

Unidad 9: Uso avanzado de JavaScript

# **Introducción a Promesas y Librerías**

## **Promesas en JavaScript**

### **Concepto de Promesas en JavaScript**

Una **Promesa** en JavaScript es un objeto que representa la eventual finalización (o fracaso) de una operación asíncrona y su valor resultante. Las Promesas son esenciales para manejar operaciones asíncronas como solicitudes a APIs o temporizadores, permitiendo que el código funcione de manera no bloqueante.

### **Estados de una Promesa**

Una Promesa puede encontrarse en uno de los siguientes tres estados:

1. **Pending (Pendiente)**: Es el estado inicial. La promesa está en proceso, lo que significa que aún no ha sido cumplida ni rechazada.
2. **Fulfilled (Resuelta)**: La operación se completó con éxito, y la promesa tiene un valor resultante.
3. **Rejected (Rechazada)**: La operación falló, y la promesa tiene una razón para el fallo, generalmente un error.

### **Manejo de Estados con then(), catch() y finally()**

Para trabajar con los resultados de una Promesa, utilizamos los siguientes métodos:

* **then(onFulfilled)**: Este método se ejecuta cuando la promesa se resuelve con éxito. Recibe una función que será llamada con el valor resultante de la promesa.

| promesa.then((resultado) => {  console.log("Promesa resuelta con valor:", resultado);  }); |
| --- |

* **catch(onRejected)**: Este método se ejecuta cuando la promesa es rechazada. Recibe una función que será llamada con la razón del rechazo.

| promesa.catch((error) => {  console.error("Promesa rechazada con error:", error);  }); |
| --- |

* **finally(onFinally)**: Este método se ejecuta independientemente de si la promesa fue resuelta o rechazada. Es útil para ejecutar código de limpieza o de finalización, sin importar el resultado de la promesa.

| promesa.finally(() => {  console.log("Promesa finalizada, sea cual sea el resultado.");  }); |
| --- |

### **Ejemplo de Uso de Promesas**

Imaginemos una función que simula una solicitud de datos a una API utilizando un setTimeout para simular la demora en la respuesta:

| const obtenerDatos = () => {  return new Promise((resolve, reject) => {  setTimeout(() => {  const exito = true; // Cambia esto para probar diferentes resultados  if (exito) {  resolve("Datos obtenidos correctamente");  } else {  reject("Error al obtener los datos");  }  }, 2000);  });  };  obtenerDatos()  .then((resultado) => {  console.log(resultado);  })  .catch((error) => {  console.error(error);  })  .finally(() => {  console.log("Operación completada.");  }); |
| --- |

En este ejemplo, obtenerDatos() devuelve una promesa que se resuelve o rechaza después de 2 segundos. Dependiendo del valor de exito, la promesa será resuelta o rechazada. El then() maneja el resultado exitoso, catch() maneja cualquier error, y finally() se ejecuta al final de cualquier manera.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Trabajando con Librerías**

## **Introducción a Librerías**

### **¿Qué son las librerías en JavaScript?**

Las **librerías en JavaScript** son colecciones de código preescrito que desarrolladores pueden integrar en sus proyectos para resolver problemas comunes de manera más rápida y eficiente. Estas librerías actúan como "cajas de herramientas", proporcionando funciones y métodos que simplifican tareas que, de otro modo, requerirían escribir y depurar mucho código desde cero.

### **¿Cómo funcionan las librerías?**

Una librería se puede imaginar como un conjunto de herramientas especializadas. Al incorporarlas en un proyecto, se pueden utilizar estas herramientas para realizar tareas específicas, lo que no solo ahorra tiempo, sino que también mejora la calidad y la eficiencia del desarrollo. Por ejemplo, en lugar de programar desde cero la validación de formularios, una librería especializada puede ofrecer funciones ya probadas y optimizadas para validar entradas de usuarios.

### **Aplicaciones comunes de las librerías**

Algunas de las áreas donde las librerías de JavaScript se utilizan con frecuencia incluyen:

* **Validación de datos**: Asegurar que los datos ingresados por el usuario cumplan con ciertos criterios antes de ser procesados.
* **Interfaces visuales**: Mejora de la presentación y la experiencia del usuario mediante elementos visuales interactivos, como sliders, modales, y alertas personalizadas.
* **Manejo de fechas**: Facilitar la manipulación de fechas y horas, como formateo, comparación y cálculo de intervalos.
* **Peticiones AJAX**: Simplificar la comunicación con servidores para enviar y recibir datos de manera asíncrona sin recargar la página completa.

### **Ejemplos de librerías populares**

* **jQuery**: Facilita la manipulación del DOM y maneja eventos, animaciones, y peticiones AJAX.
* **Sweet Alert**: Permite la creación de alertas visualmente atractivas y personalizables que mejoran la experiencia del usuario.
* **Luxon**: Ofrece herramientas para manejar fechas y horas de forma sencilla y potente.

### **Implementación**

Para usar una librería, los desarrolladores pueden descargarla e incluirla en sus proyectos o vincularla a través de un CDN (Content Delivery Network). Una vez integrada, se pueden utilizar las funciones y métodos que la librería ofrece, lo que reduce la complejidad del código y mejora la productividad.

## **Implementando Librerías**

### **Uso de CDN para Vincular Librerías**

Un CDN es una red de distribución de contenido que permite acceder a archivos hospedados en servidores distribuidos globalmente. Para vincular una librería a través de un CDN, simplemente se debe incluir un <script> en el archivo HTML que apunte a la URL del CDN.

**Ejemplo con SweetAlert**:

| <script src="<https://cdn.jsdelivr.net/npm/sweetalert2@11>"></script> |
| --- |

Este código incluye la versión 11 de SweetAlert directamente en el proyecto. De esta manera, no es necesario descargar ni alojar los archivos de la librería en tu servidor, lo que reduce el tiempo de carga y facilita las actualizaciones.

### **Uso de Archivos Minificados**

Los archivos minificados son versiones optimizadas de los archivos JavaScript, donde se han eliminado espacios, comentarios y saltos de línea para reducir el tamaño del archivo, mejorando así el tiempo de carga.

**Ejemplo**:

| <script src="<https://cdn.jsdelivr.net/npm/sweetalert2@11/sweetalert2.min.js>"></script> |
| --- |

Aquí se está utilizando la versión minificada de SweetAlert, lo que asegura un rendimiento óptimo al incluir la librería en tu proyecto.

### **Vinculación de Librerías con Fetch**

En algunos casos, es posible que quieras utilizar Fetch para descargar dinámicamente una librería y ejecutarla en tu proyecto. Esto puede ser útil si necesitas cargar librerías bajo ciertas condiciones o en respuesta a eventos específicos.

**Ejemplo básico con Fetch**:

| fetch('<https://cdn.jsdelivr.net/npm/sweetalert2@11/sweetalert2.min.js>')  .then(response => response.text())  .then(script => {  const scriptTag = document.createElement('script');  scriptTag.textContent = script;  document.body.appendChild(scriptTag);  Swal.fire('Librería cargada con éxito!');  })  .catch(error => console.error('Error cargando la librería:', error)); |
| --- |

Este código utiliza Fetch para descargar la librería SweetAlert desde un CDN y luego inyectarla en la página como un script. Una vez cargada, la librería puede ser utilizada inmediatamente.

**Conclusión**

Vincular librerías externas a un proyecto JavaScript es una tarea esencial que se puede realizar de diferentes maneras, ya sea a través de un CDN o usando archivos minificados. Además, Fetch ofrece una manera flexible de cargar librerías bajo demanda, lo que puede ser muy útil en aplicaciones dinámicas y basadas en eventos. Estos métodos optimizan tanto el rendimiento como la capacidad de actualización de tu proyecto.

# **Alertas y Notificaciones**

## **Configuración de Toastify**

### **Configuración y Personalización de Notificaciones con Toastify**

**Toastify** es una librería ligera de JavaScript que permite crear notificaciones o "toasts" personalizadas y visualmente atractivas en una aplicación web. Estas notificaciones son ideales para mostrar mensajes temporales que no requieren interacción del usuario. A continuación, te explico cómo configurar y personalizar Toastify, enfocándonos en propiedades clave como gravity, position, y style.

### **1. Instalación de Toastify**

Antes de empezar a configurar Toastify, es necesario integrarlo en tu proyecto. Puedes hacerlo de dos maneras: descargando la librería o vinculándola directamente desde un CDN.

**Vinculación desde un CDN**:

| <script src="<https://cdn.jsdelivr.net/npm/toastify-js>"></script> |
| --- |

Una vez vinculado, ya puedes empezar a utilizar Toastify en tu proyecto.

### **2. Creación de una Notificación Básica**

Para crear una notificación básica, se utiliza el método Toastify({ options }).showToast();. Las options son un objeto que define las propiedades de la notificación.

**Ejemplo básico**:

| Toastify({  text: "¡Esta es una notificación básica!",  }).showToast(); |
| --- |

### **3. Propiedad gravity**

La propiedad gravity define la verticalidad de la notificación, es decir, si aparecerá en la parte superior o inferior de la pantalla.

* **gravity: "top"**: Coloca la notificación en la parte superior (valor por defecto).
* **gravity: "bottom"**: Coloca la notificación en la parte inferior.

**Ejemplo**:

| Toastify({  text: "Notificación en la parte inferior",  gravity: "bottom", // Aparece en la parte inferior  }).showToast(); |
| --- |

### **4. Propiedad position**

La propiedad position se utiliza para definir la horizontalidad de la notificación dentro de la pantalla, especificando si aparecerá a la izquierda, centro o derecha.

* **position: "left"**: Alinea la notificación a la izquierda.
* **position: "center"**: Centra la notificación horizontalmente.
* **position: "right"**: Alinea la notificación a la derecha (valor por defecto).

**Ejemplo**:

| Toastify({  text: "Notificación centrada",  gravity: "top", // Aparece en la parte superior  position: "center", // Centra la notificación horizontalmente  }).showToast(); |
| --- |

### **5. Propiedad style**

La propiedad style permite aplicar estilos CSS personalizados a la notificación, como colores de fondo, fuente, bordes, etc.

**Ejemplo**:

| Toastify({  text: "Notificación personalizada",  gravity: "top",  position: "right",  style: {  background: "linear-gradient(to right, #00b09b, #96c93d)",  color: "white",  fontWeight: "bold",  }  }).showToast(); |
| --- |

En este ejemplo, se aplica un degradado como fondo, se establece el color del texto en blanco y se hace el texto en negrita.

**Conclusión**

Toastify es una herramienta poderosa y flexible para añadir notificaciones a tus aplicaciones web. Mediante el uso de propiedades como gravity, position, y style, puedes personalizar completamente la apariencia y ubicación de las notificaciones, asegurando que se adapten a las necesidades específicas de tu proyecto. Estas configuraciones permiten mejorar significativamente la experiencia del usuario, manteniéndolo informado de manera sutil y no intrusiva.