UTN.BA FACULTAD REGIONAL BUENOS AIRES
SECRETARÍA DE EXTENSIÓN UNIVERSITARIA FRBA UTN

Centro de e-Learning

Desarrollo Web en HTML5, CSS3 y Javascript (avanzado)

Módulo 1:

HTML5 avanzado

Unidad 2:

DOM, formularios y API form

Objetivos:

Unidad 2: Que el alumno conozca las nuevas propiedades y atributos relacionados con el uso de formularios en HTML5.

Temario:

.: Api Form
    .: setCustomValidity();
    .: Evento invalid
.: Validación en tiempo real
    .: Propiedades de validación
    .: willValidate
Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de consignas que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

1. Los foros asociados a cada una de las unidades.
2. La Web 2.0.
3. Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen las actividades sugeridas y compartan en los foros los resultados obtenidos.

API FORM

Los formularios HTML cuentan con su propia API para personalizar todos los aspectos de procesamiento y validación.

Existen diferentes formas de aprovechar el proceso de validación en HTML5. Podemos usar los tipos de campo para activar el proceso de validación por defecto (por ejemplo, email) o volver un tipo de campo regular como text (o cualquier otro) en un campo requerido usando el atributo required. También podemos crear tipos de campo especiales usando pattern para personalizar requisitos de validación. Sin embargo, cuando se trata de aplicar mecanismos complejos de validación (por ejemplo, combinando campos o comprobando los resultados de un cálculo) deberemos recurrir a nuevos recursos provistos por esta API.

setCustomValidity()

Los navegadores que soportan HTML5 muestran un mensaje de error cuando el usuario intenta enviar un formulario que contiene un campo inválido.

Podemos crear mensajes para nuestros propios requisitos de validación usando el método ***setCustomValidity(mensaje)***.

Con este método especificamos un error personalizado que mostrará un mensaje cuando el formulario es enviado.

Cuando un mensaje vacío es declarado, el error es anulado.

```
<!DOCTYPE html>
<html lang="es">
<head>
<title>Formularios</title>
<script>
function iniciar(){
nombre1=document.getElementById("nombre");
nombre2=document.getElementById("apellido");
nombre1.addEventListener("input", validacion, false);
nombre2.addEventListener("input", validacion, false);
validacion();
}
function validacion(){
if(nombre1.value=='' && nombre2.value==''){
nombre1.setCustomValidity('inserte al menos un nombre');
nombre1.style.background='#FFDDDD';
}else{
```

```
nombre1.setCustomValidity('');
nombre1.style.background='#FFFFFF';
}
}
window.addEventListener("load", iniciar, false);
</script>
</head>
<body>
<section>
<form name="registracion" method="get">
Nombre:
<input type="text" name="nombre" id="nombre">
Apellido:
<input type="text" name="apellido" id="apellido">
<input type="submit" id="send" value="ingresar">
</form>
</section>
</body>
</html>
```

El código del ejemplo presenta una situación de validación compleja.

Dos campos fueron creados para recibir el nombre y apellido del usuario. Sin embargo, el formulario solo será inválido cuando ambos campos se encuentran vacíos. El usuario necesita ingresar solo uno de los campos, su nombre o su apellido, para validar la entrada.

En casos como éste no es posible usar el atributo required debido a que no sabemos cuál campo el usuario decidirá utilizar. Solo con código Javascript y errores personalizados podremos crear un efectivo mecanismo de validación para este escenario.

Nuestro código comienza a funcionar cuando el **evento load** es disparado. La función **iniciar()** es llamada para responder al evento. Esta función crea referencias para los dos elementos <input> y agrega una escucha para el evento input en ambos. Estas escuchas llamarán a la función **validacion()** cada vez que el usuario escribe dentro de los campos.

Debido a que los elementos <input> se encuentran vacíos cuando el documento es cargado, debemos declarar una condición inválida para no permitir que el usuario envíe el formulario antes de ingresar al menos uno de los valores. Por esta razón la función **validacion()** es llamada al comienzo. Si ambos campos están vacíos el error es generado y el color de fondo del campo nombre es cambiado a rojo. Sin embargo, si esta condición ya no es verdad porque al

menos uno de los campos fue completado, el error es anulado y el color del fondo de nombre es nuevamente establecido como blanco.

Es importante tener presente que el único cambio producido durante el procesamiento es la modificación del color de fondo del campo. El mensaje declarado para el error con **setCustomValidity()** será visible sólo cuando el usuario intente enviar el formulario.

El evento invalid

Cada vez que el usuario envía el formulario, un evento es disparado si un campo inválido es detectado. El evento es llamado **invalid** y es disparado por el elemento que produce el error.

Podemos agregar una escucha para este evento y así ofrecer una respuesta personalizada, como en el siguiente ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
<title>Formularios</title>
<script>
function iniciar(){
edad=document.getElementById("miedad");
edad.addEventListener("change", cambiarrango, false);
document.informacion.addEventListener("invalid",
validacion, true);
document.getElementById("enviar").addEventListener("click",
enviar, false);
}
function cambiarrango(){
var salida=document.getElementById("rango");
var calc=edad.value-20;
if(calc<20){
calc=0;
edad.value=20;
}
salida.innerHTML=calc+' a '+edad.value;
}
function validacion(e){
var elemento=e.target;
elemento.style.background='#FFDDDD';
}
function enviar(){
var valido=document.informacion.checkValidity();
if(valido){
document.informacion.submit();
```

```
}
}
window.addEventListener("load", iniciar, false);
</script>
</head>
<body>
<section>
<form name="informacion" method="get">
Usuario:
<input pattern="[A-Za-z]{3,}" name="usuario" id="usuario"
maxlength="10" required>
Email:
<input type="email" name="miemail" id="miemail" required>
Rango de Edad:
<input type="range" name="miedad" id="miedad" min="0"
max="80" step="20" value="20">
<output id="rango">0 a 20</output>
<input type="button" id="enviar" value="ingresar">
</form>
</section>
</body>
</html>
```

En el ejemplo, creamos un nuevo formulario con tres campos para ingresar el nombre de usuario, un email y un rango de 20 años de edad.

El campo usuario tiene tres atributos para validación: el atributo pattern solo admite el ingreso de un texto de tres caracteres mínimo, desde la A a la Z (mayúsculas o minúsculas), el atributo maxlength limita la entrada a 10 caracteres máximo, y el atributo required invalida el campo si está vacío. El campo miemail cuenta con sus limitaciones naturales debido a su tipo y además no podrá enviarse vacío gracias al atributo required. El campo miedad usa los atributos min, max, step y value para configurar las condiciones del rango.

También declaramos un elemento <output> para mostrar en pantalla una referencia del rango seleccionado.

Lo que el código Javascript hace con este formulario es simple: cuando el usuario hace clic en el botón “ingresar”, un evento invalid será disparado desde cada campo inválido y el color de fondo de esos campos será cambiado a rojo por la función validacion().

Veamos este procedimiento con un poco más de detalle. El código comienza a funcionar cuando el típico evento load es disparado luego que el documento fue

completamente cargado. La función iniciar() es ejecutada y tres escuchas son agregadas para los eventos change, invalid y click.

Cada vez que el contenido de los elementos del formulario cambia por alguna razón, el evento change es disparado desde ese elemento en particular. Lo que hicimos fue escuchar a este evento desde el campo range y llamar a la función cambiarrango() cada vez que el evento ocurre. Por este motivo, cuando el usuario desplaza el control del rango o cambia los valores dentro de este campo para seleccionar un rango de edad diferente, los nuevos valores son calculados por la función cambiarrango(). Los valores admitidos para este campo son períodos de 20 años (por ejemplo, 0 a 20 o 20 a 40).

Sin embargo, el campo solo retorna un valor, como 20, 40, 60 u 80. Para calcular el valor de comienzo del rango, restamos 20 al valor actual del campo con la fórmula edad.value - 20, y grabamos el resultado en la variable calc. El período mínimo admitido es 0 a 20, por lo tanto con un condicional if controlamos esta condición y no permitimos un período menor (estudie la función cambiarrango() para entender cómo funciona).

La segunda escucha agregada en la función iniciar() es para el evento invalid. La función validacion() es llamada cuando este evento es disparado para cambiar el color de fondo de los campos inválidos. Recuerde que este evento será disparado desde un campo inválido cuando el botón “ingresar” sea presionado. El evento no contiene una referencia del formulario o del botón “ingresar”, sino del campo que generó el error. En la función validacion() esta referencia es capturada y grabada en la variable elemento usando la variable e y la propiedad target. La construcción e.target retorna una referencia al elemento <input> inválido. Usando esta referencia, en la siguiente línea cambiamos el color de fondo del elemento.

Volviendo a la función iniciar(), encontraremos una escucha más que necesitamos analizar. Para tener control absoluto sobre el envío del formulario y el momento de validación, creamos un botón regular en lugar del típico botón submit.

Cuando este botón es presionado, el formulario es enviado, pero solo si todos sus elementos son válidos. La escucha agregada para el evento click en la función iniciar() ejecutará la función enviar() cuando el usuario haga clic sobre el botón. Usando el método checkValidity() solicitamos al navegador que realice el proceso de validación y solo enviamos el formulario usando el tradicional método submit() cuando ya no hay más condiciones inválidas.

Lo que hicimos con el código Javascript fue tomar control sobre todo el proceso de validación, personalizando cada aspecto y modificando el comportamiento del navegador.

VALIDACIÓN EN TIEMPO REAL

Cuando abrimos el archivo con la plantilla del ejemplo en el navegador, podremos notar que no existe una validación en tiempo real. Los campos son sólo validados cuando el botón “ingresar” es presionado. Para hacer más práctico nuestro sistema personalizado de validación, tenemos que aprovechar los atributos provistos por el objeto ValidityState.

```
<!DOCTYPE html>
<html lang="es">
<head>
<title>Formularios</title>
<script>
function iniciar(){
edad=document.getElementById("miedad");
edad.addEventListener("change", cambiarrango, false);
document.informacion.addEventListener("invalid",
validacion, true);
document.getElementById("enviar").addEventListener("click",
enviar, false);
document.informacion.addEventListener("input", controlar,
false);
}
function cambiarrango(){
var salida=document.getElementById("rango");
var calc=edad.value-20;
if(calc<20){
calc=0;
edad.value=20;
}
salida.innerHTML=calc+' a '+edad.value;
}
function validacion(e){
var elemento=e.target;
elemento.style.background='#FFDDDD';
}
function enviar(){
var valido=document.informacion.checkValidity();
if(valido){
document.informacion.submit();
}
```

```
}
function controlar(e){
var elemento=e.target;
if(elemento.validity.valid){
elemento.style.background='#FFFFFF';
}else{
elemento.style.background='#FFDDDD';
}
}
window.addEventListener("load", iniciar, false);
</script>
</head>
<body>
<section>
<form name="informacion" method="get">
Usuario:
<input pattern="[A-Za-z]{3,}" name="usuario" id="usuario"
maxlength="10" required>
Email:
<input type="email" name="miemail" id="miemail" required>
Rango de Edad:
<input type="range" name="miedad" id="miedad" min="0"
max="80" step="20" value="20">
<output id="rango">0 a 20</output>
<input type="button" id="enviar" value="ingresar">
</form>
</section>
</body>
</html>
```

En el ejemplo, una nueva escucha fue agregada para el evento input sobre el formulario. Cada vez que el usuario modifica un campo, escribiendo o cambiando su contenido, la función **controlar()** es ejecutada para responder a este evento.

La función **controlar()** también aprovecha la propiedad target para crear una referencia hacia el elemento que disparó el evento input. La validez del campo es controlada por medio del estado valid provisto por el atributo validity en la construcción elemento.validity.valid.

El estado valid será true (verdadero) si el elemento es válido y false (falso) si no lo es. Usando esta información cambiamos el color del fondo del elemento. Este color será, por lo tanto, blanco para un campo válido y rojo para uno inválido.

Con esta simple incorporación logramos que cada vez que el usuario modifica el valor de un campo del formulario, este campo será controlado y validado, y su condición será mostrada en pantalla en tiempo real.

Propiedades de validación

En el ejemplo anterior controlamos el estado valid. Este estado particular es un atributo del objeto **ValidityState** que retornará el estado de un elemento considerando cada uno de los posibles estados de validación. Si cada condición es válida entonces el valor del atributo valid será true (verdadero).

Existen ocho posibles estados de validez para las diferentes condiciones:

- **valueMissing** Este estado es true (verdadero) cuando el atributo required fue declarado y el campo está vacío.
- **typeMismatch** Este estado es true (verdadero) cuando la sintaxis de la entrada no corresponde con el tipo especificado (por ejemplo, si el texto insertado en un tipo de campo email no es una dirección de email válida).
- **patternMismatch** Este estado es true (verdadero) cuando la entrada no corresponde con el patrón provisto por el atributo pattern.
- **tooLong** Este estado es true (verdadero) cuando el atributo maxlength fue declarado y la entrada es más extensa que el valor especificado para este atributo.
- **rangeUnderflow** Este estado es true (verdadero) cuando el atributo min fue declarado y la entrada es menor que el valor especificado para este atributo.
- **rangeOverflow** Esta estado es true (verdadero) cuando el atributo max fue declarado y la entrada es más grande que el valor especificado para este atributo.
- **stepMismatch** Este estado es true (verdadero) cuando el atributo step fue declarado y su valor no corresponde con los valores de atributos como min, max y value.
- **customError** Este estado es true (verdadero) cuando declaramos un error personalizado usando el método setCustomValidity() estudiado anteriormente.

Para controlar estos estados de validación, debemos utilizar la sintaxis elemento.validity.estado (donde estado es cualquiera de los valores listados arriba). Podemos aprovechar estos atributos para saber exactamente que originó el error en un formulario, como en el siguiente ejemplo:

```
function enviar(){
var elemento=document.getElementById("usuario");
var valido=document.informacion.checkValidity();
if(valido){
document.informacion.submit();
}else if(elemento.validity.patternMismatch ||
elemento.validity.valueMissing){
alert('el nombre de usuario debe tener mínimo de 3 caracteres');
}
}
```

En el ejemplo, la función enviar() fue modificada para incorporar esta clase de control. El formulario es validado por el método checkValidity() y si es válido es enviado con submit(). En caso contrario, los estados de validación **patternMismatch** y **valueMissing** para el campo usuario son controlados y un mensaje de error es mostrado cuando el valor de alguno de ellos es true (verdadero).

willValidate

En aplicaciones dinámicas es posible que los elementos involucrados no tengan que ser validados. Este puede ser el caso, por ejemplo, con botones, campos ocultos o elementos como <output>. La API nos ofrece la posibilidad de detectar esta condición usando el atributo willValidate y la sintaxis elemento.willValidate.