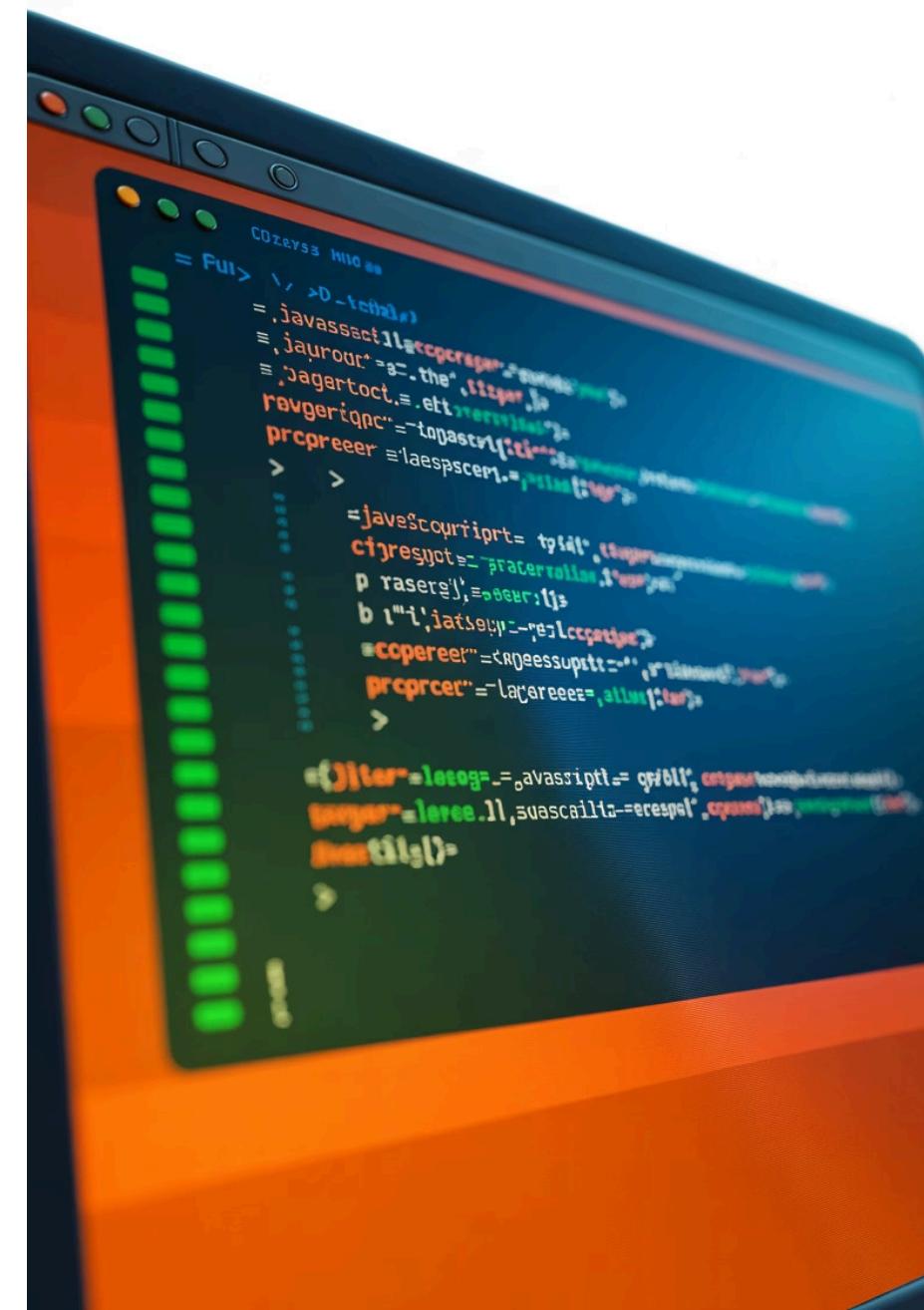


CRUD con JavaScript Vanilla

Una guía completa paso a paso para crear tu primera aplicación CRUD utilizando JavaScript puro, HTML5 y Bootstrap. Aprende desde cero cómo gestionar datos con las operaciones fundamentales de cualquier aplicación web.



¿Qué es un CRUD?

CREATE

Crear nuevos registros en tu base de datos o colección de datos

READ

Leer y mostrar los datos existentes de forma organizada

UPDATE

Actualizar y modificar registros que ya existen

DELETE

Eliminar registros que ya no necesitas

CRUD son las siglas de las cuatro operaciones básicas que se pueden realizar sobre cualquier conjunto de datos persistentes. Estas operaciones son fundamentales en el desarrollo de aplicaciones web y constituyen la base de la mayoría de sistemas de gestión de información que utilizamos diariamente.



Tecnologías del Proyecto

HTML5

Estructura semántica
del documento y
formularios

- Formularios de captura
- Tablas de visualización
- Modal de edición

Bootstrap 5.3

Framework CSS para
estilos y componentes
responsivos

- Sistema de grid
- Componentes UI
- Utilidades CSS

JavaScript Vanilla

Lógica del CRUD sin
dependencias externas

- Manipulación del DOM
- Clases ES6
- Módulos JavaScript

Módulo 1

Configuración del Entorno y Estructura

Antes de escribir código, necesitamos establecer una estructura de carpetas organizada y preparar los archivos básicos del proyecto. Una buena organización desde el inicio facilitará el mantenimiento y escalabilidad de tu aplicación.



Estructura de Carpetas del Proyecto

01

Raíz del proyecto

Crea una carpeta principal que contendrá todo el proyecto. Aquí colocarás el archivo **index.html** como punto de entrada.

02

Carpeta Data

Crea una carpeta llamada **Data** donde guardarás el archivo **Data.json** con los datos iniciales de las Gift Cards.

03

Carpeta js

Crea una carpeta **js** para los scripts de JavaScript. Aquí irán **app.js** (lógica principal) y **clases.js** (definición de objetos).

- **Tip profesional:** Mantener una estructura de carpetas clara desde el inicio te ayudará a escalar el proyecto más adelante. Considera agregar carpetas adicionales como *css* o *images* si tu proyecto crece.

Archivos Necesarios

index.html

Archivo principal HTML que contiene la estructura de la página, los enlaces a Bootstrap y Font Awesome, y las referencias a los scripts de JavaScript.

Data/Data.json

Archivo JSON que almacena los datos iniciales de las Gift Cards. Actúa como una base de datos temporal en memoria.

js/clases.js

Define la clase Gift que sirve como molde para crear objetos con una estructura consistente. Usa programación orientada a objetos.

js/app.js

Contiene toda la lógica del CRUD: funciones para crear, leer, actualizar y eliminar Gift Cards, además de la manipulación del DOM.



Configuración del HTML Base

El archivo **index.html** es el punto de entrada de nuestra aplicación. Aquí establecemos la estructura básica, enlazamos las librerías externas y conectamos nuestros scripts de JavaScript.

Es fundamental entender que utilizamos **type="module"** en la etiqueta script de JavaScript. Esta declaración es obligatoria porque nos permite usar las palabras reservadas **import** y **export** para trabajar con módulos, una característica moderna de JavaScript que facilita la organización del código.

Estructura del HTML: Head

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

  <!-- Bootstrap CSS -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
    rel="stylesheet">

  <!-- Font Awesome para iconos -->
  <script src="https://kit.fontawesome.com/b5a5b5463f.js"
    crossorigin="anonymous"></script>
</head>
```

El **meta viewport** es esencial para que la aplicación sea responsive en dispositivos móviles. Bootstrap proporciona los estilos CSS predefinidos, mientras que Font Awesome nos da acceso a iconos profesionales para los botones de editar y eliminar.

Estructura del HTML: Body y Scripts

```
<body>
  <div class="container">
    <!-- Aquí va el contenido:
        formularios y tabla -->
  </div>

  <!-- Bootstrap JavaScript -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js">
  </script>

  <!-- Nuestro JavaScript principal -->
  <script src=".js/app.js" type="module">
  </script>
</body>
</html>
```

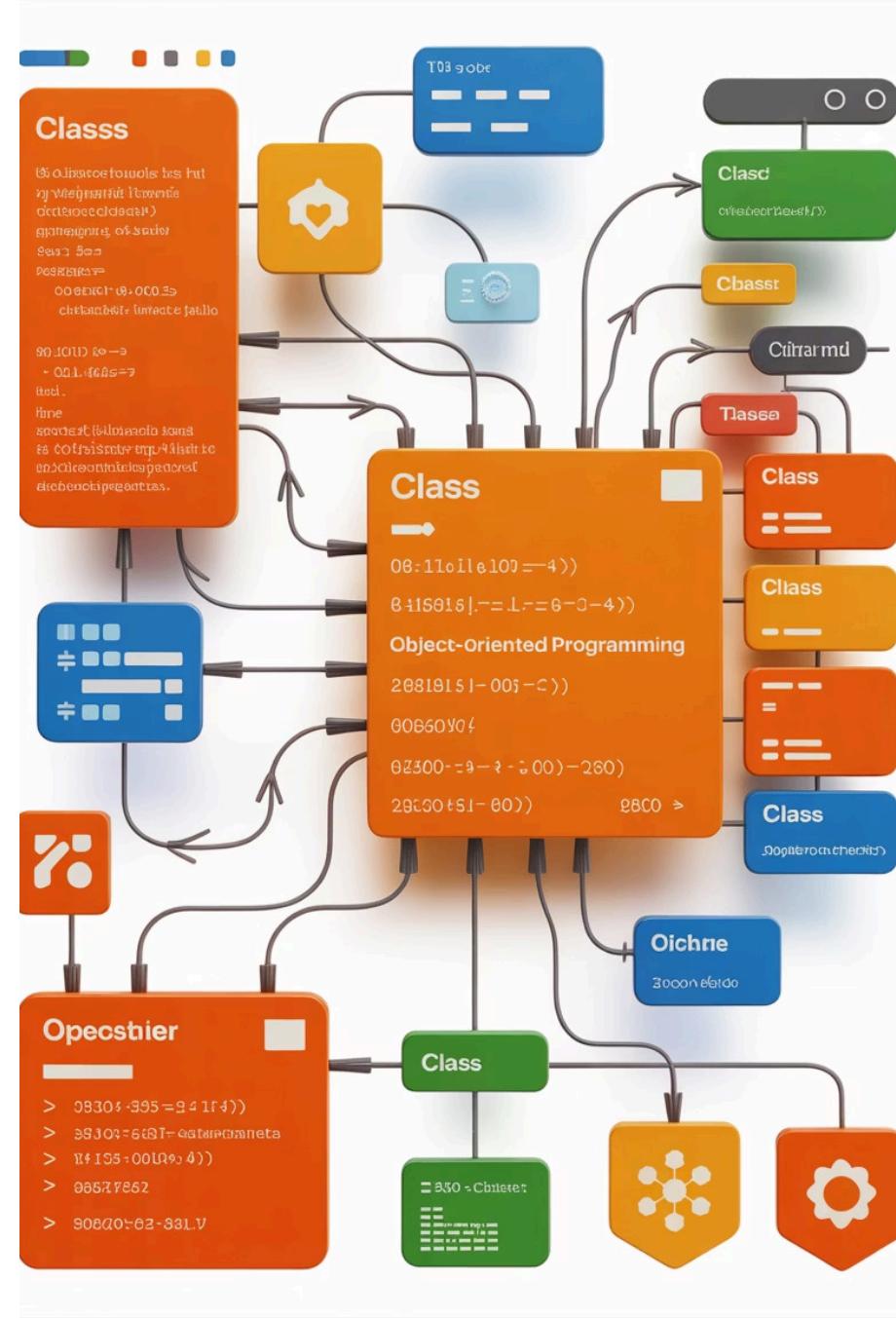
La clase **container** de Bootstrap centra el contenido y aplica márgenes responsivos. El script de Bootstrap debe cargarse antes que nuestro app.js porque necesitamos sus funcionalidades para el modal.

- **Importante:** El atributo **type="module"** es la clave que permite usar import/export. Sin él, el navegador generará errores al intentar importar la clase Gift o los datos JSON.

Módulo 2

Definición de la Clase Gift

La Programación Orientada a Objetos (POO) nos permite crear "moldes" o plantillas que definen la estructura de nuestros datos. En este proyecto, la clase **Gift** asegura que todas las Gift Cards tengan exactamente las mismas propiedades.



¿Qué es una Clase en JavaScript?

Concepto

Una clase es como un molde de galletas: define la forma y características que tendrán todos los objetos creados a partir de ella.

Cuando creamos una nueva Gift Card usando **new Gift(...)**, estamos usando ese molde para crear un objeto con propiedades específicas.

Beneficios

- **Consistencia:** Todos los objetos tienen la misma estructura
- **Claridad:** Sabemos qué propiedades esperamos
- **Mantenibilidad:** Cambios centralizados
- **Escalabilidad:** Fácil agregar métodos

Código Completo de clases.js

```
// La palabra 'export' permite usar esta clase
// en otros archivos JavaScript
export class Gift {

    // El 'constructor' se ejecuta automáticamente
    // cuando creamos un objeto con 'new Gift(...)'
    constructor(id, gift, tipo, tiempo, precio, imagen) {

        // 'this' hace referencia al objeto específico
        // que se está creando en este momento
        this.id = id;
        this.gift = gift;
        this.tipo = tipo;
        this.tiempo = tiempo;
        this.precio = precio;
        this.imagen = imagen;
    }
}
```

Palabras Reservadas de la Clase



export

Palabra clave que hace que la clase esté disponible para ser importada en otros módulos JavaScript. Sin export, no podríamos usar Gift en app.js.



class

Define una clase en JavaScript ES6. Es la estructura fundamental de la Programación Orientada a Objetos en JavaScript moderno.



constructor

Método especial que se ejecuta automáticamente cuando creamos una instancia de la clase con el operador new. Inicializa las propiedades del objeto.



this

Hace referencia al objeto actual que se está creando. Permite asignar valores a las propiedades específicas de cada instancia individual.

Ejemplo de Uso de la Clase

Crear una instancia

```
// Importar la clase
import { Gift } from './clases.js';

// Crear un nuevo objeto
const miGift = new Gift(
  1,
  "Spotify Premium",
  "Suscripción",
  "Un mes",
  150,
  "url-imagen"
);

// Acceder a propiedades
console.log(miGift.gift);
// Output: "Spotify Premium"
```

Lo que sucede internamente

1. Se ejecuta el constructor
2. Se crea un objeto vacío
3. this apunta a ese objeto
4. Se asignan las propiedades
5. Se retorna el objeto completo

El resultado es un objeto estructurado con todas las propiedades necesarias para representar una Gift Card.

Módulo 3

Lógica Principal: app.js

El archivo **app.js** es el cerebro de nuestra aplicación. Aquí reside toda la lógica del CRUD: importamos datos y clases, manipulamos el DOM, y gestionamos eventos del usuario. Este módulo conecta todos los componentes del proyecto.



Imports y Variables Globales

```
// Importar la clase Gift desde clases.js
import { Gift } from "./clases.js";

// Importar datos del archivo JSON
// 'assert { type: "json" }' es obligatorio para JSON
import datos from "./Data/Data.json" assert { type: "json" };

// Variable global para el ID del gift en edición
let idGiftUpdate = null;

// Capturar elemento del DOM: cuerpo de la tabla
const cuerpoTabla = document.querySelector("#cuerpo-tabla");

// Inicializar el modal de Bootstrap
constmyModal = new bootstrap.Modal(
  document.getElementById("modal-gift")
);
```

La variable **idGiftUpdate** es crucial: almacena temporalmente el ID del elemento que estamos editando para poder localizarlo cuando el usuario confirme los cambios en el modal.

Import de Módulos en JavaScript



Exportar

En clases.js usamos **export class Gift** para hacer la clase disponible

Importar

En app.js usamos **import { Gift } from...** para traer la clase

Usar

Ya podemos usar **new Gift(...)** para crear objetos

- ❑ **Importante:** Para importar archivos JSON, debemos usar **assert { type: "json" }**. Esta sintaxis le indica al navegador que el archivo importado es de formato JSON y no JavaScript regular.

document.querySelector

¿Qué hace?

Es un método del objeto **document** (DOM) que busca y retorna el primer elemento HTML que coincide con el selector CSS proporcionado.

Si no encuentra ningún elemento, retorna **null**.

Ejemplos de uso

```
// Por ID (más común)
const tabla = document.querySelector("#cuerpo-tabla");
```

```
// Por clase
const botones = document.querySelector(".btn-primary");
```

```
// Por etiqueta
const parrafo = document.querySelector("p");
```

```
// Combinado
const input = document.querySelector("input#nombre");
```

En nuestro proyecto, usamos `querySelector` principalmente con IDs (`#`) porque cada elemento tiene un identificador único que facilita su localización en el DOM.

El Objeto Modal de Bootstrap

```
// Crear instancia del modal
constmyModal = new bootstrap.Modal(
  document.getElementById("modal-gift")
);

// Métodos principales del modal
myModal.show(); // Abre/muestra el modal
myModal.hide(); // Cierra/oculta el modal
```

Bootstrap proporciona componentes JavaScript listos para usar. El **Modal** es un cuadro de diálogo que aparece sobre el contenido principal.

Necesitamos crear una instancia del modal para poder controlarlo programáticamente desde JavaScript.

- **new bootstrap.Modal:** Constructor del modal
- **document.getElementById:** Busca el elemento por ID
- **.show():** Método para mostrar el modal
- **.hide():** Método para ocultar el modal

READ

Función cargarTabla()

Esta función es el corazón de la operación **READ** del CRUD. Se encarga de leer el arreglo de datos y renderizar dinámicamente cada fila de la tabla en el DOM. Se ejecuta al iniciar la aplicación y después de cada operación CREATE, UPDATE o DELETE para mantener la vista sincronizada.



Código Completo de cargarTabla()

```
const cargarTabla = () => {
  // Limpiar tabla antes de cargar
  cuerpoTabla.innerHTML = "";

  // Recorrer el arreglo de datos
  datos.map((item) => {
    // Crear fila de tabla
    const fila = document.createElement("tr");

    // Crear contenido HTML de las celdas
    const celdas = `
      <td>${item.gift}</td>
      <td>${item.tipo}</td>
      <td>${item.tiempo}</td>
      <td>${item.precio}</td>
      <td>
        <div class="d-flex gap-2">
          <button class="btn btn-outline-warning"
            onclick="window.MostrarModal(${item.id})">
            <i class="fas fa-pencil-alt"></i>
          </button>
          <button class="btn btn-outline-danger"
            onclick="window.BorrarGift(${item.id})">
            <i class="fas fa-times"></i>
          </button>
        </div>
      </td>
    `;

    fila.innerHTML = celdas;
    cuerpoTabla.appendChild(fila);
  });
};

// Ejecutar al cargar la página
cargarTabla();
```

Métodos del DOM Utilizados



`.innerHTML = ""`

Limpia completamente el contenido HTML del elemento. Esencial para evitar que los datos se dupliquen cada vez que recargamos la tabla.



`.innerHTML = contenido`

Asigna contenido HTML a un elemento. Podemos usar strings de texto o template strings para generar HTML dinámicamente.



`document.createElement()`

Crea un nuevo elemento HTML en memoria (no visible aún). Recibe como parámetro el nombre de la etiqueta, en este caso "tr" para fila de tabla.



`.appendChild()`

Agrega un elemento hijo a un elemento padre. Aquí usamos para insertar cada fila (tr) al cuerpo de la tabla (tbody).

Array.map() - Iteración del Arreglo

Sintaxis

```
array.map((item) => {  
  // Código que se ejecuta  
  // por cada elemento  
  return resultado;  
});
```

El método **map()** recorre cada elemento del arreglo y ejecuta la función proporcionada. Aunque map() normalmente retorna un nuevo arreglo, aquí lo usamos por su capacidad de iteración.

En nuestro código

```
datos.map((item) => {  
  // 'item' representa cada  
  // Gift Card del arreglo  
  
  // Podemos acceder a:  
  // item.id  
  // item.gift  
  // item.tipo  
  // item.tiempo  
  // item.precio  
  // item.imagen  
});
```

En cada iteración, **item** toma el valor de uno de los objetos Gift del arreglo. Por ejemplo, en la primera iteración item es el primer objeto, en la segunda es el segundo objeto, y así sucesivamente.

Template Strings (Literals)

Los template strings utilizan **backticks** (`) en lugar de comillas y permiten incrustar expresiones JavaScript usando la sintaxis `${expresión}`. Son extremadamente útiles para generar HTML dinámico.

Sintaxis tradicional

```
// Concatenación con +
const nombre = "Juan";
const edad = 25;

const mensaje =
  "Hola " + nombre +
  ", tienes " + edad +
  " años";

// Difícil de leer y mantener
```

Con template strings

```
// Interpolación con ${}
const nombre = "Juan";
const edad = 25;

const mensaje =
  `Hola ${nombre},
  tienes ${edad} años`;

// Más limpio y legible
```

- Dentro de `${}` podemos poner cualquier expresión JavaScript válida: variables, operaciones matemáticas, llamadas a funciones, acceso a propiedades de objetos, etc.

window y el Atributo onclick

```
<button onclick="window.MostrarModal(${item.id})">  
  Editar  
</button>
```

01

El problema

Cuando usamos **type="module"** en nuestro script, las funciones definidas en app.js no son globales por defecto. No podemos llamarlas directamente desde onclick.

02

La solución

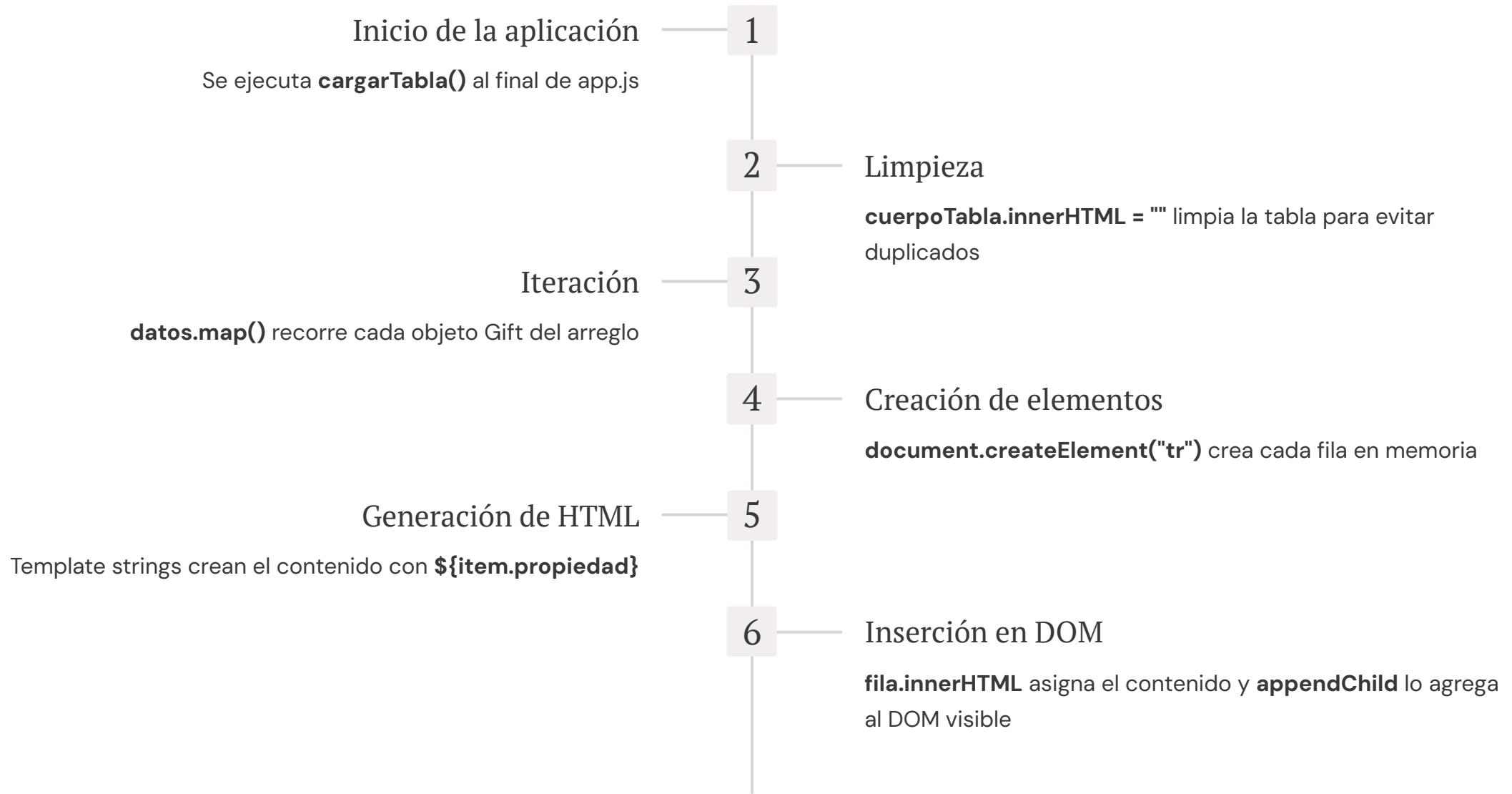
Usar **window.nombreFuncion** convierte la función en un método del objeto global window (Browser Object Model), haciéndola accesible desde HTML.

03

En el código

Definimos: **window.MostrarModal = (id) => {...}** para que pueda ser llamada con **onclick="window.MostrarModal(...)"**

Flujo Completo de READ





DELETE

Función BorrarGift()

La operación **DELETE** es una de las más directas del CRUD. Esta función localiza un elemento específico por su ID, solicita confirmación al usuario y, si se confirma, elimina el elemento del arreglo y actualiza la vista.

Código Completo de BorrarGift()

```
// Función global para ser llamada desde onclick
window.BorrarGift = (id) => {

    // Encontrar la posición del elemento en el arreglo
    const index = datos.findIndex((item) => item.id === id);

    // Mostrar diálogo de confirmación
    const validar = confirm(
        `¿Está seguro que quiere eliminar la gift Card ${datos[index].gift}?`
    );

    // Si el usuario confirma (presiona Aceptar)
    if (validar) {
        // Eliminar 1 elemento en la posición 'index'
        datos.splice(index, 1);

        // Recargar la tabla para mostrar cambios
        cargarTabla();
    }
};
```

Esta función debe ser global usando **window.BorrarGift** porque es llamada desde el atributo onclick de los botones de eliminar en la tabla HTML.

Array.findIndex() - Localizar Elementos

¿Qué hace?

Busca en el arreglo el **primer elemento** que cumpla con la condición especificada y retorna su **índice** (posición en el arreglo).

Si no encuentra ningún elemento, retorna **-1**.

```
const index = datos.findIndex(  
  (item) => item.id === id  
);  
  
// Retorna un número: 0, 1, 2, 3...  
// o -1 si no encuentra nada
```

En nuestro contexto

- **datos:** El arreglo donde buscamos
- **item:** Cada objeto durante la búsqueda
- **item.id === id:** Condición de búsqueda
- **index:** Posición del elemento encontrado

Ejemplo: si el elemento con id=2 está en la tercera posición del arreglo, findIndex retornará 2 (los índices empiezan en 0).

confirm() - Diálogo de Confirmación

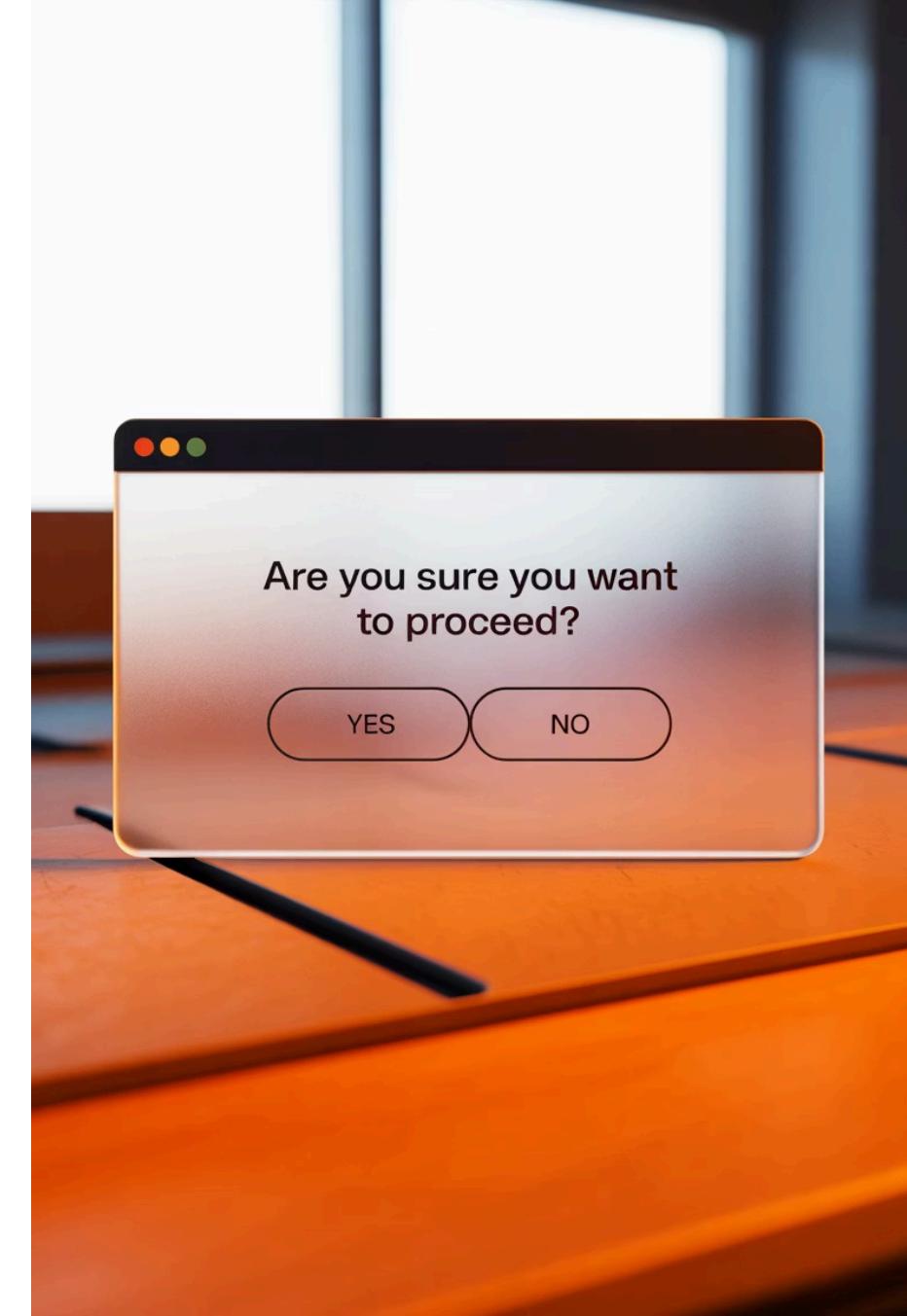
El método **confirm()** es parte del Browser Object Model (BOM) y muestra un cuadro de diálogo modal con un mensaje y dos botones: Aceptar y Cancelar.

Comportamiento

- Bloquea la ejecución hasta que el usuario responda
- Retorna **true** si presiona Aceptar
- Retorna **false** si presiona Cancelar
- Pausa el resto del código JavaScript

En nuestro código

```
const validar = confirm(`¿Está seguro...?`);  
  
if (validar) {  
    // Usuario presionó Aceptar  
    // Ejecutar eliminación  
} else {  
    // Usuario presionó Cancelar  
    // No hacer nada  
}
```



Array.splice() - Modificar Arreglos

El método **splice()** es una herramienta poderosa que modifica el arreglo original eliminando, reemplazando o agregando elementos. En nuestro caso, lo usamos exclusivamente para eliminar.

```
// Sintaxis general  
array.splice(inicio, cantidadAEliminar, ...elementosAAgregar);  
  
// En nuestro código  
datos.splice(index, 1);  
  
// 'index': posición donde comenzar  
// '1': eliminar 1 elemento  
// Sin tercer parámetro: no agregar nada
```

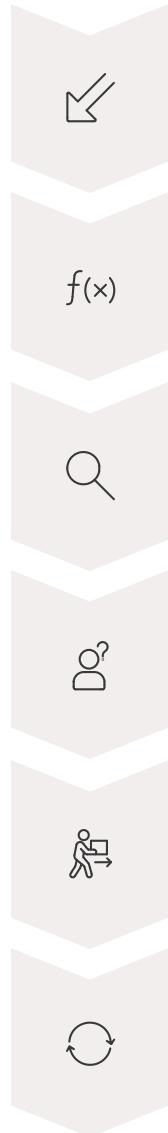
Antes de splice

```
datos = [{id:1}, {id:2}, {id:3}]  
index = 1
```

Después de splice(1, 1)

```
datos = [{id:1}, {id:3}]  
El elemento en posición 1 fue eliminado
```

Flujo Completo de DELETE



Click en botón

Usuario hace click en el botón de eliminar

Llamada a función

Se ejecuta window.BorrarGift(id)

Localizar

findIndex() busca la posición del elemento

Confirmar

confirm() pide confirmación al usuario

Eliminar

splice() elimina el elemento del arreglo

Actualizar vista

cargarTabla() re-renderiza la tabla



CREATE

Función agregarGift()

La operación **CREATE** permite agregar nuevas Gift Cards a nuestro sistema. Esta función captura los datos del formulario, genera un nuevo ID único, crea una instancia de la clase Gift y la agrega al arreglo de datos.

Event Listener del Formulario

```
// Capturar el formulario por su ID  
const formAgregar = document.querySelector("#form-gift");  
  
// Escuchar el evento 'submit' del formulario  
formAgregar.addEventListener("submit", agregarGift);
```

1

querySelector

Selecciona el formulario HTML usando su ID único #form-gift

2

addEventListener

Registra una función que se ejecutará cuando ocurra el evento especificado

3

"submit"

El tipo de evento a escuchar. Se dispara cuando se envía el formulario

4

agregarGift

La función callback que se ejecutará al enviar el formulario

Código Completo de agregarGift()

```
const agregarGift = (e) => {
  // Prevenir el comportamiento por defecto del formulario
  e.preventDefault();

  // Generar nuevo ID (último ID + 1)
  const id = datos.at(-1).id + 1;

  // Capturar valores de los inputs
  const gift = document.querySelector("#gift").value;
  const tipo = document.querySelector("#tipo").value;
  const tiempo = document.querySelector("#tiempo").value;
  const precio = document.querySelector("#precio").value;
  const imagen = document.querySelector("#imagen").value;

  // Agregar nuevo objeto Gift al arreglo
  datos.push(new Gift(id, gift, tipo, tiempo, precio, imagen));

  // Limpiar todos los campos del formulario
  formAgregar.reset();

  // Actualizar la tabla para mostrar el nuevo elemento
  cargarTabla();
};
```

e.preventDefault() - Control del Evento

El problema

Por defecto, cuando se envía un formulario HTML con un botón de tipo **submit**, el navegador:

1. Recarga la página completa
2. Envía los datos a una URL (action)
3. Pierde el estado de la aplicación

Esto haría que perdiéramos todos nuestros datos porque están en memoria.

La solución

```
const agregarGift = (e) => {
  // 'e' es el objeto Event
  e.preventDefault();

  // Ahora controlamos nosotros
  // qué sucede al enviar
};
```

preventDefault() cancela el comportamiento por defecto del navegador, permitiéndonos manejar el envío del formulario con JavaScript.

Generación de ID Único

```
const id = datos.at(-1).id + 1;
```

01

`datos.at(-1)`

El método **at(-1)** accede al último elemento del arreglo. Los índices negativos cuentan desde el final: -1 es el último, -2 el penúltimo, etc.

02

`.id`

Accede a la propiedad **id** de ese último objeto. Por ejemplo, si el último gift tiene id: 3, obtenemos el valor 3.

03

`+ 1`

Suma 1 al ID del último elemento. Esto garantiza que el nuevo ID sea único y secuencial. Si el último era 3, el nuevo será 4.

Alternativa: También podríamos usar `datos[datos.length - 1].id + 1`, pero **at(-1)** es más moderno y legible.

Captura de Valores con .value

La propiedad **.value** de los elementos de formulario (input, select, textarea) contiene el valor actual que el usuario ha ingresado o seleccionado.

Proceso de captura

```
// 1. Seleccionar el input por ID  
const inputElement =  
  document.querySelector("#gift");  
  
// 2. Obtener su valor  
const gift = inputElement.value;  
  
// O en una sola línea:  
const gift =  
  document.querySelector("#gift").value;
```

Todos los campos

```
const gift =  
  document.querySelector("#gift").value;  
const tipo =  
  document.querySelector("#tipo").value;  
const tiempo =  
  document.querySelector("#tiempo").value;  
const precio =  
  document.querySelector("#precio").value;  
const imagen =  
  document.querySelector("#imagen").value;
```

Cada variable ahora contiene el string de texto que el usuario escribió en el input correspondiente.

Array.push() y new Gift()

```
datos.push(new Gift(id, gift, tipo, tiempo, precio, imagen));
```

new Gift(...)

El operador **new** crea una nueva instancia de la clase `Gift`. Ejecuta el constructor pasándole los parámetros en orden. Retorna un objeto nuevo con todas las propiedades.

datos.push(...)

El método **push()** agrega uno o más elementos al final del arreglo. Modifica el arreglo original y retorna la nueva longitud del arreglo.

Antes de push

```
datos = [  
  {id: 1, gift: "Spotify", ...},  
  {id: 2, gift: "Xbox", ...}  
]
```

Después de push

```
datos = [  
  {id: 1, gift: "Spotify", ...},  
  {id: 2, gift: "Xbox", ...},  
  {id: 3, gift: "Netflix", ...}  
]
```

form.reset() - Limpiar Formulario

```
formAgregar.reset();
```

El método **reset()** es una funcionalidad nativa de los formularios HTML. Restaura todos los campos del formulario a sus valores por defecto (generalmente vacíos).

Comportamiento

- Limpia todos los inputs de texto
- Desmarca checkboxes y radio buttons
- Restaura selects a su primera opción
- No requiere parámetros

Ventaja

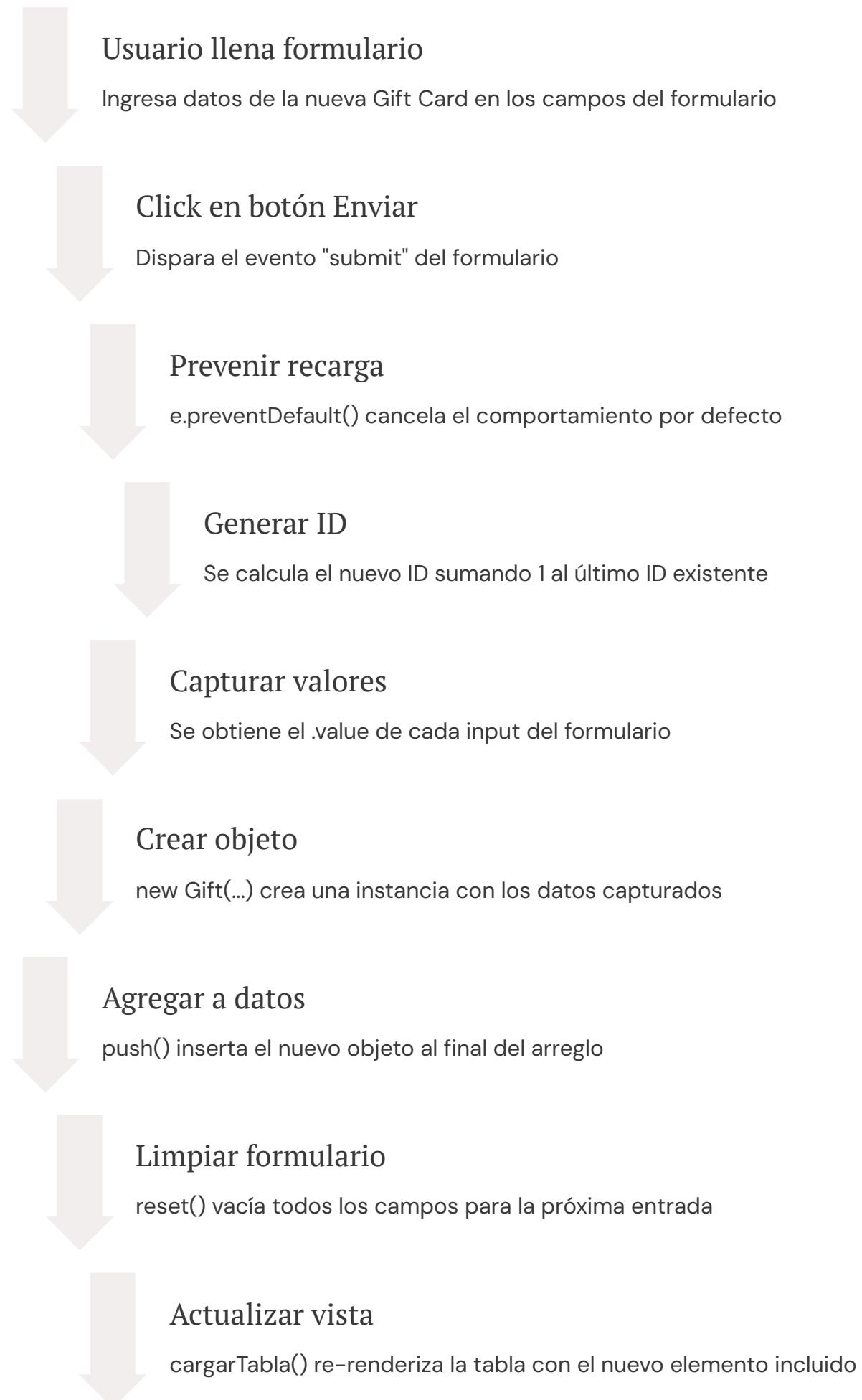
En lugar de limpiar campo por campo manualmente:

```
document.querySelector("#gift").value = "";
document.querySelector("#tipo").value = "";
// etc...
```

Usamos una sola línea: `formAgregar.reset();`



Flujo Completo de CREATE



UPDATE

Edición de Datos (Parte 1)

La operación **UPDATE** es la más compleja del CRUD porque requiere dos pasos: primero mostrar los datos actuales en un modal de edición, y luego guardar los cambios. Comenzaremos con la función que prepara y muestra el modal.



¿Por qué usar un Modal para UPDATE?

Ventajas del Modal

- **Separación visual:** El usuario sabe que está en "modo edición"
- **Evita confusión:** Formulario de agregar vs. formulario de editar separados
- **Mejor UX:** Mantiene contexto de la tabla visible
- **Confirmación clara:** Botones distintos (Aregar vs Actualizar)

Componentes necesarios

1. **HTML del modal:** Estructura Bootstrap con formulario
2. **Variable global:** idGiftUpdate para recordar qué editamos
3. **MostrarModal():** Función que carga datos y abre modal
4. **giftUpdate():** Función que guarda los cambios

Código de MostrarModal() - Parte 1

```
// Función global para ser llamada desde onclick
window.MostrarModal = (id) => {

    // PASO 1: Guardar el ID globalmente
    // Necesitamos recordar qué elemento estamos editando
    // para cuando el usuario presione "Actualizar"
    idGiftUpdate = id;

    // PASO 2: Encontrar el elemento en el arreglo
    // Localizamos la posición (index) del objeto que tiene este ID
    const index = datos.findIndex((item) => item.id === id);

    // Continuación en siguiente slide...
};

};
```

La variable **idGiftUpdate** fue declarada como **let** al inicio de app.js. Al ser global, podemos acceder a ella desde cualquier función del módulo, permitiéndonos "recordar" qué elemento estamos editando.

Código de MostrarModal() - Parte 2

```
window.MostrarModal = (id) => {
    idGiftUpdate = id;
    const index = datos.findIndex((item) => item.id === id);

    // PASO 3: Pre-llenar los inputs del modal
    // Asignamos los valores actuales del objeto a cada input
    // Nota: los inputs del modal tienen el sufijo "-modal"
    document.querySelector("#gift-modal").value = datos[index].gift;
    document.querySelector("#tipo-modal").value = datos[index].tipo;
    document.querySelector("#tiempo-modal").value = datos[index].tiempo;
    document.querySelector("#precio-modal").value = datos[index].precio;
    document.querySelector("#imagen-modal").value = datos[index].imagen;

    // PASO 4: Mostrar el modal
    // Usamos el método .show() del objeto Bootstrap Modal
   myModal.show();
};
```

Pre-llenado de Campos del Modal

¿Por qué pre-lleñar?

Cuando el usuario hace click en "Editar", queremos que vea los datos actuales de la Gift Card en el formulario del modal.

Esto proporciona contexto y permite editar solo lo necesario sin tener que reescribir todo desde cero.

Proceso técnico

```
// Leer del arreglo  
const valorActual = datos[index].gift;  
  
// Asignar al input del modal  
document.querySelector("#gift-modal")  
    .value = valorActual;  
  
// El input ahora muestra  
// el valor actual
```

- **Convención de nombres:** Los inputs del modal tienen el sufijo **-modal** (ej: #gift-modal) para diferenciarlos de los inputs del formulario principal (#gift). Esto evita confusiones en el código.

Diagrama: Flujo de MostrarModal()





Edición de Datos (Parte 2): giftUpdate()

Una vez que el usuario modifica los datos en el modal y presiona "Actualizar", necesitamos capturar los nuevos valores, actualizar el objeto en el arreglo y cerrar el modal.

Event Listener del Formulario Modal

```
// Capturar el formulario dentro del modal  
const formModal = document.querySelector("#form-modal");  
  
// Escuchar el evento 'submit' del formulario modal  
formModal.addEventListener("submit", giftUpdate);
```

Similar al formulario de agregar, el formulario del modal también tiene un evento **submit** que se dispara cuando el usuario presiona el botón "Actualizar". Este listener debe configurarse al inicio de app.js.

Diferencia clave

formAgregar → agregarGift() → CREATE

formModal → giftUpdate() → UPDATE

Son dos formularios distintos con propósitos diferentes, cada uno con su propio event listener y función manejadora.

Código Completo de giftUpdate()

```
const giftUpdate = (e) => {
  // Prevenir recarga de página
  e.preventDefault();

  // Localizar el elemento usando el ID almacenado globalmente
  const index = datos.findIndex((item) => item.id === idGiftUpdate);

  // Actualizar cada propiedad del objeto en el arreglo
  // con los nuevos valores capturados del modal
  datos[index].gift = document.querySelector("#gift-modal").value;
  datos[index].tipo = document.querySelector("#tipo-modal").value;
  datos[index].tiempo = document.querySelector("#tiempo-modal").value;
  datos[index].precio = document.querySelector("#precio-modal").value;
  datos[index].imagen = document.querySelector("#imagen-modal").value;

  // Recargar la tabla para mostrar los cambios
  cargarTabla();

  // Cerrar el modal
 myModal.hide();
};
```

Actualización Directa del Objeto

```
datos[index].gift = document.querySelector("#gift-modal").value;
```

01

datos[index]

Accede al objeto en la posición **index** del arreglo. Este es el objeto que estamos editando.

02

.gift =

Accede a la propiedad **gift** del objeto y se prepara para asignarle un nuevo valor.

03

document.querySelector("#gift-modal").value

Obtiene el nuevo valor que el usuario escribió en el input del modal.

Valor antes

```
datos[2].gift = "Spotify Premium"
```

Valor después

```
datos[2].gift = "Spotify Premium Dúo"
```

¿Por qué usamos idGiftUpdate?

El problema

Las funciones **MostrarModal()** y **giftUpdate()** se ejecutan en momentos diferentes:

1. MostrarModal() cuando el usuario hace click en editar
2. giftUpdate() cuando el usuario presiona Actualizar

¿Cómo "recordar" qué elemento estamos editando entre estas dos funciones?

La solución

```
// Variable global (inicio de app.js)
let idGiftUpdate = null;

// Función 1: Guardar ID
window.MostrarModal = (id) => {
  idGiftUpdate = id; // Guardar
  ...
}

// Función 2: Usar ID guardado
const giftUpdate = (e) => {
  const index = datos.findIndex(
    item => item.id === idGiftUpdate
  );
  ...
}
```

myModal.hide() - Cerrar el Modal

```
myModal.hide();
```

El método **hide()** del objeto Bootstrap Modal cierra y oculta el modal de edición. Es el opuesto de **myModal.show()**.

Cuándo llamarlo

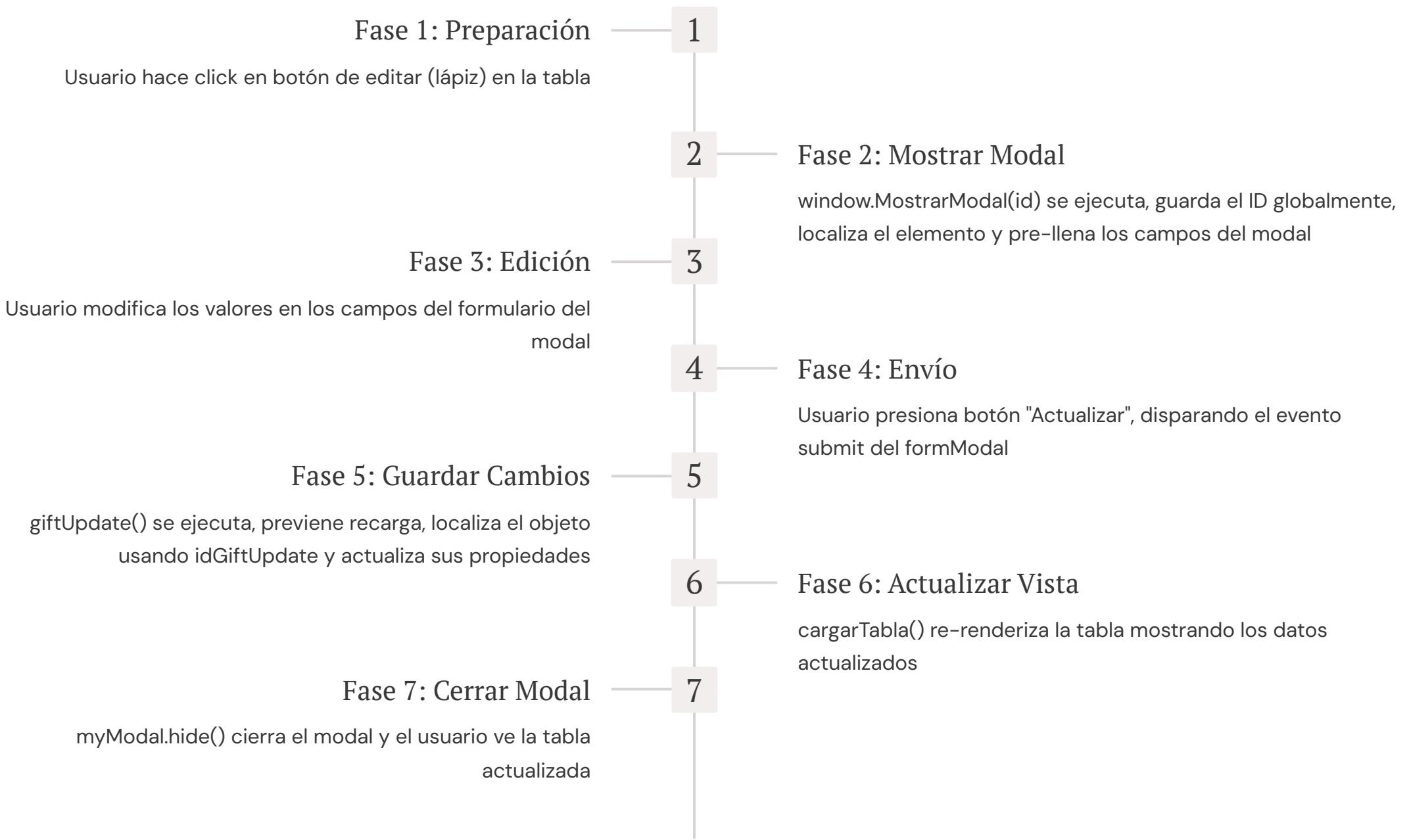
Se ejecuta **después** de actualizar el arreglo datos y recargar la tabla. El usuario necesita ver los cambios reflejados.

Qué hace

- Cierra visualmente el modal
- Restaura el scroll de la página
- Remueve el backdrop oscuro
- Devuelve el foco al documento

-  **Importante:** No es necesario limpiar los campos del modal con reset() porque la próxima vez que se abra, MostrarModal() pre-llenará los campos con los datos del nuevo elemento seleccionado.

Flujo Completo de UPDATE

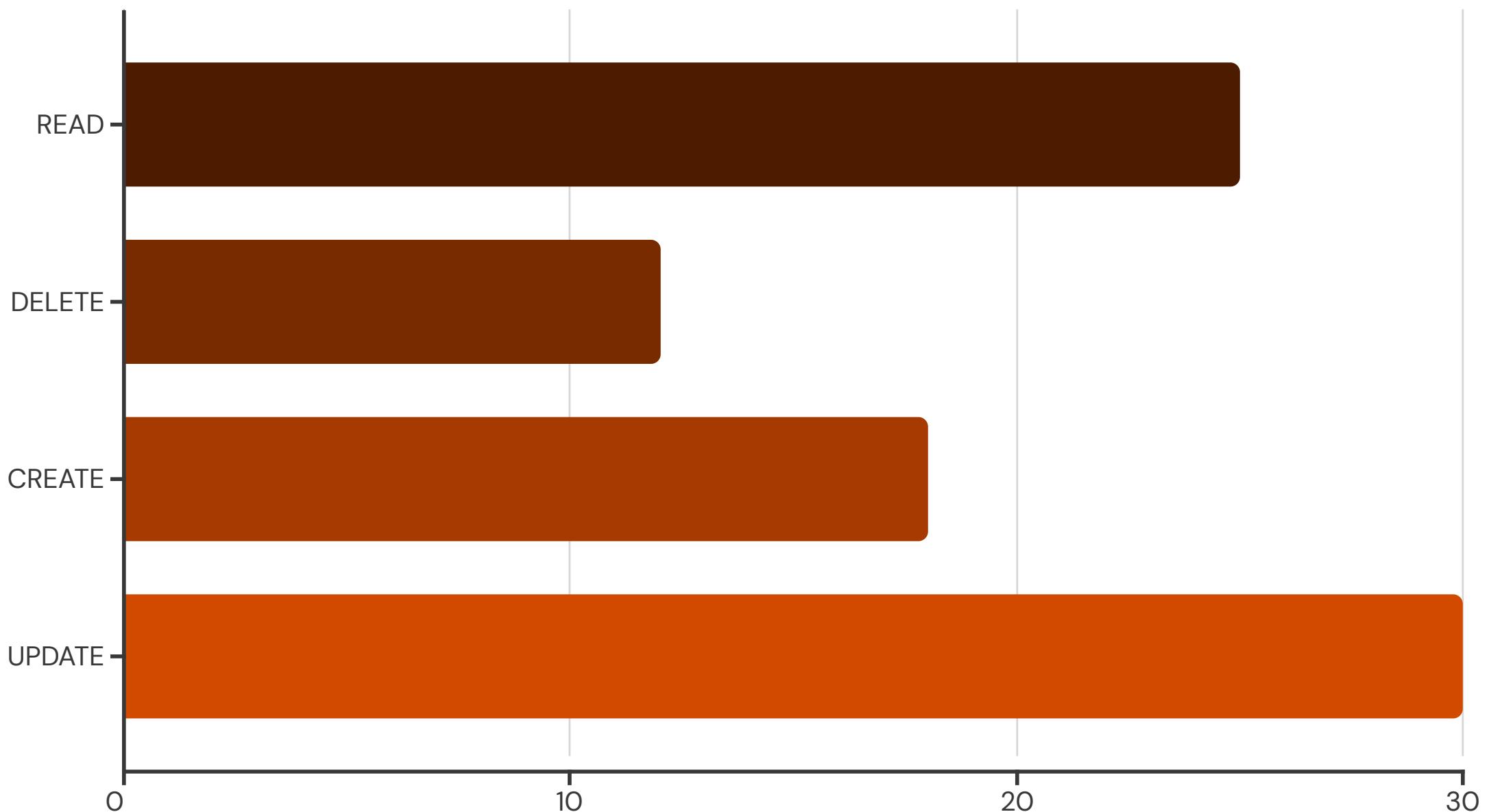


Resumen

Las 4 Operaciones del CRUD



Comparación de las Operaciones CRUD



La operación UPDATE es la más compleja porque requiere dos funciones (MostrarModal y giftUpdate) y manejo de estado global. DELETE es la más simple. READ es fundamental porque se ejecuta después de cada modificación.

Métodos de Array Utilizados



Operación: READ

Uso: Iterar sobre el arreglo datos para generar el HTML de cada fila de la tabla.



.push()

Operación: CREATE

Uso: Agregar un nuevo objeto Gift al final del arreglo datos.



.findIndex()

Operación: UPDATE y DELETE

Uso: Localizar la posición de un elemento específico por su ID.



.splice()

Operación: DELETE

Uso: Eliminar un elemento del arreglo en una posición específica.



.at(-1)

Operación: CREATE

Uso: Acceder al último elemento del arreglo para obtener el último ID.

Métodos del DOM Utilizados

`document.querySelector()`

Busca y retorna el primer elemento del DOM que coincida con el selector CSS especificado. Usado extensivamente para capturar inputs, formularios y contenedores.

`.innerHTML`

Propiedad que permite leer o asignar contenido HTML a un elemento. Usado para limpiar la tabla y para asignar el contenido de las celdas.

`.value`

Propiedad que contiene el valor actual de un elemento de formulario. Usado en CREATE y UPDATE para capturar y asignar datos.

`document.createElement()`

Crea un nuevo elemento HTML en memoria. Usado en READ para crear filas de tabla (`<tr>`) antes de insertarlas en el DOM.

`.appendChild()`

Agrega un elemento hijo al final de un elemento padre. Usado para insertar cada fila creada al cuerpo de la tabla.

`.addEventListener()`

Registra una función que se ejecutará cuando ocurra un evento específico. Usado para escuchar los eventos submit de los formularios.

Palabras Reservadas de JavaScript



import / export

Permiten compartir código entre módulos. export hace disponible una clase o función, import la trae a otro archivo.



class

Define una clase en JavaScript ES6. Estructura fundamental de la Programación Orientada a Objetos.



constructor

Método especial de una clase que se ejecuta automáticamente al crear una instancia con new.



this

Hace referencia al objeto actual. En el constructor, apunta al objeto que se está creando.



new

Operador que crea una nueva instancia de una clase, ejecutando su constructor.



const / let

Declaran variables. const para valores inmutables, let para valores que pueden cambiar.

Eventos y el BOM

Eventos

- **submit:** Se dispara al enviar un formulario
- **click:** Se dispara al hacer click en un elemento
- **e.preventDefault():** Cancela el comportamiento por defecto

Browser Object Model

- **window:** Objeto global del navegador
- **confirm():** Muestra diálogo de confirmación
- **window.funcion:** Hace función global

Bootstrap JS

- **bootstrap.Modal:** Clase del modal
- **.show():** Abre el modal
- **.hide():** Cierra el modal



Conceptos Avanzados

Más Allá del CRUD Básico

El proyecto que hemos construido es un excelente punto de partida, pero trabaja completamente en el lado del cliente (frontend) con datos en memoria. A continuación exploraremos conceptos para llevar tu aplicación al siguiente nivel.

Limitaciones del CRUD Actual



Datos en Memoria

Los datos están almacenados en un arreglo JavaScript en memoria. Al recargar la página, todos los cambios se pierden y volvemos a los datos del JSON inicial.



Usuario Único

Cada usuario ve sus propios cambios localmente. No hay forma de compartir datos entre diferentes usuarios o dispositivos.



Solo Frontend

Toda la lógica está en el navegador. No hay servidor que procese peticiones, valide datos o gestione autenticación.



Sin Persistencia

No hay una base de datos real. El archivo JSON solo proporciona datos iniciales y no se modifica con nuestras operaciones CRUD.



Arquitectura Cliente-Servidor



Cliente (Frontend)

Interfaz de usuario en el navegador. Hace peticiones al servidor.

API (Intermediario)

Punto de comunicación entre cliente y servidor. Define endpoints.

Servidor (Backend)

Procesa lógica de negocio y gestiona acceso a la base de datos.

Base de Datos

Almacena datos de forma persistente. PostgreSQL, MySQL, MongoDB, etc.

Conexión a Bases de Datos: PostgreSQL

Para hacer que nuestro CRUD sea persistente y funcione con una base de datos real como **PostgreSQL**, necesitamos agregar un backend completo.

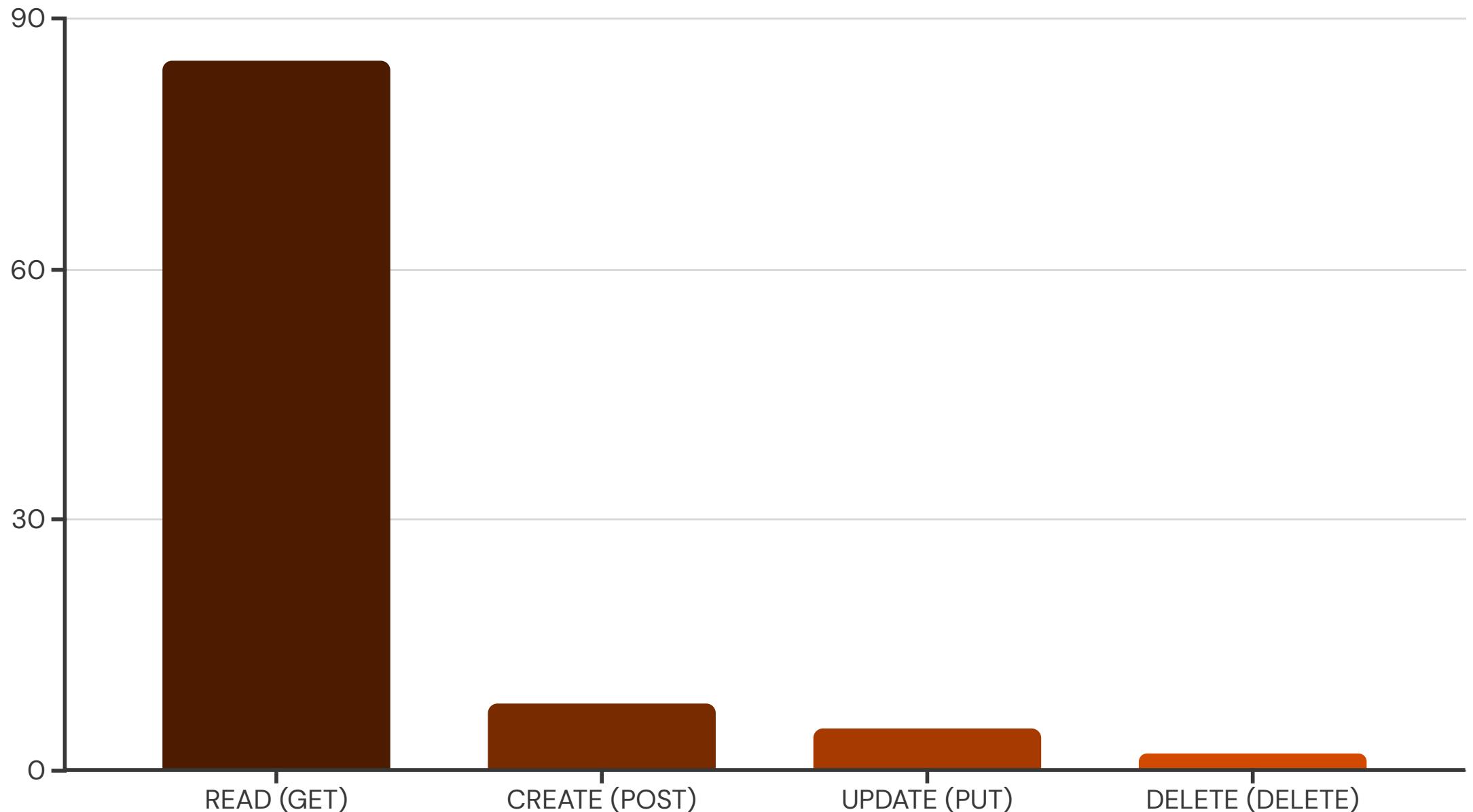
Tecnologías necesarias

- **Backend Framework:** Node.js con Express, Python con Flask/Django, PHP, etc.
- **ORM o Driver:** Librería para comunicarse con PostgreSQL (ej: pg para Node.js)
- **API REST:** Endpoints para cada operación CRUD
- **Fetch API:** En el frontend para hacer peticiones HTTP

Cambios en el frontend

```
// En lugar de manipular 'datos'  
datos.push(nuevoGift);  
  
// Haríamos petición al servidor  
fetch('/api/gifts', {  
  method: 'POST',  
  body: JSON.stringify(nuevoGift),  
  headers: {'Content-Type': 'application/json'}  
})  
.then(res => res.json())  
.then(data => cargarTabla());
```

Mapeo: CRUD a Peticiones HTTP



En una aplicación web real, la mayoría de las peticiones son lecturas (GET). Este gráfico muestra una distribución típica de peticiones en un sistema CRUD web.

Métodos HTTP para CRUD

GET → READ

Endpoint: GET /api/gifts

Respuesta: Array de todos los gifts

Frontend: Reemplaza cargarTabla() para obtener datos del servidor

PUT → UPDATE

Endpoint: PUT /api/gifts/:id

Body: Datos actualizados

Frontend: En giftUpdate() enviar cambios al servidor



POST → CREATE

Endpoint: POST /api/gifts

Body: Datos del nuevo gift

Frontend: En agregarGift() enviar datos al servidor



DELETE → DELETE

Endpoint: DELETE /api/gifts/:id

Sin body

Frontend: En BorrarGift() enviar ID al servidor

Fetch API - Comunicación con el Servidor

La **Fetch API** es la forma moderna de hacer peticiones HTTP desde JavaScript. Reemplaza a XMLHttpRequest y usa Promises para manejar respuestas asíncronas.

```
// Ejemplo: Obtener todos los gifts (READ)
fetch('http://miservidor.com/api/gifts')
  .then(response => response.json())
  .then(datos => {
    // 'datos' ahora contiene el array de gifts
    // Renderizamos la tabla
    renderizarTabla(datos);
  })
  .catch(error => {
    console.error('Error:', error);
  });
});
```

```
// Ejemplo: Crear un nuevo gift (CREATE)
fetch('http://miservidor.com/api/gifts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(nuevoGift)
})
  .then(response => response.json())
  .then(data => {
    console.log('Gift creado:', data);
    cargarTabla(); // Recargar tabla
  });
});
```

LocalStorage: Persistencia Simple en el Navegador

Si quieres agregar persistencia básica sin configurar un servidor, puedes usar **localStorage**, una API del navegador que guarda datos localmente.

Guardar datos

```
// Después de cualquier modificación
localStorage.setItem(
  'misGifts',
  JSON.stringify(datos)
);

// Los datos persisten aunque
// se cierre el navegador
```

Cargar datos

```
// Al inicio de la aplicación
const datosGuardados =
  localStorage.getItem('misGifts');

if (datosGuardados) {
  datos = JSON.parse(datosGuardados);
} else {
  // Usar datos del JSON inicial
}
```

- **Limitaciones de localStorage:** Solo funciona en el mismo navegador y dispositivo. Los datos son accesibles solo a ese usuario. Límite típico de 5-10MB. No es adecuado para aplicaciones multi-usuario.

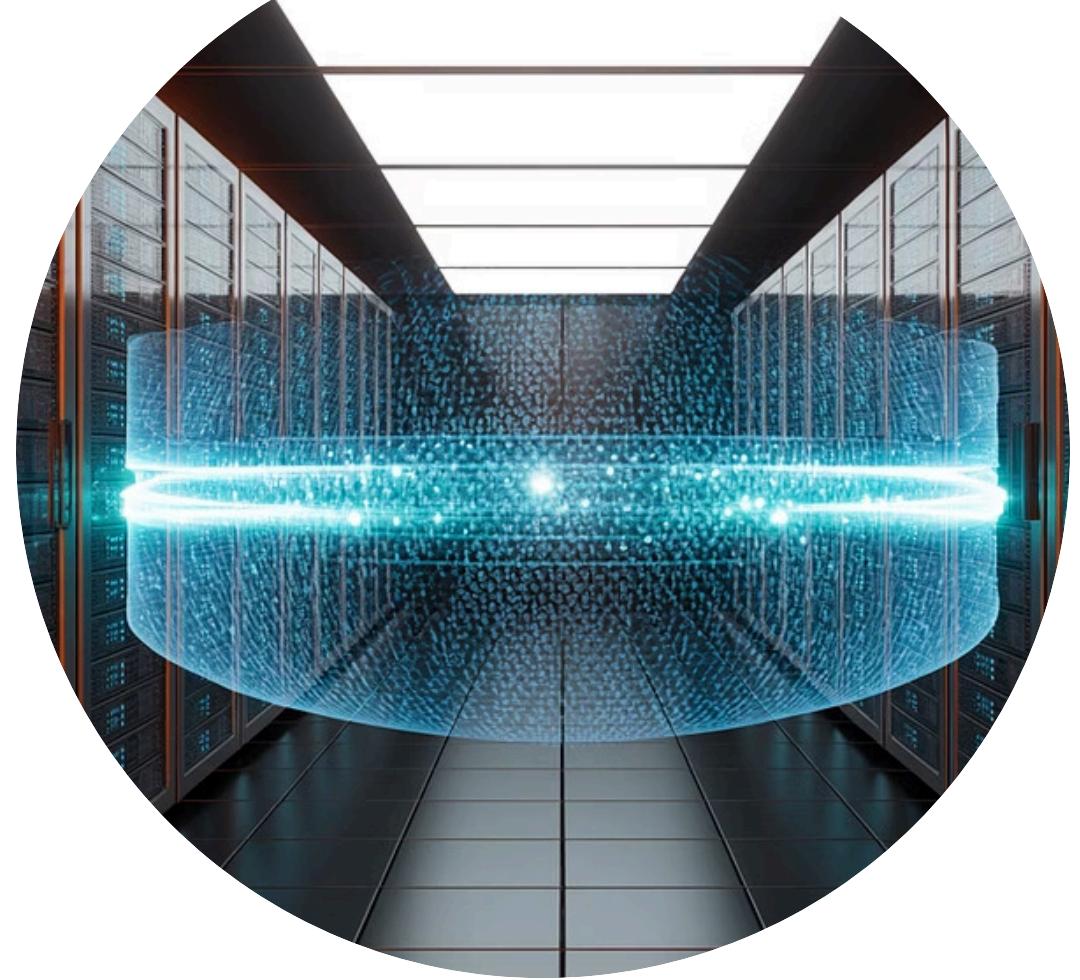
Validación de Datos

En nuestro CRUD básico no validamos los datos ingresados. En una aplicación profesional, siempre debes validar tanto en el frontend como en el backend.



Frontend (JavaScript)

Proporciona retroalimentación inmediata al usuario. Usa atributos HTML5 como required, pattern, min, max, o JavaScript para validaciones personalizadas.



Backend (Servidor)

Validación crítica de seguridad. Nunca confíes solo en validación frontend porque puede ser evadida. Valida tipos de datos, rangos y formatos.

```
// Ejemplo: Validación frontend
const agregarGift = (e) => {
  e.preventDefault();

  const precio = document.querySelector("#precio").value;

  if (precio <= 0) {
    alert('El precio debe ser mayor a 0');
    return;
  }

  // Continuar con la lógica...
};
```

Visualización de Datos: Gráficos

Para mostrar estadísticas o gráficos de los datos, necesitas una librería especializada como **Chart.js** o **D3.js**.

Chart.js (Recomendado para empezar)

- Librería simple y ligera
- Gráficos responsivos
- 8 tipos de gráficos incluidos
- Fácil de integrar

```
// Ejemplo básico
new Chart(ctx, {
    type: 'pie',
    data: {
        labels: ['Suscripción', 'Compra'],
        datasets: [
            {
                data: [12, 8]
            }
        ]
    }
});
```

D3.js (Avanzado)

- Máxima flexibilidad
- Visualizaciones complejas
- Curva de aprendizaje alta
- Control total sobre SVG

D3 manipula directamente el DOM basándose en datos, permitiendo crear visualizaciones completamente personalizadas.



Ejemplo: Estadísticas de Gifts

```
// Función para calcular estadísticas
const calcularEstadisticas = () => {
  // Contar por tipo usando reduce
  const estadisticas = datos.reduce((acc, item) => {
    acc[item.tipo] = (acc[item.tipo] || 0) + 1;
    return acc;
  }, {});
  // estadisticas = { "Suscripción": 12, "Compra": 8 }
  return estadisticas;
};

// Calcular precio promedio
const precioPromedio = datos.reduce((sum, item) => {
  return sum + parseFloat(item.precio);
}, 0) / datos.length;

// Encontrar el gift más caro
const giftMasCaro = datos.reduce((max, item) => {
  return item.precio > max.precio ? item : max;
});
```

Otros Ejemplos de CRUD

El patrón CRUD que has aprendido es universal y se puede aplicar a cualquier tipo de datos. La estructura y lógica permanecen prácticamente iguales.



Biblioteca de Libros

Clase: Libro

Propiedades: id, titulo, autor, isbn, añoPublicacion, editorial, genero

Operaciones: Mismas funciones CRUD ajustadas para libros



Gestión de Empleados

Clase: Empleado

Propiedades: id, nombre, apellido, puesto, departamento, salario, fechalngreso

Extra: Ordenar por salario, filtrar por departamento



Inventario de Productos

Clase: Producto

Propiedades: id, nombre, categoria, precio, stock, proveedor, codigo

Extra: Alertas de stock bajo, cálculo de valor total



Lista de Tareas (TODO)

Clase: Tarea

Propiedades: id, titulo, descripcion, completada, fechaLimite, prioridad

Extra: Marcar completada, filtrar por estado

Próximos Pasos en tu Aprendizaje



Dominar el CRUD básico

Practica creando diferentes aplicaciones CRUD con distintos tipos de datos hasta que el patrón sea natural para ti.



Aprender Backend

Aprende un framework backend (Node.js + Express es ideal para JavaScript developers) y conecta tu frontend con una API REST.



Trabajar con Bases de Datos

Estudia SQL y bases de datos relacionales como PostgreSQL o MySQL. Aprende a diseñar esquemas y escribir consultas.



Seguridad y Validación

Implementa autenticación, autorización y validación de datos. Aprende sobre tokens JWT y cifrado de contraseñas.



Frameworks Modernos

Explora React, Vue o Angular para manejar estados complejos y crear interfaces de usuario más sofisticadas.



Deploy y Producción

Aprende a desplegar tus aplicaciones en plataformas como Heroku, Vercel, Netlify o AWS.

Recursos Recomendados para Continuar

Documentación Oficial

- **MDN Web Docs:** Referencia completa de JavaScript, HTML y CSS
- **Bootstrap Docs:** Guía de todos los componentes y utilidades
- **Node.js Documentation:** Para backend con JavaScript
- **PostgreSQL Docs:** Aprende SQL y gestión de bases de datos

Plataformas de Aprendizaje

- **freeCodeCamp:** Cursos gratuitos de desarrollo web completo
- **The Odin Project:** Ruta de aprendizaje estructurada
- **JavaScript.info:** Tutorial moderno de JavaScript
- **Frontend Mentor:** Proyectos reales para practicar

 **Consejo del instructor:** La mejor forma de aprender es construyendo proyectos reales. Empieza con proyectos pequeños y ve aumentando la complejidad gradualmente. No tengas miedo de cometer errores, son parte esencial del aprendizaje.

¡Felicitaciones!

Has completado esta guía completa sobre la creación de un CRUD con JavaScript Vanilla. Ahora entiendes:

- ✓ La estructura y organización de un proyecto JavaScript moderno
- ✓ Programación Orientada a Objetos con clases ES6
- ✓ Manipulación del DOM y manejo de eventos
- ✓ Las cuatro operaciones fundamentales: CREATE, READ, UPDATE, DELETE
- ✓ Métodos de Array y técnicas de manipulación de datos
- ✓ Integración de Bootstrap para interfaces responsivas
- ✓ El camino hacia aplicaciones más avanzadas con backend y bases de datos

Este conocimiento es la base sólida para cualquier desarrollo web. Continúa practicando, experimenta con diferentes proyectos y nunca dejes de aprender. ¡El mundo del desarrollo web te espera!

¡Éxito en tu camino como desarrollador!

— Iván Malaver Fierro, SENA Mosquera CBA