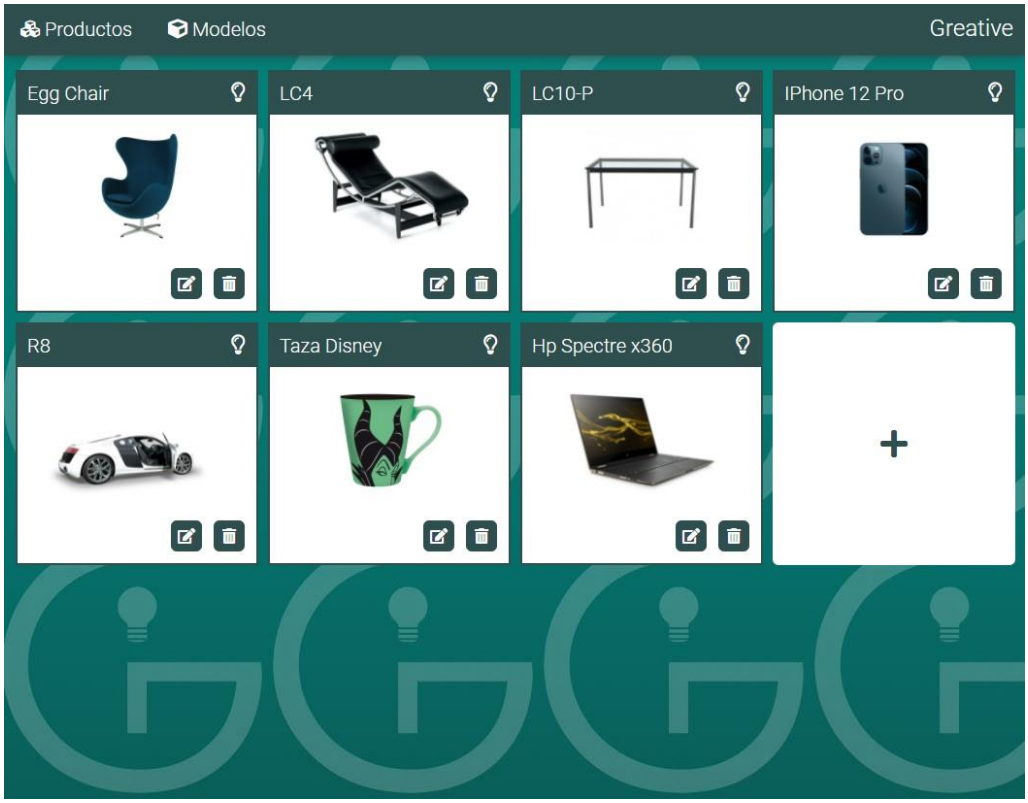

Figura 36. Apartado 'Modelos', Greative.

Para crear un modelo, se debe rellenar un formulario en el que se pide el nombre del modelo, el tipo de producto, el fabricante, una fecha (puede ser del diseño, o fecha en la que se suba el modelo, por ejemplo), una imagen, y ventajas e inconvenientes.

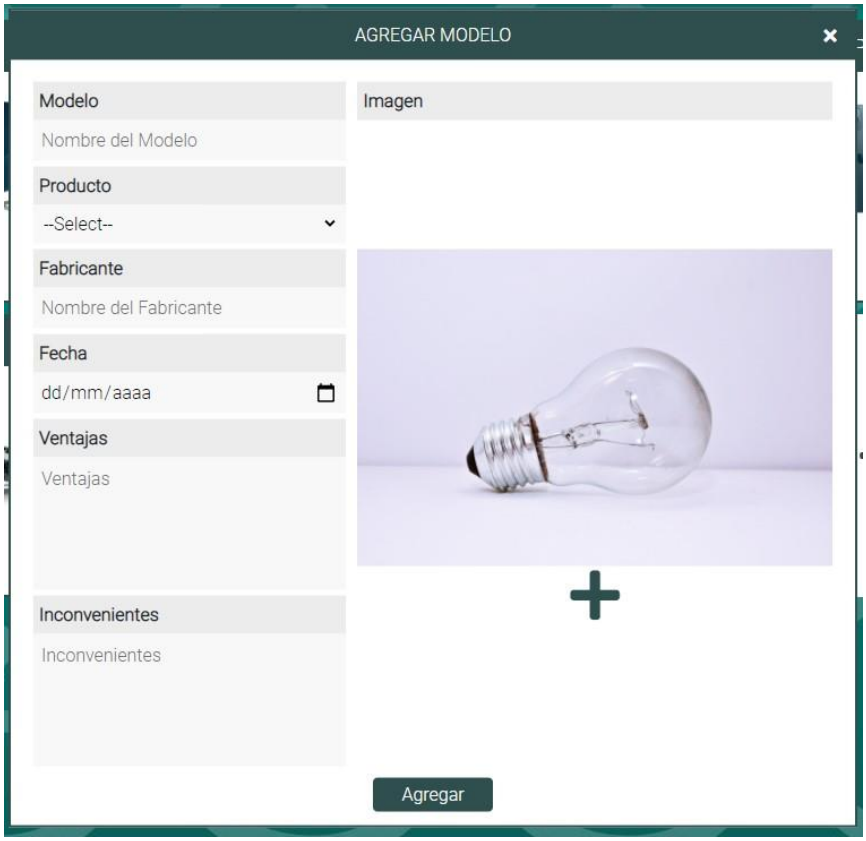

Figura 37. Ventana agregar modelo, Greative.

Si pulsamos en editar, aparece un formulario exactamente igual al de añadir modelos, pero rellenado con los datos que están asignados en ese momento.

Si pulsamos en la bombilla, aparece una ventana con la información introducida al crear el modelo.

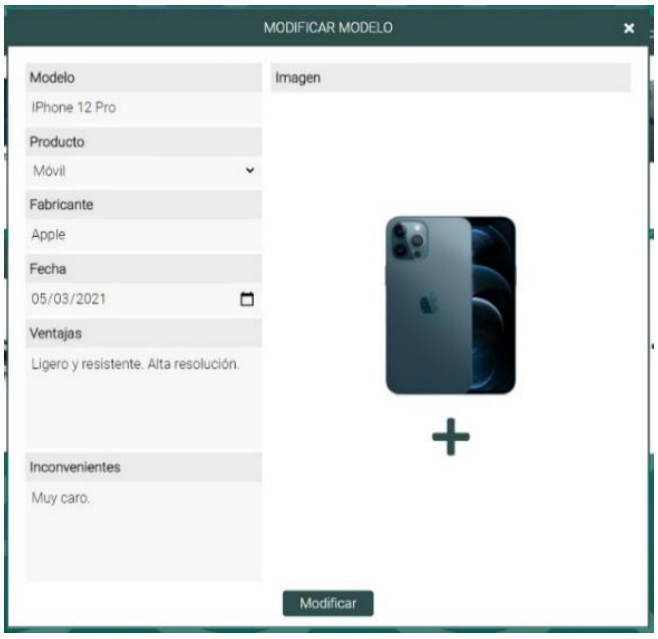

Figura 38. Ventana modificar modelo, Greative.

Figura 39. Ventana información modelo, Greative

Además, si queremos visualizar sólo los modelos correspondientes un producto concreto, por ejemplo 'silla', podemos hacer click en 'Silla', y aparecerá lo siguiente.

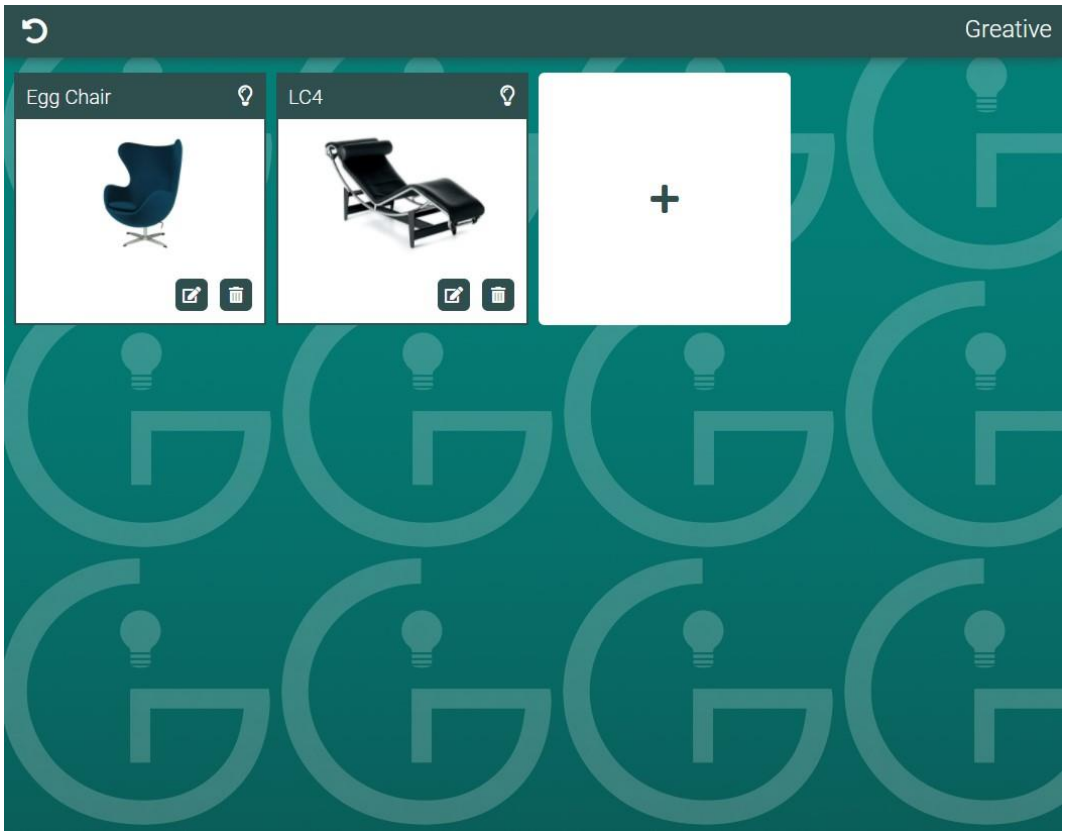

Figura 40. Visualización de modelos específicos, Greative.

Este es el funcionamiento de la aplicación. Además, se ha tenido en cuenta la resolución de cualquier dispositivo, ya que incluye un diseño *responsive*.

A continuación, unas capturas de cómo se vería la aplicación en un dispositivo con resolución 375 x 812.

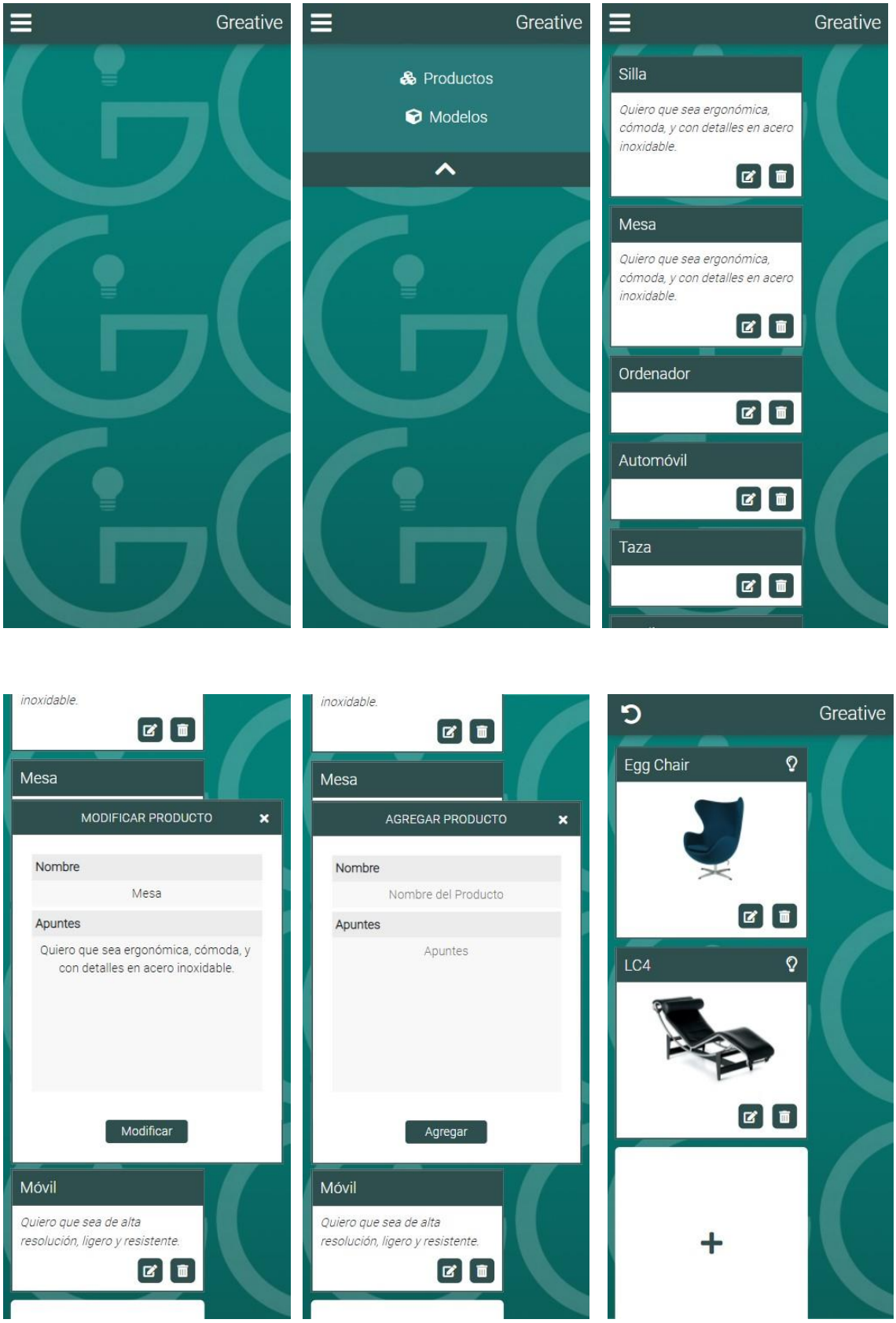

Figura 41. Diseño Responsive, Greative. (I)

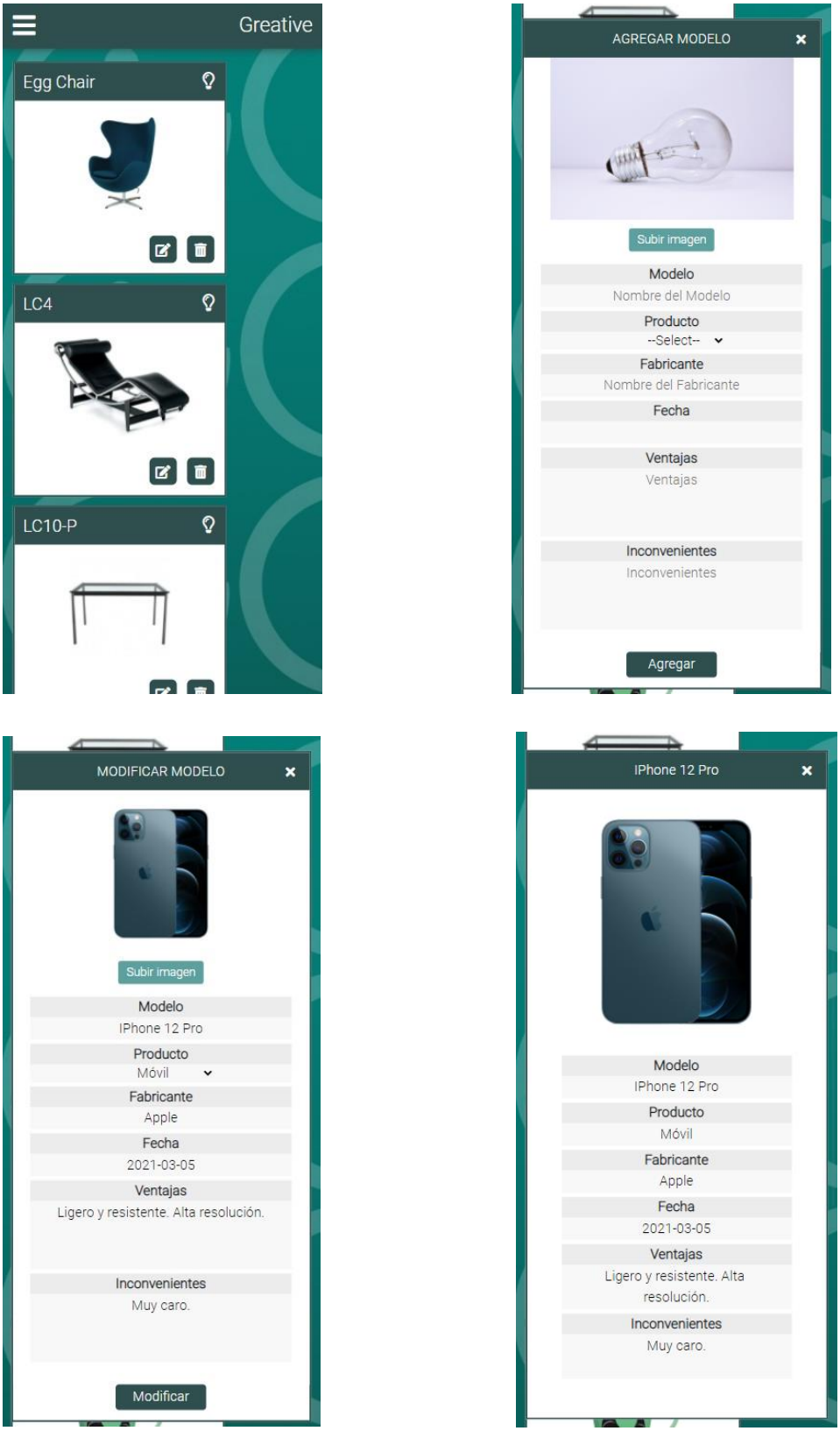

Figura 42. Diseño Responsive, Greative. (II)

5 INSTALACIONES Y DESPLIEGUE

5. INSTALACIONES Y DESPLIEGUE

5.1. INSTALACIONES

A continuación, explicaré los pasos a seguir para instalar las herramientas necesarias que he utilizado para desarrollar 'Greative', siguiendo el vídeo de *Art of Engineer* mencionado en la bibliografía [2].

5.1.1. INSTALACIÓN DE SERVIDOR SQL

Un servidor SQL se utiliza para crear bases de datos donde se almacenará la información recibida desde una aplicación. Utilizaré el que ofrece Microsoft.

Para la instalación del servidor SQL:

1. Escribimos en el buscador de Google, "Instalar sql server".
2. Hacemos click en el enlace de la página oficial de Microsoft.
3. En este caso, nos interesa obtener la edición gratuita de "Desarrollador". Hacemos click en "Descargar ahora".
4. Ejecutamos el archivo de instalación como administrador.
5. Hacemos click en "Custom".
6. Dejamos la ruta de instalación por defecto.
7. Se abrirá el "Centro de instalación de SQL Server". Accedemos a "Instalación" en el panel de la izquierda, y hacemos click en "Nueva instalación independiente de SQL Server o agregar características a una instalación existente".
8. Vamos dejando todas las opciones por defecto, hasta llegar a "Selección de características". Aquí solo interesa marcar la casilla "Servicios de motor de base de datos".
9. Continuamos, hasta llegar a "Configuración del motor de base de datos". Marcamos "Modo mixto", creamos una contraseña de administrador para el servidor SQL, y hacemos click en "Añadir usuario actual".
10. Continuamos y finalizamos la instalación.

Una vez instalado el servidor SQL, necesitamos una herramienta para la administración y el mantenimiento de este. Utilizaremos "SQL Server Management Studio (SSMS)", que es la que nos ofrece Microsoft.

Para instalar esta herramienta:

1. Desde el "Centro de instalación de SQL Server", hacemos click en "Instalar las herramientas de administración de SQL Server".

2. Se abrirá el navegador, que nos redirigirá al enlace para descargar la herramienta. Hacemos click en “Descarga de SQL Server Management Studio (SSMS)”.
3. Ejecutamos el archivo de instalación como administrador, e instalamos la herramienta.

Para comprobar que lo hemos instalado correctamente, vamos a abrir el SSMS y accederemos al servidor SQL.

Para ello:

1. Abrimos SSMS.
2. En “Server type” fijamos “Database Engine”; en “Server name” escribimos un punto “.”; y en “Authentication”, fijamos “Windows Authentication”. Hacemos click en “Connect”.

Se despliega un panel con una serie de carpetas y archivos que se crean por defecto en un servidor SQL.

5.1.2. INSTALACIÓN DE VISUAL STUDIO

Utilizaremos Microsoft Visual Studio para crear una “API”, una interfaz que permite la comunicación entre la base de datos y la aplicación. Para la instalación:

1. Escribimos en el buscador de Google “Instalar Visual Studio”.
2. Hacemos click en el enlace de la página oficial de Microsoft.
3. Descargamos la opción gratuita “Comunidad”.
4. Ejecutamos el archivo de instalación como administrador.
5. Marcamos las opciones “Desarrollo de ASP.NET y web”, “Desarrollo de escritorio de .NET” y “Desarrollo multiplataforma de .NET Core”. Hacemos click en “Instalar”.

5.1.3. INSTALACIÓN DE VISUAL STUDIO CODE

Visual Studio Code es un programa gratuito de Microsoft, editor de código fuente. Con esta herramienta implementaremos código para hacer aplicaciones.

Para la instalación:

1. Escribimos en el buscador de Google “Instalar Visual Studio Code”.
2. Hacemos click en la primera página encontrada, que es la oficial de Microsoft.
3. Descargamos la opción correspondiente a nuestro sistema operativo y procesador.
4. Ejecutamos el archivo de instalación como administrador.
5. Dejamos las opciones por defecto y finalizamos la instalación.

5.1.4. INSTALACIÓN DE POSTMAN

Esta herramienta nos servirá para probar los “métodos” que vamos a crear para la “API”.

Para la instalación:

1. Escribimos en el buscador de Google “Instalar Postman”.
2. Hacemos click en la primera página encontrada, que es la oficial.
3. Descargamos el programa haciendo click en “Download the App”.
4. Ejecutamos el archivo de instalación como administrador.
5. Una vez terminada la instalación, se abrirá el programa directamente. Nos aparecerá un formulario para crear una cuenta. No es necesario hacer esto. Podemos hacer click abajo en “Skip”.

5.1.5. INSTALACIÓN DE NODE JS

1. Escribimos en el buscador de Google “Instalar Node js”.
2. Hacemos click en la página oficial.
3. Descargamos la opción correspondiente al sistema operativo y procesador.
4. Ejecutamos el archivo de instalación como administrador.
5. Dejamos todas las opciones por defecto y finalizamos la instalación.

Para comprobar que se ha instalado correctamente, y qué versión, podemos hacer lo siguiente:

1. Escribimos “Node” en la barra de búsqueda de Windows.
2. Seleccionamos la opción “Node.js command prompt”.
3. Escribimos en la consola “npm -v”, y nos aparecerá la versión de Node.js que se ha instalado.

5.1.6. INSTALACIÓN DE ANGULAR

Angular puede instalarse de manera “local” para un proyecto concreto, o de manera “gobal”. Es más recomendable hacerlo de manera global.

Para la instalación:

1. Ejecutamos “Node.js command prompt”.
2. Escribimos “npm install -g @angular/cli” y pulsamos intro.

Para comprobar que se ha instalado correctamente, y qué versión, podemos hacer lo siguiente. Una vez abierto “Node.js command prompt”, escribimos “ng --version”.

5.2. DESPLIEGUE

Finalmente, no he subido la aplicación a un servidor. El servidor de la Uva trabaja con MySQL, y yo he utilizado SQL. Me informé debidamente, y la aplicación no funcionaría correctamente.

Como alternativa, con la ayuda de un amigo mío Ingeniero Informático, Alberto Ceruelo, se ha subido la API Web a un servidor IIS, y tras una serie de procedimientos, se pudo acceder a la aplicación desde cualquier dispositivo que estuviera conectado a la red wifi de mi casa.

Posteriormente, con la ayuda de Fernando Chico Pajares y Guillermo Méndez Vielba, se han realizado las configuraciones necesarias para utilizar mi CPU como servidor, y para abrir los puertos necesarios en el router. De esta manera, se consigue visualizar la aplicación de manera funcional, desde cualquier parte.

6 CONCLUSIONES

6. CONCLUSIONES

6.1. CONCLUSIONES GENERALES

Este proyecto presenta una revisión de las tecnologías básicas para el desarrollo Web, la cual permite una fácil comprensión de sus conceptos para una puesta en marcha de una futura aplicación. Por otro lado, también se profundiza en los fundamentos de Angular, con el fin de entender el entorno, su funcionamiento, y el flujo de ejecución de sus aplicaciones. Es destacable esta aportación, ya que se trata de un *framework* cada vez más utilizado en la industria debido a las aplicaciones modulares y de única página que permite desarrollar. Finalmente, se ha expuesto el desarrollo de 'Greative', una aplicación SPA en Angular, cuya función es facilitar a los futuros usuarios la etapa inicial del proceso de diseño. Es una propuesta de valor, ya que se trata de una etapa crucial de la que depende el resto del proceso. 'Greative' simula un portafolio de apuntes, donde el diseñador publica un producto que desea crear junto con notas o características a incluir en su propuesta, así como los modelos existentes que le inspiren.

Realizar este proyecto me ha permitido adquirir los conocimientos necesarios para desarrollar futuras aplicaciones de uso personal, o para terceros. Estoy muy orgulloso del resultado final porque, después de mucho ensayo y error, he conseguido que 'Greative' sea mejor que la idea inicial, ya que además de las funciones principales, he logrado incluir otras adicionales de gran utilidad. Por ejemplo, que al pulsar en un producto se muestren sólo los modelos de ese producto, o que la aplicación esté optimizada para su uso en dispositivos de cualquier resolución.

6.2. FUTURO TRABAJO

Esta aplicación está pensada para que un usuario que la utilice se encuentre solamente con sus productos y modelos. Para ello haría falta desarrollar algún tipo de proceso de registro, y asociar de alguna forma una base de datos a cada usuario. Es un procedimiento requerido que debo realizar si quisiera sacar la aplicación al mercado, pero supone conocimientos aún no adquiridos.

Por otro lado, sería interesante incluir una función para 'agregar a favoritos' productos o modelos. De esta manera, el diseñador localizaría más fácilmente los productos y modelos sobre los que está trabajando en ese momento.

7

BIBLIOGRAFÍA

BIBLIOGRAFÍA

[1] ALBERCA, CÉSAR, 2018. TypeScript VS JavaScript - Adictos al trabajo. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.adictosaltrabajo.com/2018/08/01/typescript-vs-javascript/.

[2] ART OF ENGINEER, 2020. (1197) Learn Angular 10, Web API & SQL Server by Creating a Web Application from Scratch - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=4a9VxZjnT7E.

[3] DALTO, LUCAS, 2019. (1197) Curso de HTML5 desde CERO (Completo) - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=kN1XP-Bef7w.

[4] DALTO, LUCAS, 2020. (1197) Curso de CSS desde CERO (Completo) - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=0WKXEJN67FE.

[5] DALTO, LUCAS, 2020. (1197) Curso de JAVASCRIPT desde CERO (Completo) - Nivel JUNIOR - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=z95mZVUcJ-E.

[6] DALTO, LUCAS, 2020. (1197) Curso de JAVASCRIPT desde CERO (Completo) - Nivel MID LEVEL - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=xOinGb2MZSk.

[7] DÍAZ, JOHN, 2020. ¿Qué es BACKEND y FRONTEND? (guía completa) | EDteam. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://ed.team/blog/que-es-backend-y-frontend-guia-completa.

[8] DIWAN, AMIT, 2020. Event bubbling vs event capturing in JavaScript? [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.tutorialspoint.com/event-bubbling-vs-event-capturing-in-javascript.

[9] EUROSTAT, 2020. Statistics | Eurostat. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://ec.europa.eu/eurostat/databrowser/view/tin00028/default/line?lang=en.

[10] FREEMAN, ADAM, 2020. *Pro Angular 9: Build Powerful and Dynamic Web Apps*. Apress; 4th ed. edición. ISBN 1484259971.

[11] GOOGLE, 2016. Angular - Getting started with Angular. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/start.

[12] GOOGLE, 2016. Angular - Introduction to Angular concepts. [en línea]. [Consulta:

2 julio 2021]. Disponible en: https://angular.io/guide/architecture.

[13] GOOGLE, 2016. Angular - Lifecycle hooks. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/lifecycle-hooks.

[14] GOOGLE, 2016. Angular - Property binding. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/property-binding.

[15] GOOGLE, 2016. Angular - Text interpolation. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/interpolation.

[16] GOOGLE, 2016. Angular - Tour of Heroes app and tutorial. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/tutorial.

[17] GOOGLE, 2016. Angular - Two-way binding. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/two-way-binding.

[18] GOOGLE, 2016. Angular - What is Angular? [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/what-is-angular.

[19] GOOGLE, 2016. Angular - Writing structural directives. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/structural-directives.

[20] GOOGLE, 2016. Angular - Introduction to components and templates. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/architecture-components.

[21] GOOGLE, 2016. Angular - Introduction to modules. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/architecture-modules.

[22] GOOGLE, 2016. Angular - Introduction to services and dependency injection. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://angular.io/guide/architecture-services.

[23] INSTITUTO NACIONAL DE ESTADÍSTICA, 2020. Proporción de personas (16 a 74 años) que utilizan Internet en los últimos tres meses por comunidades autónoma(45877). [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.ine.es/jaxiT3/Tabla.htm?t=45877&L=0.

[24] LÓPEZ QUIJADO, J., 2004. *Domine JavaScript*. book. Madrid: Ra-Ma. ISBN 84-7897-612-4.

[25] MOZILLA AND INDIVIDUAL CONTRIBUTORS, 2005. Conceptos básicos de HTML - Aprende sobre desarrollo web | MDN. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://developer.mozilla.org/es/docs/Learn/Getting_started_with_the_web/HTML_basics.

[26] MOZILLA AND INDIVIDUAL CONTRIBUTORS, 2005. CSS básico - Aprende sobre

desarrollo web | MDN. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://developer.mozilla.org/es/docs/Learn/Getting_started_with_the_web/CSS_basics

[27] MOZILLA AND INDIVIDUAL CONTRIBUTORS, 2005. ¿Qué es JavaScript? - Aprende sobre desarrollo web | MDN. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript

[28] NEGRINO, T., 2005. *Javascript : guía de aprendizaje /*. book. 5ª ed. Madrid [etc: Prentice Hall,. ISBN 84-205-4646-1.

[29] ORIOL, ENRIQUE. 1.22 – Servicios de Angular e Inyección de Dependencias - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=I72nh72brFQ

[30] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Componentes. Vídeo 5 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=CitMo3hip6Y.

[31] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Componentes II. Vídeo 6 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=DfSuph1OyZ0.

[32] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Directivas I. Vídeo 14 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=1NnPbUYZZ5I.

[33] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Estructura y flujo. Vídeo 4 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=uFA09pl_f4s.

[34] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Event Binding. Vídeo 10 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=FPjFXQf1pqM.

[35] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Interpolación. Vídeo 7 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=KfYyWNqJaVc.

[36] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Property Binding. Vídeo 9 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=ILO7-5Hnxt8.

[37] PILDORASINFORMATICAS, 2021. (1197) Curso Angular. Two way binding (binding bidireccional). Vídeo 11 - YouTube. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.youtube.com/watch?v=s1cbV27o6LE.

[38] REFSNES DATA, 1999. CSS Tutorial. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.w3schools.com/css/default.asp.

[39] REFSNES DATA, 1999. HTML Tutorial. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.w3schools.com/html/.

[40] REFSNES DATA, 1999. JavaScript Tutorial. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.w3schools.com/js/default.asp.

[41] REFSNES DATA, 1999. SQL Tutorial. [en línea]. [Consulta: 2 julio 2021]. Disponible en: https://www.w3schools.com/sql/default.asp.

[42] ULLMAN, L., 2004. *MySQL : guía de aprendizaje*. book. 2004. Ultima reimp. Madrid: Pearson Educación,. Guía de aprendizaje. ISBN 84-205-3843-4.

[43] VARIOS AUTORES, 2020. *Angular – Desarrolle sus aplicaciones web con el framework JavaScript de Google*. Eni. ISBN 2409025307.


SAPIENTIA
AEDIFICAVIT
SIBI
DOMVM