Formularios DOM

1. Introducción

Un **formulario** en HTML es una herramienta utilizada para recopilar datos del usuario a través de diversos campos de entrada. Estos datos pueden enviarse a un servidor para su procesamiento o manipularse directamente en el navegador mediante JavaScript.

Los formularios son fundamentales para cualquier aplicación web interactiva, como registros, inicios de sesión, encuestas o búsquedas.

Vamos a trabajar el acceso a los formularios de una página web desde la perspectiva del lado del cliente. Las funcionalidades básicas que podemos implementar son: 1. Validación de datos. Tenemos que recordar que la validación en cliente es meramente por usabilidad de la página, no por seguridad. La validación de datos debe hacerse también el servidor. 2. Crear dinamismo. En función de los valores recogidos podemos modificar el estado de la página.

2. Estructura básica de un formulario

Un formulario se define con la etiqueta <form> y contiene diversos **elementos de entrada** que permiten al usuario proporcionar información.

Ejemplo:

```
<form action="/submit-data" method="post">
  <label for="name">Nombre:</label>
  <input type="text" id="name" name="user_name" />

  <label for="email">Correo:</label>
  <input type="email" id="email" name="user_email" />

  <button type="submit">Enviar</button>
</form>
```

3. Atributos clave del elemento <form>

3.1. action: Define la URL o el destino donde se enviarán los datos.

- Ejemplo: action="/submit-form"

3.2. method: Especifica el método HTTP para enviar los datos.

- Valores comunes:
  - GET: Los datos se envían como parte de la URL (menos seguro, adecuado para búsquedas).
  - POST: Los datos se envían en el cuerpo de la solicitud (más seguro y usado para formularios más complejos).

3.3. enctype: Especifica cómo se codifican los datos.

- **application/x-www-form-urlencoded** (por defecto): Datos codificados como pares clave-valor.
- **multipart/form-data**: Usado para subir archivos.
- **text/plain**: Envía datos como texto sin formato.

4. Elementos comunes en un formulario

1. <input>: Campo genérico para la entrada de datos.
   - Atributo type define el tipo de entrada:
     - **text**: Entrada de texto.
     - **password**: Entrada de contraseñas (oculta caracteres).
     - **email**: Entrada para correos electrónicos.
     - **number**: Entrada numérica.

- **file**: Subida de archivos.
- **checkbox**: Casillas de verificación.
- **radio**: Botones de opción.
- **submit**: Botón para enviar el formulario.

Ejemplo:

```
<input type="text" name="username" placeholder="Escribe tu nombre" />
```

2. <textarea>: Campo para textos largos.

```
<textarea name="message" rows="4" cols="50"></textarea>
```

3. <select> **y** <option>: Menú desplegable.

```
<select name="options">
  <option value="opcion1">Opción 1</option>
  <option value="opcion2">Opción 2</option>
</select>
```

4. <button>: Botón interactivo.

```
<button type="submit">Enviar</button>
```

5. <label>: Etiqueta para asociar un texto a un campo.

```
<label for="username">Usuario:</label>
<input type="text" id="username" name="username" />
```

5. Interacción con formularios

Los formularios en HTML pueden combinarse con **JavaScript** para validación y manejo de eventos. Esto permite una interacción más dinámica sin necesidad de recargar la página.

Ejemplo básico de validación:

```
<form id="myForm">
  <input type="text" id="name" placeholder="Nombre" />
  <button type="submit">Enviar</button>
</form>

<script>
  const form = document.getElementById('myForm');
  form.addEventListener('submit', function (event) {
    const nameInput = document.getElementById('name').value;
    if (!nameInput) {
      alert('Por favor, ingresa tu nombre');
```

```
    event.preventDefault(); // Evita el envío del formulario
  }
 });
</script>
```

Conclusión

Los formularios son esenciales para interactuar con los usuarios en la web. La combinación de etiquetas HTML, atributos, y funcionalidades de JavaScript permiten crear formularios robustos, fáciles de usar y seguros para aplicaciones modernas.

6. La colección forms

La colección forms en JavaScript es una propiedad del objeto document que contiene una lista de todos los formularios (<form>) presentes en un documento HTML. Cada formulario de la página se almacena como un elemento dentro de esta colección, y puede ser accedido y manipulado desde JavaScript.

6.1. Características principales de document.forms

Tipo de objeto:

- Es una colección *HTMLCollection*, lo que significa que es una lista ordenada y dinámica de objetos que representa los formularios del documento.
- Los formularios se enumeran en el orden en que aparecen en el documento.

Acceso a los formularios:

- Puedes acceder a los formularios mediante índices numéricos (document.forms[0]) o por su atributo name o id (document.forms['formName']).

Propiedades de la colección:

- Es dinámica: Si se agrega o elimina un formulario en el DOM, la colección se actualiza automáticamente.

6.2. Acceso y manipulación de formularios con forms

Ejemplo básico:

Dado un documento HTML con varios formularios:

```
<!DOCTYPE html>
<html>
<body>
 <form id="loginForm" name="loginForm">
  <input type="text" name="username" placeholder="Usuario" />
  <input type="password" name="password" placeholder="Contraseña" />
  <button type="submit">Iniciar Sesión</button>
 </form>
```

```
  <form id="signupForm" name="signupForm">
    <input type="text" name="email" placeholder="Correo" />
    <input type="password" name="password" placeholder="Contraseña" />
    <button type="submit">Registrarse</button>
  </form>
</body>
</html>
```

Acceso a los formularios:

```
// Acceder a todos los formularios
console.log(document.forms); // HTMLCollection de los formularios

// Acceso por índice
const loginForm = document.forms[0];
console.log(loginForm.id); // "loginForm"

// Acceso por el atributo name
const signupForm = document.forms['signupForm'];
console.log(signupForm.id); // "signupForm"

// Acceso por id (equivalente al acceso directo con getElementById)
const sameForm = document.forms.signupForm;
console.log(sameForm === signupForm); // true
```

6.3. Uso común de forms

Acceder a los elementos de un formulario: **Cada formulario tiene una colección elements, que contiene todos los campos de entrada (<input>, <textarea>, etc.).**

```
const loginForm = document.forms['loginForm'];
console.log(loginForm.elements); // HTMLCollection de los campos del formulario

// Acceder a un campo por su atributo name
const usernameInput = loginForm.elements['username'];
console.log(usernameInput.placeholder); // "Usuario"

// Acceder por índice
const passwordInput = loginForm.elements[1];
console.log(passwordInput.type); // "password"
```

Modificar valores de los campos:

```
// Cambiar el valor de un campo
const signupForm = document.forms['signupForm'];
signupForm.elements['email'].value = 'test@example.com';
```

Enviar un formulario mediante JavaScript: **Puedes usar el método submit() para enviar el formulario programáticamente.**

```
const loginForm = document.forms['loginForm'];
loginForm.submit(); // Envía el formulario sin necesidad de clic en el botón
```

Evitar el envío del formulario: **Al capturar el evento submit, puedes evitar que el formulario se envíe.**

```
const loginForm = document.forms['loginForm'];
loginForm.addEventListener('submit', function (event) {
  event.preventDefault(); // Evita el envío
  console.log('Formulario validado, pero no enviado.');
});
```

6.4. Ventajas de forms

- Permite un acceso directo y ordenado a todos los formularios de la página.
- Es dinámico y siempre refleja los formularios actuales en el documento.
- Simplifica la manipulación y validación de formularios desde JavaScript.

6.5. Conclusión

La colección forms es una herramienta útil y sencilla para gestionar los formularios en una página web. A través de ella, puedes acceder a formularios, sus elementos y realizar operaciones como validaciones, cambios de valores, o envíos programáticos de forma eficiente.

7. La colección elements

La colección elements en JavaScript es una propiedad de un objeto <form> que contiene una lista de todos los elementos de formulario dentro de ese <form>. Esta colección incluye elementos como <input>, <textarea>, <select>, <button>, y otros, permitiendo acceder y manipular sus propiedades y valores.

7.1. Características principales de form.elements

Tipo de objeto:

- Es una colección *HTMLCollection*, lo que significa que es una lista ordenada de elementos de formulario.
- Los elementos se enumeran en el orden en que aparecen en el documento.

Acceso a los elementos:

- Se puede acceder a los elementos por índice (form.elements[0]) o por sus atributos name o id (form.elements['elementName']).

Elementos incluidos:

- Contiene todos los elementos de entrada que tienen los siguientes tipos: input, textarea, select, button, y fieldset.

Ejemplo básico

Dado este formulario en HTML:

```
<form id="userForm">
  <input type="text" name="username" placeholder="Nombre de usuario" />
  <input type="password" name="password" placeholder="Contraseña" />
  <button type="submit">Iniciar Sesión</button>
</form>
```

Acceder a la colección elements:

```
const form = document.getElementById('userForm');

// Acceso a todos los elementos
console.log(form.elements); // HTMLCollection de los elementos del formulario

// Acceso por índice
console.log(form.elements[0]); // Primer input (campo de nombre de usuario)
console.log(form.elements[1]); // Segundo input (campo de contraseña)

// Acceso por atributo name
console.log(form.elements['username']); // Campo de nombre de usuario
console.log(form.elements['password']); // Campo de contraseña
```

7.2. Manipulación de elementos usando form.elements

Cambiar valores de los campos:

```
const usernameField = form.elements['username'];
usernameField.value = 'JuanPerez'; // Establecer el valor del campo de texto

const passwordField = form.elements['password'];
passwordField.value = '123456'; // Establecer la contraseña
```

Leer valores de los campos:

```
console.log(usernameField.value); // "JuanPerez"
console.log(passwordField.value); // "123456"
```

7.3. Validación de formularios con form.elements

Puedes usar form.elements para validar los datos antes de enviar el formulario:

Validación básica:

```
form.addEventListener('submit', function (event) {
  const username = form.elements['username'].value;
  const password = form.elements['password'].value;
```

```
  if (!username || !password) {
    alert('Por favor, completa todos los campos.');
    event.preventDefault(); // Evita el envío del formulario
  }
});
```

7.4. Ejemplo práctico: Contar campos en un formulario

Si deseas contar cuántos campos tiene un formulario, puedes usar form.elements.length:

```
console.log(`El formulario tiene ${form.elements.length} elementos.`); // "El formulario tiene 3 elementos."
```

7.5. Interacción avanzada: Deshabilitar todos los campos

Puedes iterar sobre form.elements para realizar acciones en todos los elementos del formulario:

```
for (let element of form.elements) {
  element.disabled = true; // Deshabilita todos los campos del formulario
}
```

7.6. Características adicionales

Elementos sin el atributo **name**:

- Si un elemento dentro de un formulario no tiene un atributo name, no puede ser accedido por form.elements['name'], pero sí por índice.

```
<form id="exampleForm">
  <input type="text" name="email" placeholder="Correo" />
  <input type="text" placeholder="Campo sin nombre" />
</form>

const exampleForm = document.getElementById('exampleForm');
console.log(exampleForm.elements['email']); // Campo de correo
console.log(exampleForm.elements[1]); // Campo sin nombre
```

Dinamismo:

- La colección elements se actualiza automáticamente si los elementos del formulario cambian en tiempo de ejecución.

```
const newInput = document.createElement('input');
newInput.name = 'newField';
form.appendChild(newInput);

console.log(form.elements['newField']); // Ahora incluye el nuevo campo
```

7.7. Conclusión

La colección form.elements es una herramienta poderosa para acceder y manipular los campos de un formulario de manera organizada. Su integración directa con el objeto <form> hace que trabajar con formularios sea más eficiente, ya sea para validar datos, cambiar valores o realizar otras tareas relacionadas con la interacción del usuario.

8. La propiedad form

La propiedad form en JavaScript es una referencia que existe en ciertos elementos de un formulario HTML (como <input>, <textarea>, <button>, etc.) y que apunta al objeto <form> al que pertenecen. Es decir, permite acceder al formulario padre desde cualquiera de sus elementos.

8.1. Características principales de la propiedad form

Disponibilidad:

- La propiedad form está disponible solo en elementos que pueden estar dentro de un formulario, como:
  - <input>
  - <textarea>
  - <select>
  - <button>

Tipo de valor:

- Devuelve el objeto <form> al que pertenece el elemento.
- Si el elemento no está dentro de un formulario, la propiedad devuelve null.

Útil para:

- Acceder al formulario desde cualquier elemento del mismo, incluso sin usar document.getElementById() o document.forms.

8.2. Ejemplo básico

Dado este HTML:

```
<form id="userForm">
  <input type="text" id="username" name="username" placeholder="Usuario" />
  <input type="password" id="password" name="password" placeholder="Contraseña" />
  <button type="submit">Enviar</button>
</form>
```

8.3. Acceder al formulario desde un elemento:

```
const usernameField = document.getElementById('username');

// Usar la propiedad form
const parentForm = usernameField.form;

console.log(parentForm.id); // "userForm"
```

8.4. Uso común de form

Validar el formulario desde un elemento:

Puedes usar la propiedad form para validar todo el formulario desde un elemento específico.

```
const passwordField = document.getElementById('password');

passwordField.addEventListener('blur', function () {
  const form = this.form; // Accede al formulario desde el campo
  const username = form.elements['username'].value;

  if (!username) {
    alert('El campo "Usuario" es obligatorio.');
  }
});
```

Interacción dinámica con el formulario

Si necesitas interactuar con el formulario directamente desde un elemento (sin necesidad de buscar el formulario por su ID):

```
const submitButton = document.querySelector('button[type="submit"]');

submitButton.addEventListener('click', function (event) {
  const form = this.form; // Accede al formulario desde el botón
  console.log(`Enviando el formulario con ID: ${form.id}`);
});
```

8.5. Ejemplo práctico: Resetear el formulario

Usa la propiedad form para reiniciar el formulario desde cualquier campo o botón que esté dentro de él:

```
const usernameField = document.getElementById('username');

// Escucha un evento en el campo y reinicia el formulario
usernameField.addEventListener('dblclick', function () {
  this.form.reset(); // Reinicia el formulario desde el campo
});
```

8.6. Uso con elementos fuera del formulario

En HTML, puedes usar el atributo form en elementos como <input> o <button> para asociarlos a un formulario específico, incluso si no están dentro del formulario en el DOM.

```
<form id="externalForm">
```

```
  <input type="text" name="email" placeholder="Correo" />
</form>

<!-- Botón fuera del formulario pero asociado a él -->
<button type="submit" form="externalForm">Enviar</button>
```

8.7. Acceso desde JavaScript:

```
const externalButton = document.querySelector('button[form="externalForm"]');

// Accede al formulario asociado
console.log(externalButton.form.id); // "externalForm"
```

8.8. Características adicionales

Propiedad **form** en elementos sin formulario:

- Si un elemento no está dentro de un formulario y no tiene el atributo form, la propiedad devuelve null.

```
const standaloneInput = document.createElement('input');
console.log(standaloneInput.form); // null
```

Propiedad dinámica:

- Si un elemento se mueve entre formularios en tiempo de ejecución, la propiedad form se actualiza automáticamente.

```
const usernameField = document.getElementById('username');
const newForm = document.createElement('form');
newForm.id = 'newForm';

document.body.appendChild(newForm);
newForm.appendChild(usernameField);

console.log(usernameField.form.id); // "newForm"
```

8.9. Conclusión

La propiedad form es una forma eficiente de acceder al formulario padre de un elemento, especialmente útil cuando no se desea buscar el formulario explícitamente en el DOM. Ofrece una forma más directa y legible de trabajar con formularios en aplicaciones web dinámicas.

9. Validación de un formulario

Sin entrar en el uso de frameworks, podemos realizar la validación de formularios mediante dos métodos:

- El navegador, a través de la especificación de HTML5, es capaz de validar los elementos de un formulario. Se trata de un método declarativo a través de propiedades de los elementos, con la ventaja de que se aplica muy rápidamente, sin necesidad aplicar código en primera instancia.

- Programando mediante funciones JavaScript la validación. Este método tiene la ventaja de que la validación es personalizada. Vamos a ver varios ejemplos de cómo podemos validar un formulario. Como diseño web de los formularios, seguimos contando con la librería Bootstrap que nos facilita la maquetación. Utilizamos las siguientes clases:
  - is-valid: Aplicado al elemento, indica un estilo de valor válido.
  - is-invalid: Aplicado al elemento, indica un estilo de valor inválido.
  - valid-feedback: Lo utilizamos para el estilo de la capa contenedora de la retroalimentación del valor válido. A priori se encuentra oculta.
  - invalid-feedback: Es el estilo de la retroalimentación para un valor no válido, también se encuentra oculta.

9.1. Validación del navegador

La especificación de HTML5 define varios tipos de restricciones que podemos declarar en un formulario.

9.1.1. La propiedad required

La propiedad required es un atributo que se utiliza en elementos de formulario HTML para garantizar que un campo sea obligatorio antes de enviar el formulario. Cuando un elemento tiene este atributo, el navegador valida automáticamente que el usuario haya proporcionado un valor en ese campo antes de permitir el envío del formulario.

¿Cómo funciona el atributo required?

Elementos aplicables:

- Se puede usar en elementos de entrada como <input>, <textarea>, y <select>.
- No funciona en elementos que no aceptan datos, como <button>.

Comportamiento:

- Si el campo está vacío al intentar enviar el formulario, el navegador muestra un mensaje de error predefinido indicando que se requiere completar el campo.
- El formulario no se enviará hasta que el campo tenga un valor válido.

9.1.2. Ejemplo básico:

```
<form>
  <label for="name">Nombre:</label>
  <input type="text" id="name" name="name" required>
  <button type="submit">Enviar</button>
</form>
```

En este ejemplo:

- Si el usuario intenta enviar el formulario sin llenar el campo "Nombre", el navegador mostrará un mensaje como: "Por favor, llena este campo".

9.1.3. Compatibilidad con otros tipos de validación

El atributo required puede combinarse con otras validaciones, como:

- **Tipos de datos** (type): Por ejemplo, en un <input type="email" required>, el navegador verificará que el campo tenga un valor y que sea un formato de correo electrónico válido.
- **Rangos** (min, max, minlength, maxlength): El campo debe cumplir con estas restricciones además de no estar vacío.

9.1.4. Estilización con CSS

Puedes usar el selector :required para aplicar estilos a los campos obligatorios. Por ejemplo:

```
input:required {
  border: 2px solid red;
}
```

9.1.5. Limitaciones

1. **Solo para navegadores modernos**: Aunque la mayoría de los navegadores modernos soportan required, es posible que navegadores antiguos no lo respeten.
2. **Validación del lado del cliente**: La validación con required es una medida inicial, pero no reemplaza la validación en el servidor, que sigue siendo necesaria para la seguridad.

Los navegadores muestran mensajes de error predeterminados para el atributo required, pero puedes personalizarlos utilizando el atributo title, la API de validación del navegador o scripts de JavaScript.

1. Personalización básica con el atributo **title**

Puedes proporcionar un mensaje personalizado usando el atributo title. Cuando el usuario no llena el campo, algunos navegadores mostrarán este mensaje.

```
<form>
  <label for="email">Correo Electrónico:</label>
  <input type="email" id="email" name="email" required title="Por favor, introduce un correo válido">
  <button type="submit">Enviar</button>
</form>
```

Si el usuario no completa el campo, el navegador mostrará: *"Por favor, introduce un correo válido"*.

2. Personalización avanzada con la API de validación

Puedes usar la API de validación de formularios en JavaScript para personalizar completamente los mensajes de error.

Propiedades útiles de la API:

- checkValidity(): Comprueba si el formulario o un elemento específico es válido.

- setCustomValidity(message): Establece un mensaje de error personalizado. Si el mensaje no está vacío, el formulario será inválido.
- validationMessage: Devuelve el mensaje de error actual para un campo.

Ejemplo:

```
<form id="myForm">
  <label for="username">Nombre de Usuario:</label>
  <input type="text" id="username" name="username" required>
  <span id="error-message" style="color: red;"></span>
  <br>
  <button type="submit">Enviar</button>
</form>

<script>
  const form = document.getElementById("myForm");
  const usernameInput = document.getElementById("username");
  const errorMessage = document.getElementById("error-message");

  usernameInput.addEventListener("input", () => {
    // Limpia el mensaje personalizado si el campo es válido
    if (usernameInput.checkValidity()) {
      errorMessage.textContent = "";
      usernameInput.setCustomValidity("");
    } else {
      usernameInput.setCustomValidity("Este campo es obligatorio.");
    }
  });

  form.addEventListener("submit", (event) => {
    // Verifica si el formulario es válido
    if (!usernameInput.checkValidity()) {
      errorMessage.textContent = usernameInput.validationMessage;
      event.preventDefault(); // Evita el envío del formulario
    }
  });
</script>
```

En este ejemplo:

1. Si el campo "Nombre de Usuario" está vacío, muestra el mensaje *"Este campo es obligatorio"* en el span rojo.
2. El formulario no se enviará hasta que el usuario complete correctamente el campo.

3. Validación completamente personalizada

Puedes crear mensajes personalizados para diferentes errores usando la propiedad validity, que incluye:

- valueMissing: Cuando el campo está vacío.

- typeMismatch: Cuando el tipo de dato no coincide (por ejemplo, un correo no válido).
- tooShort o tooLong: Cuando no se cumplen las restricciones de longitud.
- patternMismatch: Cuando no coincide con un patrón especificado.

Ejemplo con múltiples validaciones:

```
<form id="emailForm">
  <label for="email">Correo Electrónico:</label>
  <input type="email" id="email" name="email" required minlength="5" maxlength="50"
pattern="[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$">
  <span id="email-error" style="color: red;"></span>
  <br>
  <button type="submit">Enviar</button>
</form>

<script>
  const emailInput = document.getElementById("email");
  const emailError = document.getElementById("email-error");

  emailInput.addEventListener("input", () => {
    let message = "";

    if (emailInput.validity.valueMissing) {
      message = "El campo de correo es obligatorio.";
    } else if (emailInput.validity.typeMismatch) {
      message = "Por favor, introduce un correo válido.";
    } else if (emailInput.validity.tooShort) {
      message = `El correo debe tener al menos ${emailInput.minLength} caracteres.`;
    } else if (emailInput.validity.tooLong) {
      message = `El correo no puede exceder los ${emailInput.maxLength} caracteres.`;
    } else if (emailInput.validity.patternMismatch) {
      message = "El correo no cumple con el formato requerido.";
    }

    emailError.textContent = message;
    emailInput.setCustomValidity(message); // Establece el mensaje de error en la API
  });
</script>
```

Resultado esperado:

1. Si el campo de correo está vacío: *"El campo de correo es obligatorio."*
2. Si no tiene un formato válido: *"Por favor, introduce un correo válido."*
3. Si es demasiado corto o largo: mensajes claros basados en minlength y maxlength.

9.2. Restricciones basadas en los tipos de datos de los campos

La API de JavaScript para formularios ofrece un conjunto de métodos y propiedades que permiten controlar y validar restricciones basadas en los **tipos de datos** de los campos.

Esto incluye elementos como <input>, <textarea> y otros controles. Aquí te explico las restricciones más comunes y cómo gestionarlas con JavaScript.

9.2.1. Restricciones básicas por tipos (type)

El atributo type en un <input> define el tipo de datos que se espera en un campo. Desde JavaScript, podemos validar estas restricciones y asegurarnos de que los valores ingresados sean correctos.

Ejemplo:

```
<form id="exampleForm">
  <label for="email">Correo Electrónico:</label>
  <input type="email" id="email" name="email" required>
  <span id="email-error" style="color: red;"></span>
  <br>
  <button type="submit">Enviar</button>
</form>

<script>
  const emailInput = document.getElementById("email");
  const emailError = document.getElementById("email-error");

  emailInput.addEventListener("input", () => {
    if (!emailInput.validity.valid) {
      if (emailInput.validity.typeMismatch) {
        emailError.textContent = "Por favor, introduce un correo válido.";
      } else if (emailInput.validity.valueMissing) {
        emailError.textContent = "Este campo es obligatorio.";
      }
    } else {
      emailError.textContent = ""; // Sin errores
    }
  });
</script>
```

Explicación:

- Si el valor no tiene formato de correo válido, el navegador detecta el error usando validity.typeMismatch.
- Si el campo está vacío y es obligatorio, validity.valueMissing se activa.

9.2.2. Restricciones de longitud (minlength y maxlength)

Controlan la longitud mínima y máxima de los valores ingresados. Desde JavaScript, puedes validar estos límites usando las propiedades minLength y maxLength.

Ejemplo:

```
<form id="usernameForm">
  <label for="username">Nombre de Usuario (entre 5 y 15 caracteres):</label>
```

```
  <input type="text" id="username" name="username" minlength="5" maxlength="15"
required>
  <span id="username-error" style="color: red;"></span>
  <br>
  <button type="submit">Enviar</button>
</form>

<script>
  const usernameInput = document.getElementById("username");
  const usernameError = document.getElementById("username-error");

  usernameInput.addEventListener("input", () => {
    if (usernameInput.validity.tooShort) {
      usernameError.textContent = `El nombre debe tener al menos
${usernameInput.minLength} caracteres.`;
    } else if (usernameInput.validity.tooLong) {
      usernameError.textContent = `El nombre no puede tener más de
${usernameInput.maxLength} caracteres.`;
    } else if (usernameInput.validity.valueMissing) {
      usernameError.textContent = "Este campo es obligatorio.";
    } else {
      usernameError.textContent = ""; // Sin errores
    }
  });
</script>
```

9.2.3. Restricciones de valores numéricos (min, max, step)

Se utilizan en campos con type="number", type="range", o type="date". Desde JavaScript, puedes validar estas restricciones y ajustar los valores.

Ejemplo:

```
<form id="ageForm">
  <label for="age">Edad (entre 18 y 60):</label>
  <input type="number" id="age" name="age" min="18" max="60" step="1" required>
  <span id="age-error" style="color: red;"></span>
  <br>
  <button type="submit">Enviar</button>
</form>

<script>
  const ageInput = document.getElementById("age");
  const ageError = document.getElementById("age-error");

  ageInput.addEventListener("input", () => {
    if (ageInput.validity.rangeUnderflow) {
      ageError.textContent = `La edad debe ser al menos ${ageInput.min}.`;
    } else if (ageInput.validity.rangeOverflow) {
      ageError.textContent = `La edad no puede ser mayor a ${ageInput.max}.`;
    } else if (ageInput.validity.stepMismatch) {
```

```
    ageError.textContent = "Por favor, introduce un valor válido.";
  } else if (ageInput.validity.valueMissing) {
    ageError.textContent = "Este campo es obligatorio.";
  } else {
    ageError.textContent = ""; // Sin errores
  }
 });
</script>
```

9.2.4. Restricciones de patrones (pattern)

El atributo pattern usa expresiones regulares para validar valores. Desde JavaScript, puedes verificar si el valor coincide con el patrón.

Ejemplo:

```
<form id="passwordForm">
 <label for="password">Contraseña (al menos 8 caracteres, una letra y un
número):</label>
 <input type="password" id="password" name="password" pattern="(?=.*[A-Za-
z])(?=.*\d)[A-Za-z\d]{8,}" required>
 <span id="password-error" style="color: red;"></span>
 <br>
 <button type="submit">Enviar</button>
</form>

<script>
 const passwordInput = document.getElementById("password");
 const passwordError = document.getElementById("password-error");

 passwordInput.addEventListener("input", () => {
  if (passwordInput.validity.patternMismatch) {
    passwordError.textContent = "La contraseña debe tener al menos 8 caracteres, una
letra y un número.";
  } else if (passwordInput.validity.valueMissing) {
    passwordError.textContent = "Este campo es obligatorio.";
  } else {
    passwordError.textContent = ""; // Sin errores
  }
 });
</script>
```

9.2.5. Restricciones en fechas (type="date")

El navegador puede restringir fechas mínimas y máximas. Desde JavaScript, puedes usar las propiedades min, max y el objeto Date.

Ejemplo:

```
<form id="dobForm">
 <label for="dob">Fecha de Nacimiento (mínimo: 2000-01-01):</label>
 <input type="date" id="dob" name="dob" min="2000-01-01" required>
 <span id="dob-error" style="color: red;"></span>
```

```
  <br>
  <button type="submit">Enviar</button>
</form>

<script>
  const dobInput = document.getElementById("dob");
  const dobError = document.getElementById("dob-error");

  dobInput.addEventListener("input", () => {
    if (dobInput.validity.rangeUnderflow) {
      dobError.textContent = `La fecha no puede ser anterior a ${dobInput.min}.`;
    } else if (dobInput.validity.valueMissing) {
      dobError.textContent = "Este campo es obligatorio.";
    } else {
      dobError.textContent = ""; // Sin errores
    }
  });
</script>
```

Resumen de la API de validación

- **Propiedades de** validity:
  - valueMissing: Campo vacío.
  - typeMismatch: Tipo de dato incorrecto.
  - patternMismatch: No coincide con el patrón.
  - rangeUnderflow y rangeOverflow: Fuera de los límites definidos por min y max.
  - tooShort y tooLong: Longitud fuera de los límites definidos.
  - stepMismatch: No cumple con el incremento definido por step.
- **Métodos clave**:
  - checkValidity(): Verifica si el campo cumple todas las restricciones.
  - setCustomValidity(mensaje): Define un mensaje de error personalizado.