Lab4 - JS, DOM y renderizadores

Pepe Calderón, Fernando Sola, Daniel Ayala, Inma Hernández, Margarita Cruz, Carlos Arévalo, David Ruiz

Escuela Técnica Superior de
Ingeniería Informática

Índice

1 Objetivo

El objetivo de esta práctica es introducir al alumno a JavaScript, el lenguaje de programación ejecutable por los navegadores que permite añadir comportamiento e interacciones complejas a la Web, así como algunos usos básicos de JS. El alumno aprenderá a:

- Escribir código con los elementos básicos de JavaScript.
- Manipular los elementos de una página Web con JavaScript.
- Programar módulos renderizadores para generar representaciones HTML de objetos del dominio.

2

Introducción

Los lenguajes estudiados hasta ahora (HTML, CSS) no son lenguajes de programación, ya que aunque definimos los elementos de una página web y su estilo, no permiten definir una secuencia de instrucciones ejecutable, ya sea al cargar la página, o como respuesta a alguna interacción por parte del usuario.

Esta funcionalidad está reservada a JavaScript, un lenguaje de programación conocido principalmente por su uso en navegadores, aunque no está limitado a éstos. JavaScript es un lenguaje interpretado, débilmente tipado y multiparadigma. Los navegadores modernos incluyen todo lo necesario para ejecutar código JavaScript durante la visualización de páginas. El código JavaScript puede interactuar con la página web, usando información relacionada con la navegación (como la posición del cursor), o alterando los elementos mostrados. Sin JavaScript, una página es completamente estática, ya que los elementos en ella son los que se definen en el código HTML mostrado, sin que estos cambien en algo más allá de su estilo. Como otros lenguajes de programación, JavaScript permite definir variables y usar tipos como objetos, arrays, funciones, incorporar librerías, etc. Pese a su nombre, es muy diferente a Java.

3

Introducción a JavaScript

El código JavaScript a ser ejecutado puede escribirse tanto en el mismo archivo en el que se encuentra el código HTML mediante la etiqueta `<script>`, o en archivos `.js` independientes. El primer caso está desaconsejado salvo en aquellos casos en los que el código a introducir no ocupe más de 4-5 líneas, y no sea reutilizable en otras páginas de la aplicación.

En nuestro proyecto, crearemos un archivo `.js` en la carpeta `web/js/` por cada vista `.html` de la aplicación, con el mismo nombre. Por ejemplo, la vista `index.html` tendrá asociado un archivo `index.js` en la carpeta anteriormente mencionada.

Para vincular el archivo JS a la vista HTML, incluiremos una etiqueta `<script>` al final del `<head>` de la vista en cuestión:

```
<script src="js/index.js" type="module"></script>
```

El atributo `type="module"` indica que usaremos las funcionalidades de módulos JS para importar elementos de otros archivos, como veremos más adelante en este boletín.

3.1. Hola, mundo

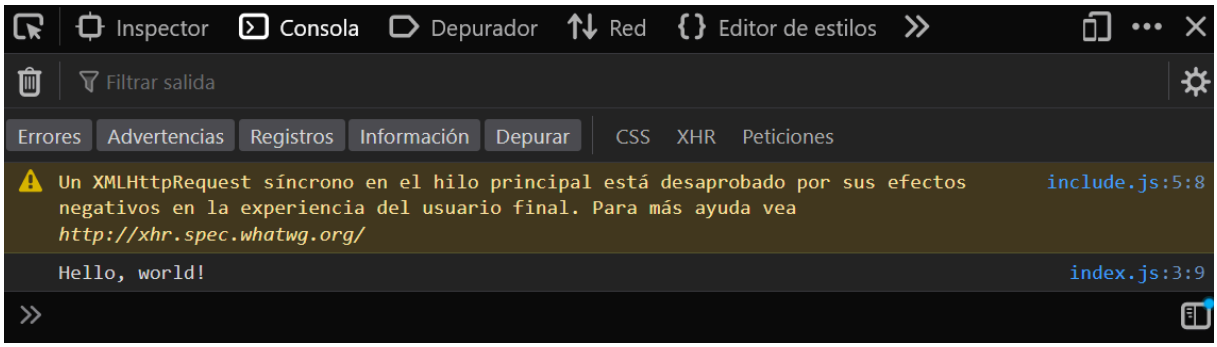
Para probar que el archivo se ha vinculado correctamente a la vista, podemos escribir un "Hola mundo" en `index.js`:

```
"use strict";

console.log("Hello, world!");
```

El `use strict";` al principio del fichero activa el modo estricto de JavaScript, que convierte en errores algunos comportamientos de JS que de otro modo únicamente se considerarían advertencias. Usar siempre el modo estricto es una buena práctica, y además garantiza la compatibilidad con futuras versiones de JS.

Más abajo, usamos la función `console.log` para imprimir una línea en la consola de depuración. Si actualizamos nuestra página, no veremos ningún cambio en ella. Para ver el mensaje impreso, debemos acceder a la consola del navegador pulsando y seleccionando la pestaña "Consola":



Además de nuestro mensaje, aparecen advertencias producidas por otras librerías usadas, que pueden ser ignoradas o deshabilitadas en la configuración de la consola de depuración.

3.2. Punto de entrada

Pese a que se puede escribir directamente el código a ejecutar fuera de cualquier función, esta no es una buena práctica. Se recomienda definir una función `main`, que se ejecutará cuando la página esté completamente cargada:

```
"use strict";

function main() {
  // Your code here
}

document.addEventListener("DOMContentLoaded", main);
```

El código que deseemos ejecutar irá dentro de la función `main()`. La última línea indica al navegador que se debe ejecutar la función `main()` una vez todo el contenido de la página esté listo. Es posible definir funciones auxiliares fuera de la función principal. En una subsección posterior se cubre en más detalle el uso de funciones.

3.3. Variables

En JS se pueden definir variables mediante `let`. También es posible usar `var` para la definición de variables, pero se desaconseja su uso en código JS moderno. El estilo recomendado de nombrado es camelCase. Los tipos básicos son los comunes a la mayoría de lenguajes: cadenas, booleans, números enteros y decimales:

```
let age = 25;
let heightCm = 1.76;
let firstName = "John";
let goodStudent = true;
```

Las cadenas pueden escribirse usando comillas simples o dobles. También pueden usarse acentos graves `` ` `` para las cadenas e intercalar en ellas variables mediante la sintaxis `${}`, por ejemplo:

```
let greeting = `My name is ${firstName} and I'm ${age} years old`;
console.log(greeting);

if(goodStudent) {
  console.log("I'm a good student!");
} else {
  console.log("I'm not a good student.");
}
```

Lo que imprimirá por consola:

```
My name is John and I'm 25 years old                    index.js:9
I'm a good student!                                     index.js:12
```

Otros tipos básicos son los arrays y los objects. Un array es una lista ordenada de elementos, que no tienen por qué ser del mismo tipo (aunque se recomienda que lo sean, por consistencia). Un Object es similar a un diccionario en otros lenguajes, asociando valores de cualquier tipo a claves textuales.

```
let myList = ["one", "two", "three"];
let myObject = {
  one: 1,
  two: 2,
  three: 3
};
```

Los arrays pueden ser iterados mediante `for..of` o mediante un bucle `for` estándar (se recomienda usar el primero siempre que sea posible):

```
// These two loops are equivalent
for(let elem of myList) {
```

```
  console.log(elem);
}

for(let i = 0; i < myList.length; i++) {
  let elem = myList[i];
  console.log(elem);
}
```

Asimismo, los elementos de un Object pueden ser accedidos de forma manual mediante un punto, o usando una variable mediantes corchetes:

```
console.log(myObject.one);  // Prints 1

for(let elem of myList) {
  console.log(myObject[elem]);  // Prints 1, 2, 3
}
```

3.4. Funciones

Las funciones se definen mediante la palabra reservada `function`:

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

let result = addTwoNumbers(5, 3);
console.log(result);
```

También se pueden definir mediante la "notación flecha". Esta notación se suele usar, sobre todo, para crear una función anónima de una o dos líneas, por ser más compacta:

```
let addTwoNumbers = (a, b) => {
  return a + b
};

let result = addTwoNumbers(5, 3);
console.log(result);
```

Si la función tiene sólo una instrucción que devuelve un valor, no es necesario incluir las llaves ni la palabra `return`:

```
let addTwoNumbers = (a, b) => a + b;
```

Se recomienda, por consistencia, definir las funciones usando `function`, salvo en el caso mencionado anteriormente.

En JavaScript, las funciones también son variables, por lo que se pueden pasar como parámetros referenciándolas por su nombre. Por ejemplo, la función `filter` de un array recibe como parámetro una función que determina si un elemento debe pasar el filtro o no:

```
function isEven(x) {
  return x % 2 === 0;
}

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let evenNumbers = numbers.filter(isEven);
console.log(evenNumbers);  // [2, 4, 6, 8]
```

En estos casos, la notación flecha para funciones puede resultar más compacta:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let evenNumbers = numbers.filter(x => x % 2 === 0);
console.log(evenNumbers);  // [2, 4, 6, 8]
```

4

Manipulación del DOM

En un documento HTML, todos los elementos contenidos en él se estructuran jerárquicamente en forma de árbol, formando una construcción llamada *Document Object Model* (DOM). El DOM es, por tanto, una representación del contenido HTML que se muestra en una web.

4.1. Selección y modificación básica

Los navegadores proporcionan una API programática para interactuar con el DOM de una página web mediante JS, pudiendo añadir, modificar y eliminar contenido de manera dinámica. Para ello, en nuestro código debemos localizar el elemento HTML con el que queremos interaccionar mediante, por ejemplo, su atributo `id`.

Podemos realizar una prueba añadiendo un nuevo elemento `<div>` con un ID determinado en cualquier lugar de nuestro `index.html`, por ejemplo:

```
<div id="dom-test"></div>
```

Normalmente, para acceder a cualquier elemento del DOM realizaremos una consulta al elemento raíz, llamado `document`. La variable `document` hace referencia al propio documento HTML y es de ámbito global, por lo que puede ser usada en cualquier lugar del código:

{: .warning }

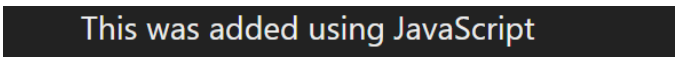
> **Info**
>
> Recuerde escribir su código en la función `main()` tal y como se definió anteriormente. Si no, es posible que su código se ejecute **antes** de que los elementos HTML estén cargados en la página, lo que hará imposible acceder a ellos a través del DOM.

```
let myDiv = document.getElementById("dom-test");
myDiv.textContent = "This was added using JavaScript";
```

Observe lo siguiente:

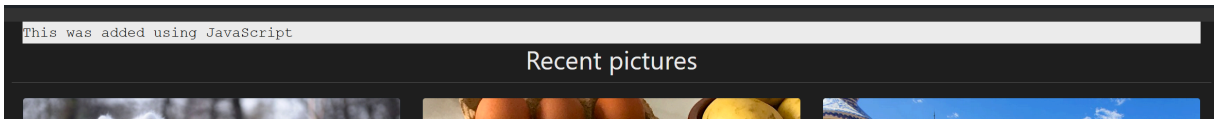
- La función `getElementById` permite, dado el ID de un elemento, buscarlo en el interior de otro elemento. En este caso, dado que `document` es el elemento raíz que contiene a todos los demás, lo estamos buscando en cualquier lugar del documento HTML.
- La función `getElementById` devuelve un objeto de tipo nodo HTML si se ha encontrado el elemento solicitado. Si no, devuelve `null`.
- El atributo `textContent` de un nodo HTML representa el texto contenido en él. Se puede sobreescribir para cambiar su contenido.

Pese a que el elemento `<div>` que hemos insertado en el HTML estaba vacío, se le ha dado un contenido de texto mediante JS:



Al elemento añadido dinámicamente se le añaden todos los estilos aplicables de las hojas de estilo de la página. Si deseamos modificar el estilo de un elemento en el árbol DOM, otro atributo de interés que tienen los nodos HTML en JS es `style`, que permite modificar de forma dinámica el estilo de un elemento alterando sus propiedades CSS:

```
myDiv.style.backgroundColor = "#EEEEEE";
myDiv.style.color = "black";
myDiv.style.fontFamily = "monospace";
```



Observará que los atributos accesibles mediante `style` tienen el mismo nombre que las propiedades CSS que se modifican mediante las hojas de estilo.

4.2. Creación de nodos HTML

Además de seleccionar elementos ya existentes, es posible crear elementos HTML nuevos e incluirlos en el documento mediante la función `document.createElement()`. Se debe indicar

el tipo de elemento a crear como parámetro. Por ejemplo, podemos añadirle un párrafo `<p>` al `<div>` que creamos anteriormente:

```
let newP = document.createElement("p");
newP.textContent = "This is a new paragraph";
myDiv.appendChild(newP);
```

En este caso, `"p"` hace referencia a que deseamos crear un elemento de tipo `<p>`. Al igual que antes, podemos establecer su contenido textual mediante `textContent`. Finalmente, lo añadimos al `<div>` usando la función `appendChild()`, que añade un elemento como hijo de otro.

La representación HTML de los elementos creados será:

```
<div id="dom-test">
  This was added using JavaScript
  <p>
    This is a new paragraph
  </p>
</div>
```

{: .warning }

Info

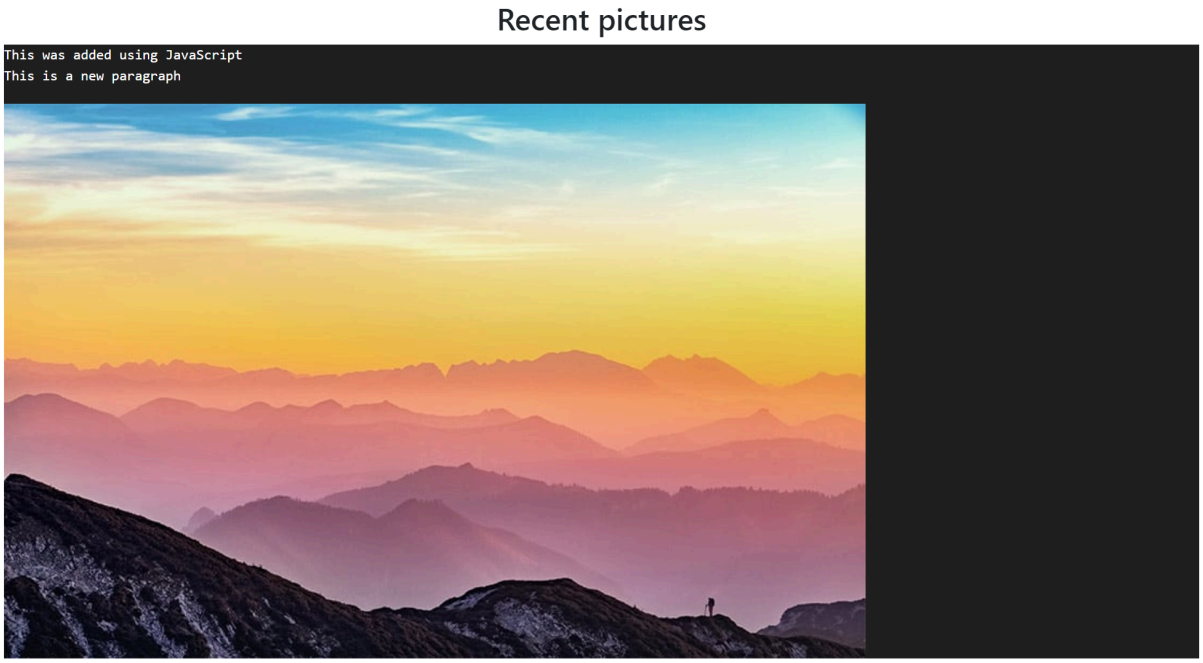
Es importante destacar que crear un nuevo elemento HTML usando la función `document.createElement()` no hace que éste aparezca inmediatamente en el documento HTML. Hasta que no se introduzca dentro de algún otro elemento de la página, el elemento creado no será visible.

Un ejemplo más complejo es el siguiente, en el que se crea dinámicamente una imagen con unos atributos `src` y `title` determinados:

```
let newImage = document.createElement("img");
newImage.src = "https://loadedlandscapes.com/wp-content/uploads/2019/07/
lighting.jpg";
newImage.title = "A beautiful landscape";

myDiv.appendChild(newImage);
```

Los atributos del elemento creado, como `src` o `title`, pueden establecerse modificando los atributos del nodo en JS. El resultado final es:



4.3. Selectores avanzados

En la sección anterior usamos la función `getElementById` para obtener un elemento contenido dentro de otro, según su atributo `id`. Sin embargo, en ocasiones puede ser útil seleccionar elementos según otros criterios. Una función relacionada es `querySelector`, que permite seleccionar nodos HTML usando los mismos selectores que se usan en CSS. Por ejemplo, podemos seleccionar el elemento `<div>` con atributo `class="container` para cambiar su fondo:

```
let container = document.querySelector("div.container");
container.style.backgroundColor = "#BBBBBB";
```

Al igual que `getElementById`, `querySelector` devuelve el primer nodo encontrado que cumpla con el selector, o `null` si no se encuentra ninguno. Note que los siguientes dos selectores son equivalentes, ya que `querySelector` generaliza a `getElementById`:

```
document.getElementById("id");
document.querySelector("#id");
```

Existe una función relacionada, `querySelectorAll`, que devuelve un array de todos los elementos que cumplen un selector concreto. A modo de ejemplo, recordemos que todas las fotos están contenidas en una tarjeta, que es un `<div class="card">`. Podemos usar este selector para encontrar todas las tarjetas en el contenedor de la galería y cambiar su estilo, por ejemplo, añadiéndoles un ligero sombreado:

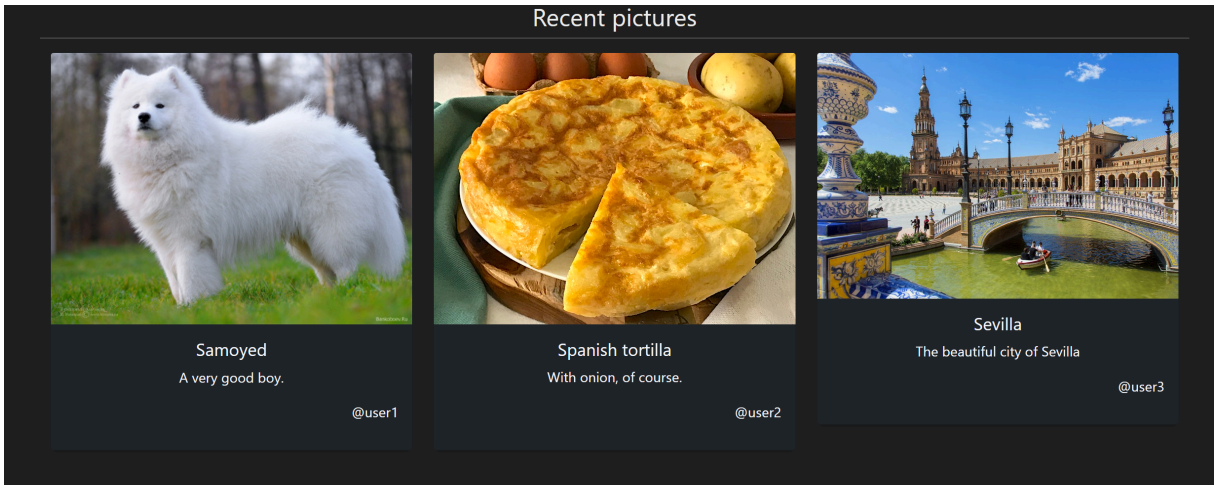
```
let container = document.querySelector("div.container");

let cards = container.querySelectorAll("div.card");
for (let card of cards) {
    card.style.boxShadow = "0 4px 8px 0 rgba(0, 0, 0, 0.2)";
}
```

Observe las siguientes consideraciones:

- `querySelectorAll` siempre devuelve un array de elementos, por lo que podemos iterar sobre ellos con `for..of`. Si no se encuentra ninguno, en lugar de `null` devuelve un array vacío.
- Todos los selectores estudiados (`getElementById`, `querySelector` y `querySelectorAll`) pueden usarse sobre `document`, para buscar los elementos en cualquier parte del DOM, o sobre un nodo concreto, para buscar los elementos en cuestión únicamente dentro de ese nodo.
- En el ejemplo mostrado, el punto anterior implica que un `<div class="card">` que no estuviera contenido en el `container` no sería seleccionado.

El código anterior modifica de forma dinámica el estilo de la galería, con este resultado:



4.4. Creación de nodos avanzada

En la Sección aprendimos a crear e introducir manualmente nodos HTML en el documento a través del DOM. Sin embargo, este método es poco escalable para nodos complejos. Tomemos por ejemplo una tarjeta de la galería:

```
<div class="col-md-4">
  <div class="card bg-dark text-light">
    <img src="https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg" class="card-img-top">

    <div class="card-body">
```

```
      <h5 class="card-title text-center">Samoyed</h5>
      <p class="card-text">A very good boy.</p>
      <p class="text-end">@user1</p>
    </div>
  </div>
</div>
```

Cada tarjeta de este estilo está compuesta de un total de 7 nodos HTML, cada uno de ellos con una serie de atributos y texto. Sin duda, crearlos todos manualmente en el código sería tedioso. Por ello, en estos casos se suele almacenar una representación en forma de cadena del código HTML, y usar una función que interprete esa cadena, generando el nodo raíz y todos los que se encuentren en su interior automáticamente, para poder entonces colocarlos en el DOM.

La plantilla de proyecto Silence incluye esta función, llamada `parseHTML()`, en un módulo de utilidades. Para utilizarla, debemos importarla del módulo correspondiente al principio del fichero JS:

```
import { parseHTML } from "/js/utils/parseHTML.js";
```

La importación de elementos funciona de manera similar a otros lenguajes: se debe indicar entre llaves los elementos a importar, y el módulo (archivo) en el que se encuentran.

Podemos entonces usar la función para convertir una cadena, representando HTML, en un objeto correspondiente a un nodo HTML que podemos manipular mediante JS como hemos visto anteriormente:

```
let html = `<div class="col-md-4">
  <div class="card bg-dark text-light">
    <img src="https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg" class="card-img-top">

    <div class="card-body">
      <h5 class="card-title text-center">Samoyed</h5>
      <p class="card-text">A very good boy.</p>
      <p class="text-end">@user1</p>
    </div>
  </div>
</div>`;

let newCard = parseHTML(html);
let container = document.getElementById("gallery");
container.appendChild(newCard);
```

En este caso, hemos de asignarle un ID “gallery” al elemento que contiene las tarjetas correspondientes a las fotos, para poder acceder a él fácilmente e insrtar nuevos elementos.

Note que hemos usado los acentos graves `` ` `` ya que permiten que una cadena se extienda por varias líneas. Si accedemos a nuestra galería, podemos ver que contiene la nueva tarjeta.

5 Renderizadores

Como hemos visto en la sección anterior, es sencillo generar nodos a partir de una cadena que contiene HTML. Esto puede ser generalizado aún más: dado un objeto JS que represente a una entidad, se podrían acceder a los diferentes campos de la entidad y colocar la información relevante en los lugares pertinentes de la cadena HTML, generando programáticamente la representación adecuada.

5.1. Renderizador de fotos

Podemos definir una función que recibe una foto y devuelve un nodo HTML representándola como una tarjeta:

```
function photoAsCard(photo) {
    let html = `<div class="col-md-4">
    <div class="card bg-dark text-light">
      <img src="${photo.url}" class="card-img-top">

      <div class="card-body">
        <h5 class="card-title text-center">${photo.title}</h5>
        <p class="card-text">${photo.description}</p>
        <p class="text-end">User ${photo.userId}</p>
      </div>
    </div>
  </div>`;

    let card = parseHTML(html);
```

```
    return card;
}
```

Observe que se usa la sintaxis `${}` para intercalar variables en la cadena HTML, que se corresponden con los atributos del objeto `photo` recibido como parámetro de la función.

Esta función nos permite abstraer la representación de una foto como tarjeta, ya que recibe un objeto JS representando una foto y nos devuelve un nodo con todo el contenido necesario. Por tanto, es muy probable que queramos reutilizarla en varios lugares de nuestra aplicación, siempre que queramos mostrar una foto como tarjeta (por ejemplo, tanto en la página principal de la galería como en el perfil de un usuario).

La manera de lograr esto es mediante un módulo de renderizado de fotos, que contendrá un objeto con funciones para renderizar una foto de diferentes maneras. Para ello, crearemos un nuevo archivo `photos.js` en la carpeta `web/js/renderers/`:

```
"use strict";

const photoRenderer = {
  ...
};

export { photoRenderer };
```

Como en todo nuestro código JS, usaremos el modo estricto. A continuación definiremos un objeto `photoRenderer`, que contendrá funciones de renderizado de fotos. Finalmente, exportaremos el renderizador mediante `export { ... }` para que pueda ser importado en otros lugares del código con `import`. Observe que está definido mediante `const` para que las funciones del renderizador no puedan ser modificadas externamente.

Podemos añadir la función anterior al renderizador, llamándola `asCard`:

```
"use strict";
import { parseHTML } from "/js/utils/parseHTML.js";

const photoRenderer = {
    asCard: function(photo) {
        let html = `<div class="col-md-4">
      <div class="card bg-dark text-light">
        <img src="${photo.url}" class="card-img-top">

        <div class="card-body">
          <h5 class="card-title text-center">${photo.title}</h5>
          <p class="card-text">${photo.description}</p>
          <p class="text-end">User ${photo.userId}</p>
        </div>
      </div>
    </div>`;
```

```
    let card = parseHTML(html);
    return card;
    }
};

export { photoRenderer };
```

Observará que se trata de un objeto cuyos atributos son funciones, lo que permite invocarlas como si se trataran de métodos del objeto.

Esto nos permite añadir una foto a la galería dados sus atributos, sin tener que preocuparnos de generar su estructura HTML. Esto será especialmente útil en el futuro, cuando las fotos procedan de una consulta a la API del proyecto. Si modificamos el código de `index.js` para usar el renderizador, resulta:

```
import { photoRenderer } from "/js/renderers/photos.js";

function main() {
    let container = document.getElementById("gallery");
    let photo = {
        title: "Samoyed",
        description: "A very good boy.",
        userId: 1,
        url: "https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg",
    };

    let card = photoRenderer.asCard(photo);
    container.appendChild(card);
}
```

El resultado es el mismo que en el apartado anterior, pero el código para generar el nodo HTML queda abstraído.

5.2. Renderizador de la galería

Una vez hemos abstraído la generación del código HTML para las fotos, podemos dar un paso más y crear un renderizador para toda la galería, que internamente usará el renderizador de fotos. Al igual que antes, crearemos un módulo (archivo) `web/js/renderers/gallery.js`:

```
"use strict";

import { parseHTML } from "/js/utils/parseHTML.js";

const galleryRenderer = {
  ...
};
```

```
export { galleryRenderer };
```

El renderizador de la galería usará el renderizador de fotos, por lo que debemos importarlo también:

```
import { photoRenderer } from "/js/renderers/photos.js";
```

En nuestro caso, tendremos una función para representar un array de fotos como una galería de tarjetas. Si se desea otro tipo de galería, se podría implementar también en el renderizador:

```
const galleryRenderer = {
    asCardGallery: function (photos) {
        ...
    }
};
```

En el código para renderizar nuestra galería, comenzaremos creando un elemento `<div>` que contendrá todas las filas de la galería, al que daremos clase `photo-gallery`. Crearemos también una fila inicial para insertar las fotos:

```
asCardGallery: function (photos) {
  let galleryContainer = parseHTML('<div class="photo-gallery"></div>');
  let row = parseHTML('<div class="row"></div>');
  galleryContainer.appendChild(row);
}
```

A continuación, iteraremos sobre todas las fotos recibidas como parámetro de la función, convirtiéndolas en su representación como tarjeta gracias al renderizador de fotos y añadiéndolas a la fila de la galería. Dado que nuestra galería contiene 3 fotos por fila, debemos llevar un conteo de las fotos insertadas. Cada 3 fotos, añadiremos una nueva fila a la galería y seguiremos insertando las nuevas fotos en esa fila creada. Finalmente, devolveremos el elemento que contiene toda la galería:

```
asCardGallery: function (photos) {
  let galleryContainer = parseHTML('<div class="photo-gallery"></div>');
  let row = parseHTML('<div class="row"></div>');
  galleryContainer.appendChild(row);

  let counter = 0;

  for (let photo of photos) {
    let card = photoRenderer.asCard(photo);
    row.appendChild(card);
    counter += 1;

    if (counter % 3 === 0) {
```

```
      row = parseHTML('<div class="row"></div>');
      galleryContainer.appendChild(row);
    }
  }

  return galleryContainer;
}
```

Para comprobar su correcto funcionamiento, crearemos un array de fotos en `index.js` y lo convertiremos dinámicamente en una galería, usando el renderizador recién creado:

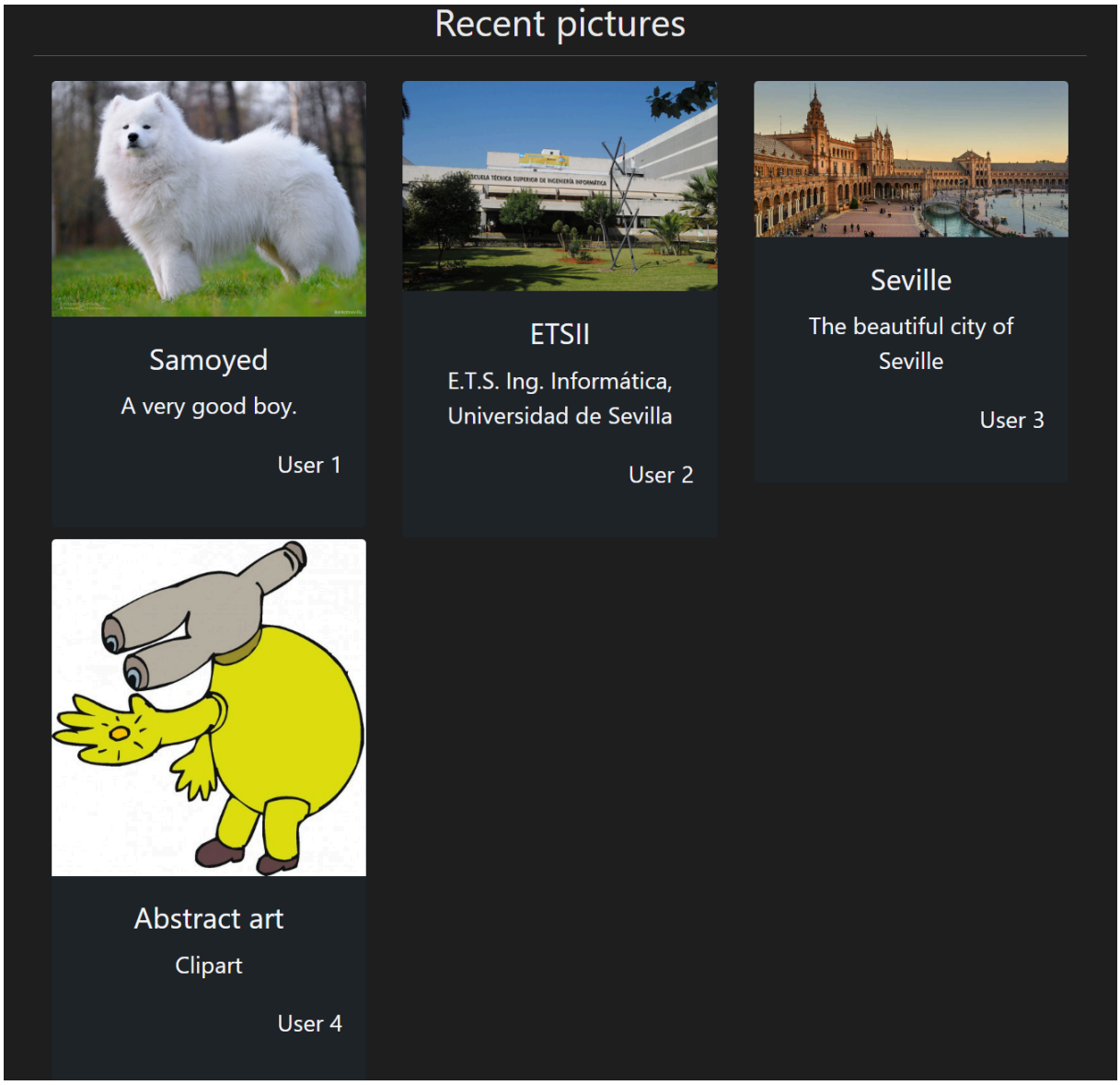
```
import { galleryRenderer } from '/js/renderers/gallery.js';

function main() {
  let container = document.getElementById("gallery");
  let photos = [
    {
      title: "Samoyed",
      description: "A very good boy.",
      userId: 1,
      url: "https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg",
      date: "15/08/2020",
    },
    {
      title: "ETSII",
      description: "E.T.S. Ing. Informatica, Universidad de Sevilla",
      userId: 2,
      url: "https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/ETSI_
Inform%C3%A1tica_Sevilla_y_DrupalCamp_Spain_2011.jpg/1920px-ETSI_Inform%C3%
A1tica_Sevilla_y_DrupalCamp_Spain_2011.jpg",
      date: "01/01/2021",
    },
    {
      title: "Seville",
      description: "The beautiful city of Seville",
      userId: 3,
      url: "https://urbansevilla.es/wp-content/uploads/2019/03/urban-
sevilla-foto-ciudad.jpg",
      date: "03/02/2019",
    },
    {
      title: "Abstract art",
      description: "Clipart",
      userId: 4,
      url: "https://clipartart.com/images/worst-clipart-ever-1.jpg",
      date: "14/08/2019",
    },
  ];

  let gallery = galleryRenderer.asCardGallery(photos);
```

```
  container.appendChild(gallery);
}
```

Observe que ya no es necesario crear la galería manualmente mediante HTML, sino que es generada automáticamente a partir de un array de fotos. Además, la abstracción en la creación de la galería permite insertarla en cualquier lugar de nuestra aplicación (por ejemplo, podemos mostrar una galería con las fotos subidas por un usuario en la página de su perfil). El resultado es el siguiente:



5.3. Renderizador de detalles de foto

El renderizador de fotos, por el momento, puede recibir una foto y mostrarla en formato tarjeta para la galería. Pero existe una vista en la que deseamos mostrar una imagen individual

en un formato diferente: la de detalles de foto. En ese caso, querremos mostrar la misma foto no en formato miniatura como hasta ahora, sino a tamaño completo.

Dado que en la práctica anterior ya definimos la estructura HTML para mostrar una foto en detalle en `photo_detail.html`, podemos generalizarla añadiéndole un nuevo método al renderizador de fotos:

```
const photoRenderer = {
  asCard: function (photo) {
    ...
  },

  asDetails: function (photo) {
    let html = `<div class="photo-details">
        <h3>${photo.title}</h3>
        <h6>${photo.description}</h6>
        <p>Uploaded by <a href="user_profile.html" class="user-link">User
${photo.userId}</a> on ${photo.date}</p>

        <hr>

        <img src="${photo.url}" class="img-fluid">
      </div>`;

    let photoDetails = parseHTML(html);
    return photoDetails;
  },
};
```

Podemos, entonces, enlazar un archivo JavaScript `photo_detail.js` a la vista `photo_detail.html` y usar el renderizador para mostrar una foto concreta en detalle. En ese caso, dejaremos vacía la columna que contiene los detalles de la foto, y le daremos un ID para poder localizarla mediante JavaScript y añadirle el contenido dinámicamente:

```
<div class="col-md-9" id="photo-details-column">

</div>
```

Entonces, en `photo_detail.js` realizamos una operación similar a la de la página principal, definiendo una foto en JS y usando el renderizador para transformarla en una representación HTML adecuada, que colocamos en el DOM:

```
"use strict";

import { photoRenderer } from "/js/renderers/photos.js";

function main() {
    let photoContainer = document.querySelector("#photo-details-column");
```

```
    let photo = {
        title: "Samoyed",
        description: "A very good boy.",
        userId: 1,
        url: "https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg",
        date: "12/01/1996",
    };

    let photoDetails = photoRenderer.asDetails(photo);
    photoContainer.appendChild(photoDetails);
}

document.addEventListener("DOMContentLoaded", main);
```

Pese a que ahora mismo parece una sobrecarga de código para lograr lo que ya estaba conseguido anteriormente usando sólo HTML, en posteriores laboratorios veremos que ésta estructura de código facilita enormemente poder cargar cualquier foto en esta vista mediante una consulta a la API del proyecto.

6

Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo `commit` a efectuar.
- `git commit -m "mensaje"` crea un nuevo `commit` con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

> **Info**
>
> Versión PDF disponible