Form Validation with JavaScript

Programming in HTML5 with JavaScript and CSS3 (Unit 635)

Setup

In this exercise, we are going to see how we can define custom validation behaviour with JavaScript.

Download `form-validation.zip` from Moodle. Inside the zip are 3 files which you should extract to a new folder in your websites folder. You should then open all 3 files in your editor.

form.html

This is a simple HTML page containing a form, similar to the forms we have seen previously.

Of interest to us:

- The `<form>` has a boolean attribute: `novalidate`. This tells the browser not to validate the fields in the form.
- Each field is followed by a `<div>` which has the class `field-error`. We will be putting our custom error messages in these.
- The fields still have validation-related attributes. i.e. Both fields have the `required` attribute and the email field has its `type` set to email. While we are not using the browser's built-in validation, we will see how we can utilise these attributes ourselves from our JavaScript.

form.css

This is mostly the same as the previous CSS we have seen used with forms, with the addition of a new rule targeting `.field-error`. This is for our custom error messages, as mentioned above.

form.js

In this file, you have been provided with an outline for the code we will write.

At the top of the file, several constants representing the DOM elements we will be using have been defined. The form and the 2 inputs are selected via their IDs, while the 2 "Error" elements use a special CSS selector (`+`):

```
#fullName + .field-error
```

The `+` is known as the Adjacent sibling combinator which allows us to select elements which come immediately after other elements in the page. In this case, the selector says: *An element with the class "field-error" that comes immediately after an element with an ID equal to "fullName"*

You will also see the outline for a number of *functions*. Inside these functions is where you will be adding your code.

At the bottom of the file, some event listeners have been setup. For the name and email fields, we are listening for the *input* event. For the form itself, we are listening for its *submit* event.

- The *input* event is fired for an element when the user interacts with that element (e.g. by typing in it). Note, this event fires every time the user presses a key while the cursor is in the field.
- The *submit* event is fired when the user clicks on the form's button, or attempts to submit the form by other means.

Goals

We are going to try to emulate the behaviour that occurs with built-in validation, while having more control over how the errors are displayed to the user.

What we want to achieve:

- If the user types an invalid value in a field, we want to display an error message immediately.
- When the user clicks the submit button, we want to check all of the fields and display error messages for them if necessary.

The full name field

For the `fullName` field, we want to make sure that the user submits a value. We don't care how many characters or what format their name is. For this reason, we have just given it the `required` attribute in the HTML. As mentioned, the browser is ignoring this attribute, however, we can still utilise it in our code.

When the user types in the `fullName` field, the `validateName()` function is executed. Inside this function we need to check to see if the data in the field is valid. If it is, we will remove any pre-existing error messages. If it is not, we will display an error message. The `showNameError()` function will be responsible for displaying the error message.

Update your `validateName()` function to look like this:

```
function validateName() {
    if (nameInput.validity.valid) {
        // Value is valid, so remove any previous error message
        nameError.textContent = '';
    } else {
        // Value is not valid, so display an error message
        showNameError();
    }
}
```

In the conditional (`if`), we are checking to see if `nameInput.validity.valid` is `true`. If it is, we are setting the corresponding error element's content to an empty string. If it is not, we are calling the `showNameError()` function.

The `validity` property of the DOM element gives us access to the same information the browser uses to display its built-in messages. The `valid` property of the `validity` object checks to see if the data entered in the field conforms to the validation attributes defined in the HTML.

In this case, the HTML for the field has the `required` attribute, so if there is at least one character in the field, `validity.valid` will be `true`. If the field is empty, it will be `false`.

Update your `showNameError()` function to look like this:

```
function showNameError() {
    nameError.textContent = 'Please enter your name';
}
```

All that we are doing here is updating the content of the error element to be our custom error message.

Save your files and open `form.html` in your browser. By default, no errors are shown. When you type in the full name field, no errors are shown. But if you type something in the field and then delete all of the characters, you should see an error message appear when you delete the final character.

The email field

For the email field, the code will be very similar, however, there are 2 constraints on the field's data:

- The field is `required` in the HTML, so it must have a value (same as the full name field)
- The field's `type` is set to `email` in the HTML, so the value must be a valid email address.

The `validateEmail()` function will be almost identical to the `validateName()` function:

```
function validateEmail() {
    if (emailInput.validity.valid) {
        // Value is valid, so remove any previous error message
        emailError.textContent = '';
    } else {
        // Value is not valid, so display an error message
        showEmailError();
    }
}
```

In this case, because the HTML for the email field has 2 constraints, the value of `validity.valid` will be `true`, only if **both** constraints have been met. i.e. There is a value, and the value is a valid email.

The `showEmailError()` function will also be similar to the `showNameError()` function, but we need to modify it slightly to cater for the 2 different error states and display messages suitable for each of them:

```
function showEmailError() {
    if (emailInput.validity.valueMissing) {
        // The field is empty
        emailError.textContent = 'Please enter your email address';
    } else if (emailInput.validity.typeMismatch) {
        // The field contains an invalid email address
        emailError.textContent = 'Your email address does not appear to be correct';
    }
}
```

Here we can see some more properties from the `validity` object being used.

E.g. `emailInput.validity.valueMissing`

For each of the constraints that can be applied in the HTML, there is a corresponding property on the `validity` object which can tell us if that constraint has been broken or not. The `valueMissing` property is mapped to the `required` attribute, while the `typeMismatch` property is mapped to the element's `type` attribute.

For example, if the user enters an invalid email in the field:

- `emailInput.validity.valueMissing` will equate to `false` (because there is a value)
- `emailInput.validity.typeMismatch` will equate to `true` (because the format does not match the expected format for that input *type*)

By using these in the `if/else if` statement, we can tailor the messages to match the error the user has made.

Save your files and reload the page in your browser. As you start typing in the email field, you should see the `typeMismatch` error message appear. If you type in a valid email address, the message will disappear. If you delete all of the characters in the field, the `valueMissing` error message should appear.

The form

What you may have noticed by now, is that we are only validating fields when the user types in them. While we are displaying error messages for the user, they can still submit the form with nonsense data!

This is where the event listener attached to the form and the `validateForm()` function make their appearance.

Remember, the *submit* event occurs when the user attempts to submit the form, normally by clicking on the button. We have specified that the `validateForm()` function will be executed when this event fires.

You will notice that the `validateForm()` function has defined `event` as its argument. If you recall, **all** event listeners are passed the `event` object when they are invoked, but in many cases we don't need to use it, so we don't define it as an argument (NB: JavaScript is one of the few languages where you can be loose like this!).

What we want to do in this function is:

- Check **all** the fields to see if they are valid
- If all of them are valid
  - we do nothing and let the browser submit the form as normal
- If *any* of them are not valid
  - We display the error messages for each field that is invalid
  - We prevent the browser from submitting the form

Thankfully, we can re-use some of our functions to achieve this.

Update your `validateForm()` function to look like this:

```
function validateForm(event) {
    // Keep track of our validation "state"
    let formHasErrors = false;

    if (!nameInput.validity.valid) {
        // Name input is invalid, show its error and change "state"
        formHasErrors = true;
        showNameError();
    }

    if (!emailInput.validity.valid) {
        // Email input is invalid, show its error and change "state"
        formHasErrors = true;
        showEmailError();
    }

    // Check the "state" to see if the form should be submitted
    if (formHasErrors) {
        event.preventDefault();
    }
}
```

Let's break down the code...

- `let formHasErrors = false;`
  - We are validating 2 fields, and need to know if **any** of them are invalid. This variable allows us to keep track of things. When it is defined, it is set to `false` (i.e. no errors). As we validate the fields, if we find one to be invalid, we change the value of `formHasErrors` to `true`. When we get to the end of the function, we can check the value of this variable to see if any errors were detected.
- `if (!nameInput.validity.valid) {}`, `if (!emailInput.validity.valid) {}`
  - Both of these conditional statements are similar to those seen in `validateName()` and `validateEmail()`. This time, we are checking to see if the `validity.valid` property is `false`. If it is, this means the field is invalid, so we change the `formHasErrors` variable to `true`, and then call the appropriate error display function for that field.
- `if (formHasErrors) {}`
  - The final conditional statement is checking the value of `formHasErrors`. If that variable is `true`, we know that *something* is invalid, so we shouldn't let the browser submit the form. We do this by calling the `preventDefault()` method which is a member of the `event` object we received as an argument. This method can be used in many scenarios where you don't want the browser's default behaviour to be invoked (it is a bit like a "cancel" button for browser functions!).

Save your files and reload the form page in your browser. Try submitting the form with no data in either field. You should see 2 error messages. Try submitting with one valid field and one invalid. You should see one error message. Try submitting the form with valid data. The form should submit as normal.

References

This exercise is based on an excerpt from the MDN Client-side form validation tutorial. Some changes were made to the code so that it would be more readable and would support more than one field in the form.