UT07.1: Formularios DOM

Contenido

1. Introducción

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.

En este documento nos vamos a enfocar en el tratamiento nativo de los formularios y sus objetos.

2. Objetos relacionados con los formularios

Además del uso de identificadores, el API de DOM ofrece dos tipos de colecciones que nos sirven para seleccionar los formularios de las páginas y sus elementos, las colecciones `forms` y `elements`. Veamos en qué consisten.

Como ejemplo vamos a utilizar un formulario de *login* en la página.

```
<h1 class="display-5">Login</h1>
<form name="login" role="form">
    <div class="form-group">
        <label for="email">Email address:</label>
        <input type="email" class="form-
control" id="email" name="email" value="test">
    </div>
    <div class="form-group">
        <label for="pwd">Password:</label>
        <input type="password" class="form-
control" id="pwd" name="pwd" value="12345">
    </div>
    <div class="checkbox">
        <label><input type="checkbox" name="remember">Remember me</label>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

2.1. Colección `forms`

El objeto `document` tiene la colección de formularios de la página en la propiedad `forms`. Podemos consultar el número total de formularios de la página con la propiedad `length`.

```
console.log(document.forms.length);
```

La colección nos ofrece un mecanismo para evitar el exceso de identificadores en la página. Como toda colección nos permite utilizar una notación corchete para acceder al objeto que representa el formulario. Accedemos al primer formulario para comprobar su atributo `name`.

```
console.log(document.forms[0].name);
```

También podemos acceder con esta notación utilizando el nombre del formulario.

```
console.log(document.forms['login'].name);
```

Los formularios de la colección están mapeados para facilitar el acceso, por lo que podemos también utilizar una notación punto.

```
console.log(document.forms.login.name);
```

Tenemos que recordar que el protocolo HTTP los elementos de los formularios para ser transmitidos al servidor deben utilizar el atributo `name`. Los elementos con atributo `name` también están mapeados dentro del objeto formulario, pudiendo acceder a sus atributos como por ejemplo el atributo `value`.

```
console.log(document.forms.login.pwd.name);
console.log(document.forms.login.pwd.value);
```

2.2. Colección `elements`

Cada formulario tiene asociado una colección `elements` que contiene todos los elementos `input`, `select` y `button` asociados con el formulario, incluidos los que no tiene asociado un atributo `name`. Como colección podemos utilizar una notación corchete para acceder a los elementos tanto con índice como con string.

```
console.log(document.forms.login.elements.length);
console.log(document.forms.login.elements[0].name);
console.log(document.forms.login.elements[0].value);
console.log(document.forms['login'].elements['pwd'].name);
console.log(document.forms['login'].elements['pwd'].value);
```

Los elementos están mapeados por nombre para seleccionarlo con una notación punto o con el corchete utilizando el string.

```
console.log(document.forms.login.elements.pwd.name);
console.log(document.forms.login.elements.pwd.value);
```

Por último, debemos destacar que los elementos `button`, y los `input` sin atributo `name` también estarán incluidos en la colección, aunque no se recogerán para ser transmitidos con el protocolo HTTP al servidor.

```
console.log(document.forms.login.elements[3].type);
```

2.3. Propiedad `form`

De la misma forma que desde un objeto tenemos acceso a sus elementos, cada uno de los elementos tiene referencia al formulario donde está contenido a través de la propiedad `form`.

En el siguiente ejemplo hemos añadido un manejador de eventos al botón de tipo *submit* del formulario, para que al hacer click cancele la ejecución de evento *submit* vinculado como acción por defecto. La referencia `this` apunta al propio botón, y en su propiedad `form` nos da acceso al formulario, pudiendo recorrer el resto de los elementos que tiene vinculados.

```
document.forms.login.elements[3].addEventListener('click',
function (event){
  event.preventDefault();
  console.log(this.type);
  console.log(this.form.name);
  this.form.email.value = 'cambiado@test.es';
  this.form.pwd.value = '67890';
  console.log(this.form.email.value);
  console.log(this.form.pwd.value);
});
```

3. Manipulación de la página a través de formularios

Los elementos de un formulario son parte del árbol DOM, por lo que podemos obtener sus valores, en base a los cuales podemos alterar el contenido de la página, o el contenido del propio formulario.

Vamos a crear un conjunto de botones de tipo radio que nos permita seleccionar entre tipos de productos. En base al que tengamos seleccionado, debemos modificar los elementos del formulario para adaptarlo al tipo de producto seleccionado.

Creamos un formulario utilizando como contenedores elementos `filedset`. El primer `filedset` contendrá los botones `radio`, el resto contendrá los elementos adaptados al tipo de producto. Una ventaja del `filedset` es que, si le añadimos la propiedad `disabled`, todos los elementos contenidos automáticamente quedan deshabilitados y no son enviados al servidor con el evento *submit*. El primer `filedset` correspondiente al primer producto estará visible y habilitado, el resto estarán ocultos y deshabilitados.

```
<h1 class="display-5">Formulario dinámico</h1>
<form name="dynamic" role="form">
    <fieldset class="form-group">
        <div class="row">
            <legend class="col display-6">Tipos de productos</legend>
        </div>
        <div class="row">
            <div class="form-check col-3">
                <input class="form-check-
input" type="radio" name="product" id="laptop" value="Laptop" checked>
                <label class="form-check-label" for="laptop">
                    Portátil
                </label>
            </div>
            <div class="form-check col-3">
                <input class="form-check-
input" type="radio" name="product" id="camera" value="Camera">
                <label class="form-check-label" for="camera">
                    Cámara
                </label>
            </div>
            <div class="form-check col-3">
```

```
                <input class="form-check-
input" type="radio" name="product" id="smartphone" value="Smartphone">
                <label class="form-check-label" for="smartphone">
                    Teléfono móvil
                </label>
            </div>
            <div class="form-check col-3">
                <input class="form-check-
input" type="radio" name="product" id="tablet" value="Tablet">
                <label class="form-check-label" for="tablet">
                    Tablet
                </label>
            </div>
        </div>
    </fieldset>
    <fieldset name="fsLaptop" class="form-group details">
        <legend class="col display-7">Portátil</legend>
        <div class="form-group">
            <label for="laptop-os">Sistema operativo portátil</label>
            <input type="text" id="laptop-os" name="os" class="form-
control" placeholder="Sistema operativo">
        </div>
        <div class="form-group">
            <label for="laptop-memory">Memoria</label>
            <input type="text" id="laptop-
memory" name="memory" class="form-control" placeholder="Memoria">
        </div>
    </fieldset>
    <fieldset class="d-none form-group details" name="fsCamera" disabled>
        <legend class="col display-7">Cámara</legend>
        <div class="form-group">
            <label for="camera-memory">Memoria</label>
            <input type="text" id="camera-
memory" name="memory" class="form-control" placeholder="Memoria">
        </div>
    </fieldset>
    <fieldset class="d-none form-
group details" name="fsSmartphone" disabled>
        <legend class="col display-7">Teléfono</legend>
        <div class="form-group">
            <label for="smartphone-size">Tamaño</label>
            <input type="text" id="smartphone-
size" name="size" class="form-control" placeholder="Tamaño">
        </div>
        <div class="form-group">
            <label for="smartphone-memory">Memoria</label>
```

```
            <input type="text" id="smartphone-
memory" name="memory" class="form-control" placeholder="Memoria">
        </div>
    </fieldset>
    <fieldset class="d-none form-group details" name="fsTablet" disabled>
        <legend class="col display-7">Tablet</legend>
        <div class="form-group">
            <label for="tablet-os">Sistema operativo tablet</label>
            <input type="text" id="tablet-os" name="os" class="form-
control" placeholder="Sistema operativo">
        </div>
        <div class="form-group">
            <label for="tablet-size">Tamaño</label>
            <input type="text" id="tablet-size" name="size" class="form-
control" placeholder="Tamaño">
        </div>
    </fieldset>
    <button id="b-dynamic" type="submit" class="btn btn-
primary">Enviar</button>
</form>
```

Para cada *radio botón* le asociamos un manejador de eventos para capturar el evento *click*. Para seleccionar fácilmente los elementos `fieldset` los hemos nombrado con el prefijo *fs* seguido del valor de cada *radio botón*, así como una clase *details* que los diferencian del principal.

```
const radios = document.forms.dynamic.product;
for (const radio of radios) {
  radio.addEventListener('click', radioHandler);
}
```

En el manejador encontramos todos los elementos `fieldset` y le añadimos una clase `d-none` para ocultarlos. Los elementos `fieldset` también están mapeados en el formulario, por lo que a través de su atributo `name` lo podemos buscar concatenando el prefijo, este elemento será el activo, por lo que le borramos la clase `d-none`. Podemos modificar la propiedad `disabled` para deshabilitar los elementos contenidos en el `fieldset`.

```
const radioHandler = (event) => {
  const fieldset =
event.currentTarget.form[`fs${event.currentTarget.value}`];
  const fieldsets =
event.currentTarget.form.querySelectorAll('fieldset.details');
  for (const fd of fieldsets) {
    fd.classList.add('d-none');
    fd.disabled = true;
  }
  fieldset.classList.remove('d-none');
  fieldset.disabled = false;
};
```

En el *botón de submit* hemos cancelado el comportamiento por defecto. Podemos acceder a los *radio botones* utilizando el atributo `name`, al ser 4 objetos están contenidos en una colección. La propiedad `checked` nos indica cuál de ellos está habilitado.

```
document.getElementById('b-dynamic').addEventListener('click', (event) =>
{
  event.preventDefault();
  console.log('##########');
  console.log(event.currentTarget.form.product.value);
  for (const radio of event.currentTarget.form.product) {
    console.log(radio.checked);
  }
});
```

Nota aclaratoria: si como función manejadora de eventos utilizamos una función arrow tal y como hemos hecho en los ejemplos anteriores, la referencia `this` apunta al objeto `window`, debemos recordar este tipo de funciones no tienen contexto por lo tanto, para recoger el elemento que captura el evento lo tenemos que hacer a través del propio evento con `event.currentTarget`.

4. 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.

4.1. Validación del navegador

La especificación de HTML5 define varios tipos de **restricciones** que podemos declarar en un formulario. Empezamos construyendo un formulario para que el navegador pueda realizar la validación.

https://developer.mozilla.org/es/docs/Web/Guide/HTML/HTML5/Constraint_validation

Creamos dos botones un `submit` y un `reset`.

```
<div class="container" id="cValidation">
    <h1 class="display-5">Validación de formulario</h1>
    <form name="fValidation" role="form" class="row g-3">
        <div class="mb-12">
            <button class="btn btn-primary" type="submit">Enviar</button>
            <button class="btn btn-primary"
type="reset">Cancelar</button>
        </div>
    </form>
</div>
```

Definimos algunas de las restricciones que nos ofrece la especificación.

4.1.1. Propiedad `requiered`

Nos permite indicar si un elemento es obligatorio. Creamos tres elementos para recoger un nombre y dos apellidos. Los dos primeros los declaramos como obligatorios con la propiedad.

```
<div class="col-md-4">
    <label class="form-label" for="vfName">Nombre *</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-person-
square"></i></span>
        <input type="text" class="form-control" id="vfName" name="vfName"
placeholder="Nombre" value=""
            required>
        <div class="invalid-feedback">El nombre es obligatorio.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
<div class="col-md-4">
    <label class="form-label" for="vfSurname1">Primer apellido *</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-person-
square"></i></span>
        <input type="text" class="form-control" id="vfSurname1"
name="vfSurname1"
            placeholder="Primer apellido" value="" required>
        <div class="invalid-feedback">El primer apellido es
obligatorio.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
<div class="col-md-4">
    <label class="form-label" for="vfSurname2">Segundo apellido</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-person-
square"></i></span>
```

```
        <input type="text" class="form-control" id="vfSurname2"
name="vfSurname2"
            placeholder="Segundo apellido" value="">
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
```

Al ejecutar el *submit*, si no hemos cumplimentado los elementos obligatorios, el navegador nos impide realizarlo.

4.1.2. Tipos de datos

La siguiente restricción que podemos hacer es mediante tipos de elementos. Además del tipo `text`, podemos encontrar `date` que podemos utilizar para una fecha de nacimiento, `email`, que valida una dirección de correo electrónico, `url`, para introducir una url que necesitemos, o `number`, para utilizar un valor numérico, entre otros tipos. Para este último tipo tenemos los atributos `min` y `max` que nos permiten indicar el número mínimo y máximo.

```
<div class="col-md-3">
    <label class="form-label" for="vfBirth">Fecha de nacimiento *</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-calendar-
date"></i></span>
        <input type="date" class="form-control" id="vfBirth"
name="vfBirth"
            placeholder="Fecha de nacimiento" value="" required>
        <div class="invalid-feedback">El primer apellido es
obligatorio.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
<div class="col-md-3">
    <label class="form-label" for="vfEmail">Correo electrónico *</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-envelope-
at"></i></span>
        <input type="email" class="form-control" id="vfEmail"
name="vfEmail"
            placeholder="test@test.es" value="" required>
        <div class="invalid-feedback">El formato del correo electrónico
no es correcto.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
<div class="col-md-3">
    <label class="form-label" for="vfUrl">URL *</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-link"></i></span>
        <input type="url" class="form-control" id="vfUrl" name="vfUrl"
            placeholder="http://www.test.es" value="" required>
```

```
        <div class="invalid-feedback">La URL no es válida.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
<div class="col-md-3">
    <label class="form-label" for="vfNumber">Número entre 1 y 10.
*</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-list-
ol"></i></span>
        <input type="number" class="form-control" id="vfNumber"
name="vfNumber" value="" required
            min="1" max="10">
        <div class="invalid-feedback">El número debe ser positivo entre 1
y 10.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
```

4.1.3. Patrones

Los tipos de elemento definen patrones predeterminado. Podemos aplicar expresiones regulares a los valores de un elemento de tipo texto utilizando el atributo `pattern`. Hemos definidos tres elementos que deben cumplir un patrón específico, DNI, un teléfono y una tarjeta de crédito.

```
<div class="col-md-4">
    <label class="form-label" for="vfDni">DNI *</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-person-
square"></i></span>
        <input type="text" class="form-control" id="vfDni" name="vfDni"
placeholder="12345678A"
            value="" required maxlength="9" pattern="^[0-9]{8}[A-Z]$">
        <div class="invalid-feedback">El DNI debe estar formado por 8
digitos y una letra.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
<div class="col-md-4">
    <label class="form-label" for="vfPhone">Teléfono *</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-
telephone"></i></span>
        <input type="text" class="form-control" id="vfPhone"
name="vfPhone"
            placeholder="(6-9)XXXXXXXX" value="" required maxlength="9"
pattern="^[96][0-9]{8}$">
```

```
        <div class="invalid-feedback">El número de teléfono debe ser un
móvil o un fijo.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
<div class="col-md-4">
    <label class="form-label" for="vfCreditCard">Tarjeta de crédito
*</label>
    <div class="input-group">
        <span class="input-group-text"><i class="bi bi-credit-
card"></i></span>
        <input type="text" class="form-control" id="vfCreditCard"
name="vfCreditCard"
            placeholder="XXXX XXXX XXXX XXXX" value="" required
maxlength="19"
            pattern="^[0-9]{4} [0-9]{4} [0-9]{4} [0-9]{4}$">
        <div class="invalid-feedback">La tarjeta de crédito no es
válida.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
```

4.1.4. Contraseñas

Los tipos de elementos `password` son realmente tipos de elementos `text` con visualización codificada. Vamos a crear un elemento para solicitar la contraseña y su confirmación. El problema que tenemos es que la especificación no nos permite relacionar dos elementos para que tengan ambos el mismo valor. Hemos añadido un patrón para que la contraseña tenga al menos un dígito, una mayúscula, una minúscula y una longitud entre 8 y 16 caracteres.

```
<div class="col-md-6 form-floating mb-3">
    <input type="password" class="form-control" id="vfPwd" name="vfPwd"
placeholder="" value=""
        pattern="^(?=\w*\d)(?=\w*[A-Z])(?=\w*[a-z])\S{8,16}$" required>
    <label for="vfPwd">Contraseña *</label>
    <div class="invalid-feedback">La contraseña debe tener al entre 8 y
16 caracteres, al
        menos
        un dígito, al menos una minúscula y al menos una mayúscula. Puede
tener otros
        símbolos.
    </div>
    <div class="valid-feedback">Correcto.</div>
</div>
<div class="col-md-6 form-floating mb-3">
    <input type="password" class="form-control" id="vfConfirm"
name="vfConfirm" placeholder=""
        value="" pattern="^(?=\w*\d)(?=\w*[A-Z])(?=\w*[a-z])\S{8,16}$"
required>
```

```
    <label for="vfConfirm">Confirmación de contraseña *</label>
    <div class="invalid-feedback">La contraseña y la confirmación deben
ser iguales.</div>
    <div class="valid-feedback">Correcto.</div>
</div>
```

4.1.5. Más tipos de elementos

Terminamos el formulario utilizando tipos de elemento `file` y un `checkbox` típico para aceptar términos de condiciones.

```
<div class="col-mb-12">
    <label for="vfFile" class="form-label">Adjunta imagen con extenxión
jpg, png o gif. *</label>
    <input class="form-control" type="file" id="vfFile" name="vfFile"
required>
    <div class="invalid-feedback">Debe seleccionar un archivo con
extensión jpg, png o gif.</div>
    <div class="valid-feedback">Correcto.</div>
</div>
<div class="col-mb-12">
    <div class="form-check">
        <input class="form-check-input" type="checkbox" value=""
id="vfTerms" name="vfTerms">
        <label class="form-check-label" for="vfTerms" required>
            Acepta los términos de uso.
        </label>
        <div class="invalid-feedback">Debe aceptar los términos de
uso.</div>
        <div class="valid-feedback">Correcto.</div>
    </div>
</div>
```

4.2. API de Validación

Tenemos una serie de métodos y propiedades asociadas a los elementos que nos permiten personalizar la validación del navegador con JavaScript.

4.2.1. Cambiar mensaje por defecto

Podemos cambiar el mensaje por defecto que muestra los elementos no válidos con el método `setCustomValidity()`. El evento `invalid` se ejecuta cuando el valor no es válido. Asignamos un manejador de evento que cambia el mensaje por defecto para el elemento.

```
let form = document.forms.fValidation;
form.vfName.addEventListener('invalid', function(){
  this.setCustomValidity('El nombre es obligatorio.');
});
```

El método `setCustomValidity()` se utiliza para hacer validaciones personalizadas. Tener un mensaje fijado implica que el elemento tiene un valor inválido, por lo que antes de volver a validarse debemos asignarle la cadena vacía. Lo hacemos en el evento `change`.

```
form.vfName.addEventListener('change', function(){
  this.setCustomValidity('');
});
```

Repetimos para el resto de los elementos.

```
form.vfSurname1.addEventListener('invalid', function(){
  this.setCustomValidity('El primer apellido es obligatorio.');
});
form.vfSurname1.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfBirth.addEventListener('invalid', function(){
  this.setCustomValidity('La fecha de nacimiento es incorrecta.');
});
form.vfBirth.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfEmail.addEventListener('invalid', function(){
  this.setCustomValidity('El formato del correo electrónico no es correct
o.');
});
form.vfEmail.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfUrl.addEventListener('invalid', function(){
  this.setCustomValidity('La URL no es válida.');
});
form.vfUrl.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfNumber.addEventListener('invalid', function(){
  this.setCustomValidity('El número debe ser positivo entre 1 y 10.');
});
form.vfNumber.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfDni.addEventListener('invalid', function(){
  this.setCustomValidity('El DNI debe estar formado por 8 digitos y una l
etra.');
});
form.vfDni.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfPhone.addEventListener('invalid', function(){
  this.setCustomValidity('El número de teléfono debe ser un móvil o un fi
jo.');
});
form.vfPhone.addEventListener('change', function(){
```

```
  this.setCustomValidity('');
});
form.vfCreditCard.addEventListener('invalid', function(){
  this.setCustomValidity('La tarjeta de crédito no es válida.');
});
form.vfCreditCard.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfPwd.addEventListener('invalid', function(){
  this.setCustomValidity('La contraseña debe tener entre 8 y 16 caractere
s, al menos un dígito, al menos una minúscula y al menos una mayúscula. P
uede tener otros símbolos.');
});
form.vfPwd.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfConfirm.addEventListener('invalid', function(){
  this.setCustomValidity('La contraseña debe tener entre 8 y 16 caractere
s, al menos un dígito, al menos una minúscula y al menos una mayúscula. P
uede tener otros símbolos.');
});
form.vfConfirm.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfFile.addEventListener('invalid', function(){
  this.setCustomValidity('Debe seleccionar un archivo con extensión jpg,
png o gif.');
});
form.vfFile.addEventListener('change', function(){
  this.setCustomValidity('');
});
form.vfTerms.addEventListener('invalid', function(){
  this.setCustomValidity('Debe aceptar los términos de uso.');
});
form.vfTerms.addEventListener('change', function(){
  this.setCustomValidity('');
});
```

4.2.2. Objeto `validity`

Este objeto recoge propiedades booleanas relacionadas con la validación del elemento.

- `customError`: Indica si hemos fijado el mensaje de validación para personalizar la validación.
- `patternMismatch`: Indica que si el valor cumple con el patrón.
- `rangeOverflow`: Nos dice si valor está por encima del atributo `max`.
- `rangeUnderflow`: Nos dice si el valor está por debajo del atributo `min`.
- `stepMismatch`: Nos dice si el valor no coincide con los especificado en el atributo `step`.

- `tooLong`: El valor tiene una longitud mayor que el atributo `maxlength`.
- `typeMismatch`: El tipo no coincide con el especificado en el elemento.
- `valueMissing`: Nos indica que no existe valor siendo el elemento obligatorio.
- `valid`: Nos dice que el valor es válido.

Vamos a modificar el evento `invalid` del dato numérico para personalizar el tipo de mensaje en función de si hemos superado el límite por encima o por debajo del rango.

```
form.vfNumber.addEventListener('invalid', function(){
  if (this.validity.rangeOverflow){
    this.setCustomValidity('El valor está po encima del límite.');
  }
  if (this.validity.rangeUnderflow){
    this.setCustomValidity('El valor está por debajo del límite.');
  }
});
```

4.2.3. Personalizar errores

El método `setCustomValidity()` nos permite configurar nuestros propios errores. Validamos que las contraseñas sean iguales con el evento `change` de la confirmación.

```
form.vfConfirm.addEventListener('change', function(){
  if (form.vfPwd.value !== form.vfConfirm.value){
    this.setCustomValidity('Las constaseñas deben ser iguales.');
  } else {
    this.setCustomValidity('');
  }
});
```

La no coincidencia implica que se dispare el evento `invalid`, por lo que tenemos que comprobar qué tipo de error tenemos para asignar el mensaje correcto.

```
form.vfConfirm.addEventListener('invalid', function(){
  if (this.validity.patternMismatch){
    this.setCustomValidity('La contraseña debe tener al entre 8 y 16 cara
cteres, al menos un dígito, al menos una minúscula y al menos una mayúscu
la. Puede tener otros símbolos.');
  }
});
```

Podemos validar las propiedades del fichero adjuntado. Creamos dos funciones para chequear la extensión y el tamaño del archivo.

```
function checkFileExtension(file, allowedExtensions){
  let fileExtension = file.name.split('.').pop().toLowerCase();
  return allowedExtensions.some((extension) => {
    return extension === fileExtension
  });
}
function checkFileSize(file, size) {
  return (file.size > size * 1024);
}
```

En el evento `change` del fichero, chequeamos que la extensión sea la de una imagen, y que su tamaño sea de 10KB.

```
form.vfFile.addEventListener('change', function(){
  this.setCustomValidity('');
  if (!checkFileExtension(this.files[0], ['jpg','png','gif'])){
    this.setCustomValidity('Debe seleccionar un archivo con extensión jpg
, png o gif.');
  }
  let size = 10;
  if (checkFileSize(this.files[0], size)){
    this.setCustomValidity(`"El archivo ${this.files[0].name} no debe ser
 mayor a ${size}KB"`);
  }
});
```

En este caso podemos eliminar el evento `invalid`.

4.3. Validación JavaScript

La validación por parte del navegador tiene los problemas de que no se integra con el diseño de nuestra página, y que puede causar frustración en el usuario, ya que la validación se hace elemento a elemento en envío de la información. La solución pasa por realizar una validación a media utilizando código JavaScript.

Lo primero que debemos hacer es deshabilitar de forma declarativa la validación del formulario utilizando la propiedad `novalidate`.

```
<form name="fValidation" role="form" novalidate>
```

O por código sería así.

```
let form = document.forms.fValidation;
form.setAttribute('novalidate', true);
```

4.3.1. Validación en el `submit`

En este caso, al lanzarse el evento `submit` debemos validar todos los elementos del formulario, indicando cuales son válidos y cuales no de una manera clara para el usuario. Añadimos un manejador el evento `submit`. En el manejador añadimos una variable booleana que indica si está todo correcto, en caso de existir algún problema debemos cancelar al final de la función el comportamiento por defecto del evento. También guardamos la referencia del primer elemento inválido para pasarle el foco.

```
form.addEventListener('submit', function(event){
  let isValid = true;
  let firstInvalidElement = null;

  if (!isValid) {
    event.preventDefault();
    event.stopPropagation();
    firstInvalidElement.focus();
  }
});
```

Vamos a crear una función que muestre la retroalimentación al usuario para cada elemento del formulario. La función recibe el elemento como argumento, si es válido y el mensaje como optativo. La capa de retroalimentación la calculamos en base al elemento, el estilo se aplica en base a si es válido o no y el mensaje puede sobrescribir el texto por defecto que tiene la capa.

En el formulario HTML tenemos ya las capas para la retroalimentación con el mensaje creado, aunque hemos preparado la función para sobrescribir el mensaje en caso de tener varios tipos de errores.

```
function showFeedBack(input, valid, message) {
  const validClass = (valid) ? 'is-valid' : 'is-invalid';
  const messageDiv = (valid) ?
input.parentElement.querySelector('div.valid-feedback') :
input.parentElement.querySelector('div.invalid-feedback');
  for (const div of input.parentElement.getElementsByTagName('div')) {
    div.classList.remove('d-block');
  }
  messageDiv.classList.remove('d-none');
  messageDiv.classList.add('d-block');
  input.classList.remove('is-valid');
  input.classList.remove('is-invalid');
  input.classList.add(validClass);
  if (message) {
    messageDiv.innerHTML = message;
  }
}
```

Al tener que validar todos los elementos del formulario, es independiente empezar por el primero o por el último. Vamos a empezar por el último para que al final tengamos la referencia del primer elemento no válido guardada en su variable.

4.3.1.1. Validación de cajas de chequeo

Comenzamos verificando que hemos aceptado los términos de uso. La caja de chequeo tiene una propiedad `checked` que indica si lo hemos cliqueado. Buscamos la capa de retroalimentación en base al elemento a través de sus hermanos, ya que la estructura HTML es conocida.

```
if (!this.vfTerms.checked) {
  isValid = false;
  showFeedBack(this.vfTerms, false);
  firstInvalidElement = this.vfTerms;
} else {
  showFeedBack(this.vfTerms, true);
}
```

4.3.1.2. Validación de fichero

Los elementos de tipo `file` tienen una propiedad `files` con un array con todos los ficheros que hayamos seleccionado. Utilizamos esta referencia para realizar el chequeo de la extensión y del peso de fichero reutilizando la función de la validación del navegador. Realizamos una encadenación de validaciones.

```
const size = 10;
if (!this.vfFile.value) {
  isValid = false;
  firstInvalidElement = this.vfFile;
  const message = 'Adjuntar un fichero es obligatorio.';
  showFeedBack(this.vfFile, false, message);
} else if (!checkFileExtension(this.vfFile.files[0], ['jpg', 'png',
'gif'])) {
  isValid = false;
  firstInvalidElement = this.vfFile;
  const message = 'Debe seleccionar un archivo con extensión jpg, png o
gif.';
  showFeedBack(this.vfFile, false, message);
} else if (checkFileSize(this.vfFile.files[0], size)) {
  isValid = false;
  firstInvalidElement = this.vfFile;
  const message = `El archivo ${this.vfFile.files[0].name} no debe ser
mayor a ${size}KB`;
  showFeedBack(this.vfFile, false, message);
} else {
  showFeedBack(this.vfFile, true, message);
}
```

4.3.1.3. Validación de elementos relacionados

Las contraseñas son elementos relacionados que deben ser validados en común, además deben cumplir un patrón. Empezamos por el patrón. El método `checkValidity()` devuelve un booleano indicando si un elemento cumple con las restricciones declaradas en el documento.

Primero validamos si la contraseña cumple con las restricciones, para posteriormente comparar contraseña y su confirmación para ver si son iguales.

```
if (!this.vfPwd.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfPwd, false);
  firstInvalidElement = this.vfPwd;
} else if (this.vfPwd.value !== this.vfConfirm.value) {
  isValid = false;
  showFeedBack(this.vfPwd, false, 'La contraseña y la confirmación deben
ser iguales.');
  showFeedBack(this.vfConfirm, false, ' ');
  firstInvalidElement = this.vfPwd;
} else {
  showFeedBack(this.vfPwd, true);
  showFeedBack(this.vfConfirm, true, ' ');
}
```

4.3.1.4. Validación de elementos basados en patrones

Utilizamos el método `checkValidity()` para chequear estas restricciones.

```
this.vfCreditCard.value = this.vfCreditCard.value.trim();
if (!this.vfCreditCard.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfCreditCard, false);
  firstInvalidElement = this.vfCreditCard;
} else {
  showFeedBack(this.vfCreditCard, true);
}

this.vfPhone.value = this.vfPhone.value.trim();
if (!this.vfPhone.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfPhone, false);
  firstInvalidElement = this.vfPhone;
} else {
  showFeedBack(this.vfPhone, true);
}

this.vfDni.value = this.vfDni.value.trim();
if (!this.vfDni.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfDni, false);
  firstInvalidElement = this.vfDni;
} else {
  showFeedBack(this.vfDni, true);
}
```

4.3.1.5. Validación de elementos basados en tipos

Los tipos son restricciones que hace el navegador, por lo que podemos utilizar el método `checkValidity()` para comprobar si el valor cumple con la declaración del documento. El tipo numérico podemos validar qué tipo de error tiene el valor.

```
if (!this.vfNumber.checkValidity()) {
  isValid = false;
  let message;
  if (this.vfNumber.validity.rangeOverflow) {
    message = 'El valor está po encima del límite.';
  }
  if (this.vfNumber.validity.rangeUnderflow) {
    message = 'El valor está por debajo del límite.';
  }
  showFeedBack(this.vfNumber, false, message);
  firstInvalidElement = this.vfNumber;
} else {
  showFeedBack(this.vfNumber, true);
```

```
}

if (!this.vfUrl.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfUrl, false);
  firstInvalidElement = this.vfUrl;
} else {
  showFeedBack(this.vfUrl, true);
}

if (!this.vfEmail.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfEmail, false);
  firstInvalidElement = this.vfEmail;
} else {
  showFeedBack(this.vfEmail, true);
}

if (!this.vfBirth.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfBirth, false);
  firstInvalidElement = this.vfBirth;
} else {
  showFeedBack(this.vfBirth, true);
}
```

4.3.1.6. Validación de elementos obligatorios

Seguimos utilizando el método `checkValidity()` para validar esta restricción.

```
this.vfSurname2.value = this.vfSurname2.value.trim();
showFeedBack(this.vfSurname2, true);

if (!this.vfSurname1.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfSurname1, false);
  firstInvalidElement = this.vfSurname1;
} else {
  showFeedBack(this.vfSurname1, true);
}

if (!this.vfName.checkValidity()) {
  isValid = false;
  showFeedBack(this.vfName, false);
  firstInvalidElement = this.vfName;
} else {
  showFeedBack(this.vfName, true);
}
```

4.3.2. Reseteo del formulario

Capturamos el evento `reset` del formulario para eliminar las capas de retroalimentación y los estilos de los elementos.

```
form.addEventListener('reset', (function (event) {
  for (const div of this.querySelectorAll('div.valid-feedback,
div.invalid-feedback')) {
    div.classList.remove('d-block');
    div.classList.add('d-none');
  }
  for (const input of this.querySelectorAll('input')) {
    input.classList.remove('is-valid');
    input.classList.remove('is-invalid');
  }
  this.vfName.focus();
}));
```

4.4. Validación en línea

Para formularios muy grandes con muchos elementos, hacer una validación justo en el evento *submit* puede causar frustración al usuario si tiene mucha información que corregir. La validación en línea la tenemos que realizar elemento a elemento, lo que implica añadir un manejador a cada uno de ellos.

La validación en línea mejora la usabilidad del formulario, pero es un complemento a la validación en el *submit* del formulario.

Vamos a crear un manejador de eventos genérico para asociarlo con el evento change de cada uno de los elementos del formulario. De esta forma podemos realizar la validación en línea.

```
function defaultCheckElement(event) {
  this.value = this.value.trim();
  if (!this.checkValidity()) {
    showFeedBack(this, false);
  } else {
    showFeedBack(this, true);
  }
}
```

Ahora tenemos que asociarlo con cada elemento del formulario.

```
form.vfName.addEventListener('change', defaultCheckElement);
form.vfSurname1.addEventListener('change', defaultCheckElement);
form.vfSurname2.addEventListener('change', defaultCheckElement);
form.vfBirth.addEventListener('change', defaultCheckElement);
form.vfEmail.addEventListener('change', defaultCheckElement);
form.vfUrl.addEventListener('change', defaultCheckElement);
form.vfDni.addEventListener('change', defaultCheckElement);
form.vfPhone.addEventListener('change', defaultCheckElement);
form.vfCreditCard.addEventListener('change', defaultCheckElement);
form.vfPwd.addEventListener('change', defaultCheckElement);
form.vfConfirm.addEventListener('change', defaultCheckElement);
```

```
form.vfTerms.addEventListener('change', defaultCheckElement);
```

Los elementos que tienen una validación más compleja como el dato numérico o el fichero, tenemos que asignar manejadores a medida.

```
form.vfNumber.addEventListener('change', function (event) {
  if (!this.checkValidity()) {
    let message;
    if (this.validity.rangeOverflow) {
      message = 'El valor está po encima del límite.';
    }
    if (this.validity.rangeUnderflow) {
      message = 'El valor está por debajo del límite.';
    }
    showFeedBack(this, false, message);
  } else {
    showFeedBack(this, true);
  }
});

form.vfFile.addEventListener('change', function (event) {
  const size = 10;
  if (!this.value) {
    const message = 'Adjuntar un fichero es obligatorio.';
    showFeedBack(this, false, message);
  } else if (!checkFileExtension(this.files[0], ['jpg', 'png', 'gif'])) {
    const message = 'Debe seleccionar un archivo con extensión jpg, png o
gif.';
    showFeedBack(this, false, message);
  } else if (checkFileSize(this.files[0], size)) {
    const message = `El archivo ${this.files[0].name} no debe ser mayor a
${size}KB`;
    showFeedBack(this, false, message);
  } else {
    showFeedBack(this, true, message);
  }
});
```

5. Creación de máscaras

Las máscaras son similares a los patrones ya que define el formato del valor de un elemento, pero en lugar de utilizarse como restricción del valor, las utilizamos como ayuda al usuario para cumplir con la restricción del patrón que define el elemento, modificando su valor a medida que se va cumplimentando. Se trata de un recurso de apoyo para la usabilidad de la página.

Para crear máscaras es necesario el uso de expresiones regulares, ya que nos permitirán reconocer los patrones para modificar los valores de los elementos.

Veamos algunos ejemplos. Crearemos varios eventos para implementar las máscaras, estos eventos deben **implementarse antes de la validación en línea** ya que la validación debe producirse después de aplicar la máscara.

5.1. Ejemplo con DNI

Según escribimos el DNI queremos solamente permitir introducir dígitos, y cuando lleguemos a la letra solamente una letra, evitando con ello errores. El evento `input` se dispara cada vez que introducimos un carácter en un elemento de un formulario. El evento dispone de una propiedad `data` con el dato introducido. Este evento se dispara antes de que el elemento cambie su valor, por lo que es ideal para verificar qué tipo de dato estamos introduciendo, cancelando y comportamiento en caso de no ser el valor que esperamos.

El problema del evento input es que no es cancelable. Si consultamos la propiedad `cancelable` del evento veremos que es `false`. Por esa razón, vamos a utilizar el evento `beforeinput` con la misma función, pero en este caso si que es cancelable.

En función de la longitud del valor del elemento solo permitiremos dígitos o letras. Si la propiedad `data` está asignada a `null`, indica que el carácter es no imprimible como por ejemplo el borrado, por lo que solamente verificamos el carácter.

```
form.vfDni.addEventListener('beforeinput', function (event) {
  let isValid = true;
  if (event.data) {
    if (this.value.length < 8) {
      if (!/\d/.test(event.data)) isValid = false;
    } else if (this.value.length === 8) {
      if (!/[A-Za-z]/.test(event.data)) isValid = false;
    } else {
      isValid = false;
    }
    if (!isValid) {
      event.preventDefault();
    } else {
      this.value += event.data.toUpperCase();
    }
  }
});
```

5.2. Ejemplo para el teléfono

En primer lugar, vamos a cambiar la máscara del elemento para el teléfono para que admita el número con el prefijo internacional. El formato de la máscara será el prefijo internacional entre paréntesis, agrupando los dígitos del número 3-2-2-2. Tenemos que cambiar el atributo `maxlength` para tener un máximo de 18 caracteres y el patrón debe ser `^\(\+\d\d\) \d\d\d \d\d \d\d \d\d$`.

```
<input type="text" class="form-control" id="vfPhone" name="vfPhone"
placeholder="(+34) XXX XX XX XX" value="" required maxlength="18"
pattern="^\(\+\d\d\) \d\d\d \d\d \d\d \d\d$">
```

Como caracteres válidos aplicamos que el primer carácter puede ser 0, 6, 9 o + para el prefijo. Los paréntesis y las agrupaciones se deben crear según escribimos los dígitos. El resto de los caracteres no son válidos.

Primero vamos a crear una función que, dado el valor del elemento, los transforme en la máscara utilizando expresiones regulares. En el último reemplazo utilizamos elementos

opcionales con `?`, y la extracción de patrones. Debemos tener en cuenta en no finalizar la expresión regular con `$` para reconocer los agrupamientos parciales.

```
function formatPhoneNumber(phone) {
  phone = phone.replace(/^[+|(+]/, '00');
  phone = phone.replace(/\D/g, '');
  phone = phone.replace(/^([69])/, '0034$1');
  phone = (/^00/.test(phone))
    ? phone.replace(/^00(\d\d)(\d\d\d)?(\d\d)?(\d\d)?(\d\d)/, '(+$1) $2
$3 $4 $5')
    : '';
  return phone;
}
```

Sobre el elemento vamos a crear varios eventos para generar la máscara. Añadimos el evento `beforeinput` donde aplicamos la validación de entrada de caracteres. En caso de tener un valor válido, invocamos la función de formato sobre el valor del elemento.

```
form.vfPhone.addEventListener('beforeinput', function (event) {
  let isValid = true;
  const character = event.data;
  if (character) {
    if (this.value.length === 0) {
      if (!/[690+]/.test(character)) isValid = false;
    } else if (this.value.length >= 1) {
      if (!/[\d]/.test(character)) isValid = false;
    } else {
      isValid = false;
    }
    if (!isValid) {
      event.preventDefault();
    } else {
      this.value = formatPhoneNumber(this.value);
    }
  }
});
```

Además, debemos añadir el evento `paste` y el evento `change` para formatear el valor del texto cada vez que se disparen.

```
form.vfPhone.addEventListener('paste', function (event) {
  this.value = formatPhoneNumber(this.value);
});
form.vfPhone.addEventListener('change', function () {
  this.value = formatPhoneNumber(this.value);
});
```

5.3. Ejemplo para tarjeta de crédito

Agruparemos los dígitos de la tarjeta en grupos de cuatro separándolos por espacios. Primero modificamos el elemento en la página HTML.

```
<input type="text" class="form-control" id="vfCreditCard"
name="vfCreditCard" placeholder="XXXX XXXX XXXX XXXX" value="" required
maxlength="19"
    pattern="^[0-9]{4} [0-9]{4} [0-9]{4} [0-9]{4}$">
```

Creamos una función de formato para la tarjeta de crédito basándonos en separar los agrupamientos con espacios.

```
function formatCreditCard(card) {
  card = card.replace(/\D/g, '');
  card = card.replace(/^(\d{4})?(\d{4})?(\d{4})?(\d{4})/, '$1 $2 $3 $4
');
  card = card.replace(/\s{2}/g, ' ');
  return (card.length < 19) ? card.trimStart() : card.trim();
}
```

Por último, como en el caso anterior, asignamos los manejadores de eventos para `beforeinput`, `paste` y `change`.

```
form.vfCreditCard.addEventListener('beforeinput', function (event) {
  let isValid = true;
  const character = event.data;
  if (character) {
    if (!/[\d]/.test(character)) isValid = false;
    if (!isValid) {
      event.preventDefault();
    }
    this.value = formatCreditCard(this.value);
  }
});
form.vfCreditCard.addEventListener('paste', function (event) {
  this.value = formatCreditCard(this.value);
});
form.vfCreditCard.addEventListener('change', function () {
  this.value = formatCreditCard(this.value);
});
```