Desarrollar un CRUD con Vanilla JavaScript y Firebase

El objeto del presente proyecto es el desarrollo de una aplicación JS que interaccione con una base de datos alojada en un servidor remoto. Como base de datos usaremos ***Firestore***. Firestore es una base de datos NoSQL flexible, escalable y en la nube, creada en la infraestructura de Google Cloud, a fin de almacenar y sincronizar datos para el desarrollo tanto del lado del cliente como del servidor.

Estructura del proyecto

El proyecto consistirá en un aplicación básica de gestión de tareas que permitirá añadir nuevas, listarlas, borrar de forma individual o actualizarlas

La estructura del proyecto será la habitual. Una carpeta alojará el archivo `index.html` y a las subcarpetas `js` y `css`. Dentro de la carpeta `css` tendremos un archivo style.css, para la hoja de estilos del proyecto y en la carpeta `js` tendremos los módulos JS. En el caso que nos ocupa serán dos: Por un lado un archivo `app.js`, que recogerá el código que se ocupe de la lógica de la aplicación y por otro un archivo `firestore.js` que se encargará de la conexión con la base de datos remota.

Preparación de la base de datos

Para empezar a trabajar con Firestore es necesario registrar un proyecto en ***firebase***, la plataforma de desarrollo de google. Así pues, con nuestra cuenta de google podremos acceder a la consola de Firebase (https://console.firebase.google.com/) y crear un nuevo proyecto. La creación del proyecto es un proceso sencillo. Sólo tendremos en cuenta **no habilitar google analytics**, ya que a los efectos de este proyecto no tiene mucho sentido.

Una vez creado el nuevo proyecto tendremos que añadir una nueva aplicación al mismo. En este caso **una aplicación web** (los proyectos Firebase son multiplataforma y permiten el desarrollo de aplicaciones de distinta naturaleza). A la hora de registrar la nueva aplicación lo único que tendremos que tener en cuenta es **no activar la opción de hosting**. Hecho esto se nos ofrecerá la posibilidad de copiar un script para agregar el SDK de Firebase a nuestra aplicación. Podemos copiar este código en un fichero de texto para usarlo más tarde

Por último, registrada la aplicación, solo queda activar una nueva base de datos. Para ello buscaremos en el menú de compilación la entrada ***Firestore Database***. Seguiremos los pasos para registrar una nueva base de datos teniendo en cuenta la elección de la ubicación de servidor (que no tiene mucha relevancia para este proyecto en concreto) y **activar el modo de pruebas**. Esto último nos permitirá empezar rápidamente sin tener que configurar reglas de seguridad de protección de los datos.

Hasta aquí todo lo necesario para configurar la base de datos remota.

Conexión de la aplicación con la base de datos

Lo primero es conectar nuestro proyecto con la instancia de Firestore Database definida anteriormente. Se ha comentado que es muy recomendable que las operaciones que tengan que ver con la interacción con la base de datos, se desde desarrollen desde un módulo dedicado a tal efecto. En nuestro caso, dicho módulo es `firebase.js`. El siguiente código define la conexión con la estructura de la aplicación web de nuestro proyecto Firebase, así como la conexión con Firestore.

```
// firebase.js


import { initializeApp } from
"https://www.gstatic.com/firebasejs/10.7.1/firebase-app.js";

  // Your web app's Firebase configuration
  const firebaseConfig = {
    apiKey: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
    authDomain: "crud-js-394b8.firebaseapp.com",
    databaseURL: "https://crud-js-394b8-default-rtdb.firebaseio.com",
    projectId: "crud-js-394b8",
    storageBucket: "crud-js-394b8.appspot.com",
    messagingSenderId: "260132245242",
    appId: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
  };

  // Initialize Firebase
  export const app = initializeApp(firebaseConfig);
```

El primer `import` nos permite interaccionar con la aplicación y definir el objeto `app` que la inicializa. El segundo import permite la conexión con la base de datos. En este último caso iremos añadiendo entre las llaves aquellos módulos de Firestore que nos hagan falta para desarrollar la funcionalidad de CRUD de nuestra aplicación de gestión de tareas. La última línea permite hacer disponible la conexión con la aplicación Firebase al resto de módulos de nuestro proyecto JS

Llegados a este punto vamos a ver en el siguiente esquema cuál sería la estructura de este módulo de gestión de la base de datos.

```
firebase.js
├─ importación
├─ configuración de conexión
├─ inicialización
└─ funcionalidades
    ├─ salvar
    ├─ listar
    ├─ borrar
    └─ actualizar
```

A partir de aquí iremos añadiendo elementos al módulo conforme avancemos en el desarrollo

Estructura del módulo app.js

El módulo principal que soporta la lógica de la aplicación que no tiene que ver con la conexión con Firestore también tiene una estructura definida que podemos ver en el siguiente esquema:

```
app.js
├─ importación
├─ conexión con elementos DOM
├─ definición de variables globales
├─ punto de entrada - evento "DOMContentLoad"
│   └─ listar tareas y manejo de eventos asociados
└─ manejador del formulario -- evento "submit"
```

Vamos a ir trabajando ahora en el desarrollo de los distintos elementos

Presentación de la aplicación

Vamos a empezar a trabajar con el fichero HTML que da la estructura a la aplicación. Básicamente tenemos que programar un formulario que permita insertar tareas en la base de datos y habilitar una zona de la página donde se presente el listado de tareas guardas junto a botones que permitan editarlas o borrarlas. Un código posible podría ser el siguiente:

```
// index.html

<!DOCTYPE html>
<html lang="es">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>FireStore CRUD</title>
    </head>

    <body>
        <h1>Gestión de tareas</h1>

        <!-- Task Form -->
        <form id="task-form">
            <label for="task-title">Title:</label>
            <input type="text" id="task-title" placeholder="Título" />

            <label for="task-description">Description:</label>
            <textarea id="task-description" rows="3" placeholder="Descripción">
</textarea>

            <button id="btn-task-form">Guardar</button>
```

```
        </form>

        <!-- Tasks List -->
        <div id="tasks-container"></div>

        <!-- código externo -->
        <script src="js/app.js" type="module"></script>
    </body>
</html>
```

A partir de aquí podemos desarrollar el código base del módulo `app.js` según la estructura que se vio anteriormente. Para ello se definirán las conexiones con el DOM en la sección de definición de variables globales

```
// app.js

// importar funciones de módulo externo firebase.js
import {  } from './firebase.js'

// variables globales
const taskForm = document.getElementById("task-form");
const tasksContainer = document.getElementById("tasks-container");

// manejador de evento principal
window.addEventListener("DOMContentLoaded", (e) => {


});

// manejador de evento de envío del formulario
taskForm.addEventListener("submit", (e) => {
    e.preventDefault();


});
```

Hasta aquí ya tendríamos el esqueleto principal del proyecto. Las funciones que tenemos que programar son las propias de una estructura CRUD. Esta estructura consta de una serie de funcionalidades habituales en el trabajo con bases de datos. Estas son: (C)reate, (R)ead, (U)pdate y (D)elete. Quiere decir que tendremos que programar estas funcionalidades en el módulo firebase.js, tal y como hemos visto en el esquema anterior. Una vez que programemos estas funciones en `firebase.js`, las importaremos en `app.js`. Las funciones que habría que definir y desarrollar serían:

- onGetTasks → listar tareas almacenadas en la base de datos de manera síncrona
- saveTask → salvar tareas en la base de datos

- deleteTask → borrar tareas de la base de datos
- getTask → seleccionar una tarea de la base de datos
- updateTask → actualizar una tarea en la base de datos

Desarrollo de componentes de interacción con la base de datos e inserción en la lógica de la aplicación

Para desarrollar cada funcionalidad en concreto tendremos que, a su vez, definir una serie de funciones que necesitarán importar métodos propios del SDK de Firestore que la aplicación Firebase pone a nuestra disposición.

Crear un registro

Para crear una nueva tarea necesitaremos importar los métodos `getFirestore`, `collection` y `addDoc`. En realidad los dos primeros son necesarios para cualquier funcionalidad, ya que se encargan de inicializar la base de datos (¡ojo! no la aplicación, que se inicializa con el método `initializeApp`). Por su parte, `addDoc` permite añadir nuevos registros (documentos) a la base de datos. Nuestro código quedaría:

```
1  // firebase.js
2
3  // importación SDK de firebase y firestore
4  import { initializeApp } from
   "https://www.gstatic.com/firebasejs/10.7.1/firebase-app.js";
5  import { getFirestore, collection, addDoc } from
   "https://www.gstatic.com/firebasejs/10.7.1/firebase-firestore.js"
6
7  // Configuración de la aplicación web de firebase
8  const firebaseConfig = {
9      /* Aquí va el código de configuración de firebase */
10 };
11
12 // inicialización de firebase y firestore
13 export const app = initializeApp(firebaseConfig);
14 export const db = getFirestore();
15
```

Los objetos `app` y `db` permiten hacer uso de los métodos del SDK en el resto del código

Definimos la función `saveTask` que nos permitirá crear una nueva tarea

```
1  export const saveTask = (title, description) => {
2      return addDoc(collection(db, "tasks"), { title, description });
3  };
```

Es importante tener en cuenta que `addDoc` creará la colección “task” en FireStore DB **si no está definida de antemano**

Una vez definida la función podríamos integrarla en el módulo `app.js`. Así se podrían capturar los valores del formulario al hacer click en el botón “Guardar”. Veámoslo:

```
// app.js

// importar funciones de módulo externo firebase.js
import { saveTask } from './firebase.js'

// variables globales
const taskForm = document.getElementById("task-form");
const tasksContainer = document.getElementById("tasks-container");

// manejador de evento principal
window.addEventListener("DOMContentLoaded", (e) => {


});

// manejador de evento de envío del formulario
taskForm.addEventListener("submit", async (e) => {
    e.preventDefault();

    const title = taskForm["task-title"].value;
    const description = taskForm["task-description"].value;

    try {
        // Llamada a la función del módulo firebase.js
        await saveTask(title.value, description.value);
    } catch (error) {
        console.log(error)
    }
});
```

Hemos añadido la definición `async` en la función callback del el manejador de evento “submit” y `await` para llamar a la función `saveTask` por la razón de que estamos trabajando de modo asíncrono con una base de datos remota. A su vez usamos el bloque `try-catch` para capturar posibles errores en las peticiones.

Listar tareas guardadas

Al igual que hemos hecho en el caso anterior, tendremos primero que definir la función en el módulo de interacción con FireStore DB y después encajarla en la funcionalidad de `app.js`

En `firebase.js` necesitaremos importar el método `onSnapshot`, que nos permite, haciendo referencia a la colección donde guardamos los datos, escuchar los cambios en los documentos en la base de datos. Cómo segundo parámetro pasaremos una función callback que maneje los cambios detectados. En resumen, este método permite, en tiempo real, almacenar los cambios en la base de datos en un objeto manejable por JS y **sin tener que refrescar la página HTML**. Lo vemos en el siguiente código, que habrá que añadirse al que ya tenemos:

```
// firebase.js

// importación SDK de firebase y firestore
import { initializeApp } from
"https://www.gstatic.com/firebasejs/10.7.1/firebase-app.js";
import { getFirestore, collection, addDoc, onSnapshot } from
"https://www.gstatic.com/firebasejs/10.7.1/firebase-firestore.js"

/* resto de código */


export const onGetTasks = (callback) => {
    return onSnapshot(collection(db, "tasks"), callback);
};
```

La función callback la definiremos convenientemente en `app.js`

Para listar las tareas de forma dinámica importaremos y añadiremos en el bucle principal la función `onGetTasks` definida anteriormente. El siguiente código, que una nueva versión del módulo `app.js` muestra los detalles. Los comentarios aclaran las operaciones

```
// app.js

// importar funciones de módulo externo firebase.js
import { saveTask, onGetTasks } from './firebase.js'

// variables globales
const taskForm = document.getElementById("task-form");
const tasksContainer = document.getElementById("tasks-container");

// manejador de evento principal
window.addEventListener("DOMContentLoaded", async (e) => {

    await onGetTasks((querySnapshot) => {
        tasksContainer.innerHTML = "";
        // bucle que recorre todos los documentos del objeto querySnapshot
        querySnapshot.forEach((doc) => {
            // método .data() convierte el objeto de la DB en un objeto JS
            const task = doc.data();

```

```
            // inyectamos el código HTML de forma dinámica
            tasksContainer.innerHTML += `
            <div>
              <h3 class="h5">${task.title}</h3>
              <p>${task.description}</p>
              <div>
                <button class="btn-delete">Borrar</button>
                <button class="btn-edit">Editar</button>
              </div>
            </div>`;
        });
    });

});

// manejador de evento de envío del formulario
taskForm.addEventListener("submit", async (e) => {
    e.preventDefault();

    const title = taskForm["task-title"].value;
    const description = taskForm["task-description"].value;

    try {
        // Llamada a la función del módulo firebase.js
        await saveTask(title.value, description.value);
    } catch (error) {
        console.log(error)
    }
});

```

Algunas claves más de este código:

- Uso de Async/await para manejar las llamadas a funciones que operan de forma asíncrona.
- Uso de las clases `btn-edit` y `btn-delete` en los botones generados dinámicamente para poder delegar eventos posteriores al hacer click en ellos (Se verá en la siguiente sección)

Borrar tareas almacenadas

De nuevo seguimos el esquema anterior. Primero definimos función en `firebase.js` y posteriormente integramos en `app.js`

En `firebase.js` importamos el método `deleteDoc` y definimos la función correspondiente. El código quedaría así:

```
// firebase.js

// importación SDK de firebase y firestore
import { initializeApp } from
"https://www.gstatic.com/firebasejs/10.7.1/firebase-app.js";
import { getFirestore, collection, addDoc, onSnapshot, deleteDoc } from
"https://www.gstatic.com/firebasejs/10.7.1/firebase-firestore.js"

/* resto de código */

export const deleteTask = (id) => {
    return deleteDoc(doc(db, "tasks", id));
};
```

Es importante tener en cuenta que la función definida necesita el parámetro `id`, que tendremos que pasar desde la llamada de `app.js`. El método `deleteDoc` necesita ese parámetro para identificar el documento que tenemos que eliminar.

Así pues en `app.js` tendremos que ver de qué manera podemos capturar ese valor a partir del listado de tareas. Una posible solución podría ser usar la clase de atributos de HTML5 data-*, denominados **atributos de datos modificables** . Permite a la información propietaria ser intercambiada entre el HTML y su representación en el DOM que, a su vez podría ser usada por JS. La propiedad HTMLElement.dataset daría acceso a estos valores. El * de data-* puede ser remplazado por cualquier nombre válido en una sintaxis XML. Por tanto, se podría definir el atributo `data-id="${doc.id}"` al insertar los botones de borrado y edición, quedando el `id` del documento disponible para su uso posterior.

Todo se detalla en el siguiente código que integra la lógica del botón de borrado en la función `onGetTasks`

```
// app.js

// importar funciones de módulo externo firebase.js
import { saveTask, onGetTasks, deleteTask } from './firebase.js'

/* resto de código */

    await onGetTasks((querySnapshot) => {
        tasksContainer.innerHTML = "";
        // bucle que recorre todos los documentos del objeto querySnapshot
        querySnapshot.forEach((doc) => {
            // método .data() convierte el objeto de la DB en un objeto JS
            const task = doc.data();

            // inyectamos el código HTML de forma dinámica
            tasksContainer.innerHTML += `
                <div>
                  <h3 class="h5">${task.title}</h3>
```

```
                    <p>${task.description}</p>
                    <div>
                      <button class="btn-delete" data-
id="${doc.id}">Borrar</button>
                      <button class="btn-edit" data-id="${doc.id}">Editar</button>
                    </div>
                  </div>`;
        });

        const btnsDelete = tasksContainer.querySelectorAll(".btn-delete");
        btnsDelete.forEach((btn) =>
            btn.addEventListener("click", async (e) => {
                try {
                    await deleteTask(e.target.dataset.id);
                } catch (error) {
                    console.log(error);
                }
            });
        );
    });

/* resto de código */
```

Claves del código añadido:

- Definimos un array `btnsDelete` con todos los botones con la clase `btn-delete`
- Recorremos el array y establecemos un manejador del evento click que haga una llamada a la función `deleteTask`
- El parámetro `e.target.dataset.id` recoge el valor del *id* del documento que se guardó en el atributo modificable `data-id="${doc.id}"`. Usamos las propiedad `target` del objeto `event` y desde ahí se captura el *id* con la propiedad `dataset`
- Se usa de Async/await para manejar la llamada a `deleteTask` por operar de forma asíncrona.

Editar las tareas listadas

Igualmente que con el borrado de tareas definiremos las funciones correspondientes en `firebase.js` y las usaremos en `app.js`. En este caso particular hay que hacer previamente que los datos de la tarea a editar se carguen en el formulario que hemos usado para crear nuevas tareas, ya que también lo usaremos para esta funcionalidad. Esto último hace que de alguna manera tengamos que distinguir ambas funcionalidades (crear o actualizar) a la hora de enviar los datos del formulario.

Vamos entonces primeramente a resolver la cuestión de la carga de los datos de la tarea listada en el formulario. Haremos de nuevo uso del procedimiento de captura del *id* de la tarea tal y como lo hicimos a implementar el botón de editar. El resultado se puede ver en el siguiente código:

```
// app.js

// importar funciones de módulo externo firebase.js
```

```
import {
    onGetTasks,
    saveTask,
    deleteTask,
    getTask
  } from "./firebase.js";

// variables globales
  const taskForm = document.getElementById("task-form");
  const tasksContainer = document.getElementById("tasks-container");
  let editStatus = false;
  let id = "";

window.addEventListener("DOMContentLoaded", async (e) => {

    await onGetTasks((querySnapshot) => {
        tasksContainer.innerHTML = "";
        // bucle que recorre todos los documentos del objeto querySnapshot
        querySnapshot.forEach((doc) => {
            // método .data() convierte el objeto de la DB en un objeto JS
            const task = doc.data();

            // inyectamos el código HTML de forma dinámica
            tasksContainer.innerHTML += `
                <div>
                  <h3 class="h5">${task.title}</h3>
                  <p>${task.description}</p>
                  <div>
                    <button class="btn-delete" data-
id="${doc.id}">Borrar</button>
                    <button class="btn-edit" data-id="${doc.id}">Editar</button>
                  </div>
                </div>`;
        });

        const btnsDelete = tasksContainer.querySelectorAll(".btn-delete");
        btnsDelete.forEach((btn) =>
            btn.addEventListener("click", async (e) => {
                try {
                    await deleteTask(e.target.dataset.id);
                } catch (error) {
                    console.log(error);
                }
            });
        });

        const btnsEdit = tasksContainer.querySelectorAll(".btn-edit");
        btnsEdit.forEach((btn) => {
            btn.addEventListener("click", async (e) => {
                try {
```

```
                    const doc = await getTask(e.target.dataset.id);
                    const task = doc.data();
                    taskForm["task-title"].value = task.title;
                    taskForm["task-description"].value = task.description;

                    editStatus = true;
                    id = doc.id;
                    taskForm["btn-task-form"].innerText = "Actualizar";
                } catch (error) {
                    console.log(error);
                }
            });
        });
    });
});

/* resto de código */
```

Vemos que para implementar esta funcionalidad es necesario hacer uso de una función que se ha importado de `firebase.js` denominada `getTask`. Volveremos a ella en un momento, pero antes detallemos las claves para entender este código nuevo. Casi todo funciona igual que en el caso del botón de borrado, pero con alguna salvedad:

- la función `getTask` devuelve un documento de la base de datos con una *id* concreta. A ese documento le aplicamos el método `.data()` para convertirlo en un objeto manejable por JS.
- Una vez tenemos el objeto JS actuamos sobre el DOM para insertar los valores en el formulario.
- Se han definido en la sección variables globales dos variables auxiliares: `editStatus` e `id`. Estas variables nos permitirán distinguir si estamos en un proceso de edición o de creación de un nuevo registro y cual es el *id* del documento que estamos editando.

Volvamos ahora al módulo `firebase.js` para ver como podemos definir la función `getTask`, usada en la lógica del botón de edición. Para implementar esta función es necesario usar los métodos `doc` y `getDoc`. Vamos a verlo en el siguiente código:

```
// firebase.js

// importación SDK de firebase y firestore
import { initializeApp } from
"https://www.gstatic.com/firebasejs/10.7.1/firebase-app.js";
import { getFirestore, collection, addDoc, onSnapshot, deleteDoc, doc, getDoc }
from "https://www.gstatic.com/firebasejs/10.7.1/firebase-firestore.js"

/* resto de código */

export const getTask = (id) => {
    return getDoc(doc(db, "tasks", id));
};
```

Vemos como `getTask` necesita del parámetro *id* para hacer su función. El método `getDoc` extrae el documento al que apunta a su vez el método `doc`

Por último quedaría implementar la escritura de los cambios realizados en la tarea seleccionada en la base de datos. Para ello necesitaremos definir la función correspondiente en `firebase.js` y posteriormente implementarla en `app.js`.

El código a añadir en `firebase.js` sería el siguiente:

```
// firebase.js

// importación SDK de firebase y firestore
import { initializeApp } from "https://www.gstatic.com/firebasejs/10.7.1/firebase-app.js";
import { getFirestore, collection, addDoc, onSnapshot, deleteDoc, doc, getDoc, updateDoc } from "https://www.gstatic.com/firebasejs/10.7.1/firebase-firestore.js"

/* resto de código */

export const updateTask = (id, newFields) => {
      return updateDoc(doc(db, "tasks", id), newFields);
};
```

Vemos que se ha importado el método `updateDoc` que sirve para escribir cambios en la base de datos. Ese método hace uso de dos parámetros: el primero es el documento seleccionado y el segundo es un objeto con los campos que hay que modificar.

Una vez definida la función `updateTask` la importaremos en `app.js` y ajustaremos la sección de envío del formulario para incluirla de la manera conveniente. El código quedaría así:

```
// app.js

// importar funciones de módulo externo firebase.js
import {
    onGetTasks,
    saveTask,
    deleteTask,
    getTask,
    updateTask
} from "./firebase.js";

// variables globales
const taskForm = document.getElementById("task-form");
const tasksContainer = document.getElementById("tasks-container");
let editStatus = false;
let id = "";
```

```
17
18
19  /* resto de código */
20
21
22  taskForm.addEventListener("submit", async (e) => {
23      e.preventDefault();
24
25      const title = taskForm["task-title"];
26      const description = taskForm["task-description"];
27
28      try {
29          if (!editStatus) {
30              await saveTask(title.value, description.value);
31          } else {
32              await updateTask(id, {
33                  title: title.value,
34                  description: description.value,
35              });
36
37              editStatus = false;
38              id = "";
39              taskForm["btn-task-form"].innerText = "Grabar";
40          }
41
42          taskForm.reset();
43          title.focus();
44
45      } catch (error) {
46          console.log(error);
47      }
48  });
```

Vemos como se ha modificado el manejador del evento *submit* del formulario para que distinga si estamos editando un registro o creando uno nuevo. Si la variable `editStatus` es `false` estaremos en un proceso de creación de una nueva tarea y por tanto se llamará a la función `saveTask`. por contra, si `editStatus` es `true`, la llamada se hará a la función `updateTask` para proceder con la actualización del registro. Después de que la actualización se escriba en la base de datos `editStatus` vuelve a hacerse `false` y el valor de la variable `id` se pone a nulo. Por último, sea cual sea el tipo de operación el formulario se reseteará y se pondrá el foco en el primer campo.

Estilos y finalización

Con todo lo anterior habríamos implementado un CRUD en una aplicación de gestión de gestión de tareas básica que usa como almacenamiento de la información la plataforma Firebase de Google. Ciertamente, el hecho de haber separado la lógica de interacción con la base de datos de la lógica de manejo de la aplicación en el frontend hace que cambiar la plataforma de la base de datos se reduzca a trabajar sobre el módulo `firebase.js`, no teniendo que actuar sobre `app.js`. Bastaría con adaptar el código a la nueva base de datos y cambiar las funciones importadas al módulo `app.js`. De la misma forma, adaptar este proyecto CRUD a una aplicación con un modelo de datos distinto, por ejemplo, una aplicación de inventario o una gestión básica de clientes, conllevaría trabajar sobre todo el módulo `app.js`, mientras que los cambios en `firebase.js` serían mínimos.

Por último quedaría aplicar diseños y estilos al proyecto. Se recomienda usar Bootstrap para ello, aunque pueden usarse otras alternativas. Como ejercicio puede intentarse aplicar la misma configuración usada para el proyecto de gestor de tareas que se trabajo en la prueba de evaluación del primer trimestre. Es una tarea sencilla, aunque habría que tener cuidado con renombrar las clases en caso necesario.

Otro recurso interesante puede ser **bootswatch.com**, una web de temas basados en Bootstrap listos para su descarga o uso mediante cdn.

Una versión final del código, con diseño Bootstrap, listo para ejecutar puede descargarse de GitHub.