Week 6 WDD330

lunes, 22 de mayo de 2023 10:24

Ruth Cardenas!

FORM PROPERTIES AND METHODS

DOCUMENT.FORM ⇒ NAMED COLECTION : IT'S BOTH NAMED & ORDERED

NAME OF THE FORM

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>

<script>
  // get the form
  let form = document.forms.my; // <form name="my"> element

  // get the element
  let elem = form.elements.one; // <input name="one"> element

  alert(elem.value); // 1
</script>
```

- FOR ELEMENTS WITH THE SAME NAME `form.elements[name]` is a *collection*.

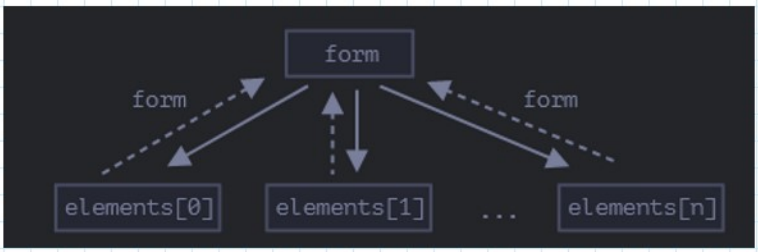
FIELDSETS AS "SUBFORMS"

```
<body>
  <form id="form">
    <fieldset name="userFields">
      <legend>info</legend>
      <input name="login" type="text">
    </fieldset>
  </form>

  <script>
    alert(form.elements.login); // <input name="login">

    let fieldset = form.elements.userFields;
    alert(fieldset); // HTMLFieldSetElement

    // we can get the input by name both from the form and from the fieldset
    alert(fieldset.elements.login == form.elements.login); // true
  </script>
</body>
```

- SHORTER NOTATION : FORM.NAME = FORM.ELEMENT.NAME

- BACKREFERENCE : A FORM REFERENCES ALL ELEMENTS, AND ELEMENTS

REFERENCE THE FORM.




```
<form id="form">
  <input type="text" name="login">
</form>

<script>
  // form -> element
  let login = form.login;

  // element -> form
  alert(login.form); // HTMLFormElement
</script>
```

- WE CAN ACCESS THE VALUE OF INPUTS AND TEXTAREAS:
  - INPUT.VALUE
  - TEXTAREA.VALUE
- <SELECT>:
  - SELECT.OPTIONS → COLLECTION OF <OPTIONS> SUBELEM.
  - SELECT.VALUE → VALUE OF CURRENTLY SELECTED OPTION.
  - SELECT.SELECTINDEX → NUMBER OF CURRENTLY SELECTED OPTION.

- MULTIPLE SELECTED VALUES:

```
// get all selected values from multi-select
let selected = Array.from(select.options)
  .filter(option => option.selected)
  .map(option => option.value);
```

- SHORT SYNTAX TO CREATE <OPTION> ELEMENT

```
option = new Option(text, value, defaultSelected, selected);
```

OPTIONAL PARAM.

IF TRUE: A SELECTED HTML-ATTR. IS CREATED

IF TRUE: THE OPTION IS SELECTED.

FORMS: EVENT AND METHOD SUBMIT

THE SUBMIT EVENT IS USUALLY USED TO VALIDATE THE FORM BEFORE SENDING

THE SUBMIT EVENT IS USUALLY USED TO VALIDATE THE FORM BEFORE SENDING IT TO THE SERVER OR TO ABORT THE SUBMISSION AND PROCESS IT IN JS.

There are two main ways to submit a form:

1. The first – to click `<input type="submit">` or `<input type="image">`.
2. The second – press `Enter` on an input field.

IF THERE ARE ERRORS CALL THE EVENT.PREVENTDEFAULT() TO AVOID THE FORM SUBMITION

METHOD: SUBMIT = FORM.SUBMIT()

IF USED = - THE SUBMIT EVENT IS NOT GENERATED
- THE SCRIPT ALREADY DID ALL RELATED PROCESS.

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// the form must be in the document to submit it
document.body.append(form);

form.submit();
```

FormData

TO SEND HTML FORMS WITH OR WITHOUT FILES

```
let formData = new FormData([form]);
```

- WITH FORMDATA, NETWORK METHODS CAN ACCEPT THE OBSECT AS A BODY
- IT'S ENCODED AND SENT OUT WITH CONTENT-TYPE: MULTIPART/FORM-DATA

EXAMPLE

```
<form id="formElem">
  <input type="text" name="name" value="John">
  <input type="text" name="surname" value="Smith">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user', {
      method: 'POST',
      body: new FormData(formElem)
```

EXAMPLE

```html
<form id="formElem">
  <input type="text" name="name" value="John">
  <input type="text" name="surname" value="Smith">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

FORMDATA METHODS:

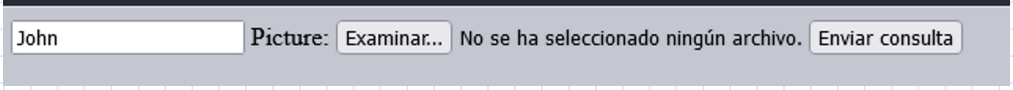
- `formData.append(name, value)` – add a form field with the given `name` and `value`,
- `formData.append(name, blob, fileName)` – add a field as if it were `<input type="file">`, the third argument `fileName` sets file name (not form field name), as it were a name of the file in user's filesystem,
- `formData.delete(name)` – remove the field with the given `name`,
- `formData.get(name)` – get the value of the field with the given `name`,
- `formData.has(name)` – if there exists a field with the given `name`, returns `true`, otherwise `false`

- FORMS ALLOW TO HAVE MANY FIELDS WITH THE SAME NAME
- SET METHOD HAS SAME SYNTAX AS APPEND
  - ↳ .SET REMOVES ALL FIELDS WITH THE GIVEN NAME, AND THEN APPENDS A NEW FIELD.
  - ↳ IT MAKES SURE THERE ONLY ONE FIELD WITH SUCH NAME, THE REST IS JUST LIKE APPEND.

SENDING A FORM WITH A FILE

THE FORM IS ALWAYS SENT AS CONTENT-TYPE: MULTIPART/FORM-DATA, THIS ENCODING ALLOWS TO SEND FILES.

```html
<form id="formElem">
  <input type="text" name="firstName" value="John">
  Picture: <input type="file" name="picture" accept="image/*">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user-avatar', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```



Sending a form with Blob data

```html
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

  <input type="button" value="Submit" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));

      let formData = new FormData();
      formData.append("firstName", "John");
      formData.append("image", imageBlob, "image.png");

      let response = await fetch('/article/formdata/post/image-form', {
        method: 'POST',
        body: formData
      });
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```

SAME AS IF THERE WHERE <INPUT TYPE="FILE" NAME="IMAGE">

CLIENT-SIDE FORM VALIDATION

- CATCHING INVALID DATA ON THE CLIENT-SIDE THAT THE USER CAN FIX IT STRAIGHT AWAY.
- VALIDATION DONE IN THE BROWSER IS CALLED CLIENT-SIDE VALIDATION.
- CLIENT-SIDE VALIDATION SHOULD NOT BE CONSIDERED AN EXHAUSTIVE SECURITY CHECKS ON ANY FORM-SUBMITTED DATA ON THE SERVER-SIDE AS WELL ON THE CLIENT-SIDE
- CLIENT-SIDE VALIDATION IS TOO EASY TO BYPASS
- IF INFO IS INCORRECT, THE BROWSER GIVES THE USER AN ERROR MESSAGE EXPLAINING WHAT NEEDS TO BE CORRECTED, AND LET THEM TRY AGAIN

WHY VALIDATION?

- WE WANT TO GET THE RIGHT DATA, IN THE RIGHT FORMAT
- WE WANT TO PROTECT OUR USERS' DATA
- WE WANT TO PROTECT OURSELVES

DIFFERENT TYPES OF CLIENT-SIDE VALIDATION

- BUIT-IN FORM VALIDATION => USES HTML FORM VALIDATION FEATURES
  - => IT DOESN'T REQUIRE MUCH JS
  - => HAS BETTER PERFORMANCE THAN JS
  - => IT'S NOT AS CUSTOMIZABLE AS JS VALIDATION

EXAMPLE =>

- `required`: Specifies whether a form field needs to be filled in before the form can be submitted.
- `minlength` and `maxlength`: Specifies the minimum and maximum length of textual data (strings).
- `min` and `max`: Specifies the minimum and maximum values of numerical input types.
- `type`: Specifies whether the data needs to be a number, an email address, or some other specific preset type.
- `pattern`: Specifies a regular expression that defines a pattern the entered data needs to follow.

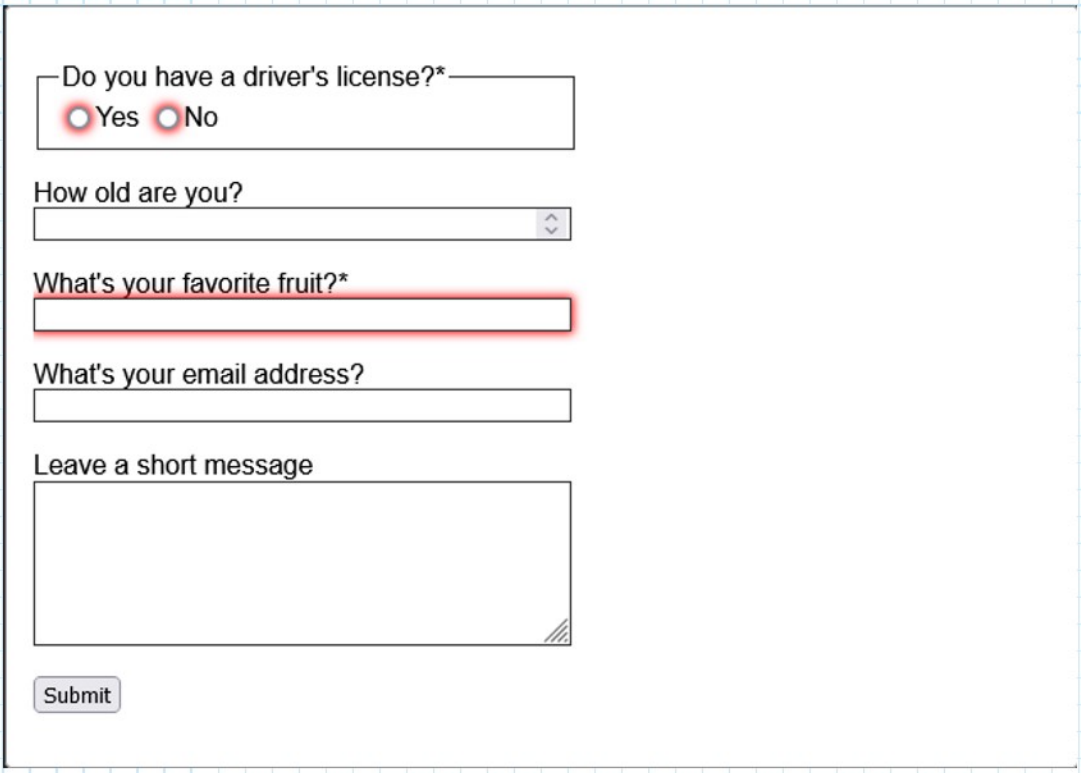
=> IF ELEMENT IS VALID:
- MATCHES THE :valid CSS PSEUDO-CLASS
- THE BROWSER WILL SUBMIT THE FORM

=> IF ELEMENT IS INVALID:
- IT MATCHES THE :invalid CSS PSEUDO-CLASS
- THE BROWSER WILL BLOCK THE FORM AND DISPLAY AN ERROR MESSAGE.

EXAMPLE

```
<form>
  <p>
    <fieldset>
      <legend>Do you have a driver's license?<span aria-label="required">*</span>
</legend>
      <!-- While only one radio button in a same-named group can be selected at a
time,
           and therefore only one radio button in a same-named group having the
"required"
           attribute suffices in making a selection a requirement -->
      <input type="radio" required name="driver" id="r1" value="yes"><label
for="r1">Yes</label>
      <input type="radio" required name="driver" id="r2" value="no"><label
for="r2">No</label>
    </fieldset>
  </p>
  <p>
    <label for="n1">How old are you?</label>
    <!-- The pattern attribute can act as a fallback for browsers which
         don't implement the number input type but support the pattern attribute.
         Please note that browsers that support the pattern attribute will make it
         fail silently when used with a number field.
         Its usage here acts only as a fallback -->
    <input type="number" min="12" max="120" step="1" id="n1" name="age"
           pattern="\d+">
  </p>
  <p>
    <label for="t1">What's your favorite fruit?<span aria-label="required">*</span>
</label>
    <input type="text" id="t1" name="fruit" list="l1" required
       pattern="[Bb]anana|[Cc]herry|[Aa]pple|[Ss]trawberry|[Ll]emon|[Oo]range">
    <datalist id="l1">
      <option>Banana</option>
      <option>Cherry</option>
      <option>Apple</option>
      <option>Strawberry</option>
      <option>Lemon</option>
      <option>Orange</option>
    </datalist>
```

```
      <option>Orange</option>
    </datalist>
  </p>
  <p>
    <label for="t2">What's your email address?</label>
    <input type="email" id="t2" name="email">
  </p>
  <p>
    <label for="t3">Leave a short message</label>
    <textarea id="t3" name="msg" maxlength="140" rows="5"></textarea>
  </p>
  <p>
    <button>Submit</button>
  </p>
</form>
```



- JAVASCRIPT VALIDATION => THIS VALIDATION IS COMPLETELY CUSTOMIZABLE
  => YOU NEED TO CREATE ALL (IN MANY WAYS)
  => THE CONSTRAINT VALIDATION API:

- `HTMLButtonElement` (represents a `<button>` element)
- `HTMLFieldSetElement` (represents a `<fieldset>` element)
- `HTMLInputElement` (represents an `<input>` element)
- `HTMLOutputElement` (represents an `<output>` element)
- `HTMLSelectElement` (represents a `<select>` element)
- `HTMLTextAreaElement` (represents a `<textarea>` element)

• THIS API HAS MANY PROPERTIES AVAILABLE, SUCH AS: VALIDATION MESSAGE AND VALIDITY (STATE)

• METHODS AVAILABLES

Guides > Client-side form validation

- `checkValidity()`: Returns `true` if the element's value has no validity problems; `false` otherwise. If the element is invalid, this method also fires an `invalid` event on the element.
- `reportValidity()`: Reports invalid field(s) using events. This method is useful in combination with `preventDefault()` in an `onSubmit` event handler.
- `setCustomValidity(message)`: Adds a custom error message to the element; if you set a custom error message, the element is considered to be invalid, and the specified error is displayed. This lets you use JavaScript code to establish a validation failure other than those offered by the standard HTML validation constraints. The message is shown to the user when reporting the problem.

=> YOU WILL HAVE TO CUSTOMIZED AN ERROR MESSAGE

EXAMPLE:

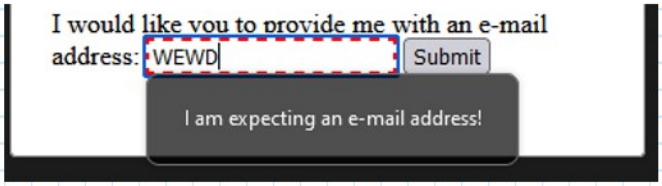
HTML

```
<form>
  <label for="mail">
    I would like you to provide me with an
email address:
  </label>
  <input type="email" id="mail" name="mail"
/>
  <button>Submit</button>
```

RESULT:

I would like you to provide me with an e-mail address: WEWD Submit

```
  <input type="email" id="mail" name="mail"
/>
  <button>Submit</button>
</form>
```



JS

```
const email =
document.getElementById("mail");

email.addEventListener("input", (event) => {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("I am expecting
an email address!");
  } else {
    email.setCustomValidity("");
  }
});
```

WHEN VALIDATING INFORMATION:

- MAKE SURE TO THINK CAREFULLY ABOUT THE USER
- HELP USERS CORRECT THE DATA THEY PROVIDE
- DISPLAY EXPLICIT ERROR MESSAGES
- BE PERMISSIVE ABOUT THE INPUT FORMAT
- POINTS OUT EXACTLY WHERE THE ERROR OCCURS, ESPECIALLY ON LARGE FORMS.