Forms:

<form> contains form controls such as input fields, menus, and buttons.

<!doctype html><html lang='en'><head> <meta charset='utf-8'> <title>Search</title></head><body> <form name='search' action='/search'> <input name='searchInput'> <button type='submit'>Search</button> </form><script src='main.js'></script></body></html>

Reset buttons are generally considered poor for usability, as they are too easy to click and then wipe out all the data that’s been entered.

<button type='reset'>Reset</button>

The focus event occurs when an element is focused on. In the case of an <input> element, this is when the cursor is placed inside the element (either by clicking or tapping on it or navigating to it using the keyboard).

*const* input = form.elements.searchInput;

input.addEventListener('focus', () => alert('focused'), false);

The blur event occurs when the user moves the focus away from the form element.

input.addEventListener('blur', () => alert('blurred'), false);

Possibly the most important form event is the submit event, occurring when the form is submitted. Usually this will send the content of the form to the server to be processed, but we can use JavaScript to intercept the form before it’s sent by adding a submit event listener.

*const* form = document.forms['search'];form.addEventListener ('submit', search, false);

*function* search() { alert(' Form Submitted');}

Text input element objects have a value property that can be used to retrieve the text inside the field.

We can use this to report back what the user has searched for.

*function* search(event) { alert(`You Searched for: ${input.value}`); event.preventDefault();}

Using the placeholder attribute in the HTML markup.

<input type='text' name='search-box' placeholder='Search Here'>

This has slightly different behavior in that the placeholder text is not actually a value of the input field, so it won’t be submitted as the field’s value if the user fails to fill it in.

The maxlength attribute will limit the number of characters that can be entered in the field to the value given (in this case 32).

input type='password' is used to enter passwords or secret information. This works in the same way as an input field with type='text', except the characters are concealed as they are entered so they’re unable to be read on the screen.

<label *for*='realName'>Real Name:<input type='password' name='realName' id='realName'></label>

To process this information, we add the following line to the makeHero()function in main.js:

hero.realName = form.realName.value;

As you can see, values from a password input field are accessed in the same way as text input fields using the value property.

Check boxes are created using input fields with type='checkbox'. They are used to select different options that can be checked (true) or left unchecked (false). The user can select more than one checkbox from a list.

<p>Super Powers:</p><label *for*='flight'>Flight: <input type='checkbox' id='flight' value='Flight' name='powers'></label><label *for*='strength'>Super Strength: <input type='checkbox' id='strength' value='Strength' name='powers'></label><label *for*='speed'>Super Speed: <input type='checkbox' id='speed' value='Super Speed' name='powers'></label><label *for*='energy'>Energy Blasts: <input type='checkbox' id='energy' value='Energy Blasts' name='powers'> </label><label *for*='telekinesis'>Telekinesis: <input type='checkbox' id='telekinesis' value='Telekinesis' name='powers'></label>

Notice that all the checkbox elements have the same 'name' property of 'powers'. This means they can be accessed as an HTML collection, like so:

form.powers;

We can then iterate over this collection using a for loop to see if each checkbox was checked. Checkbox objects have a checked property that tells us if it has been checked or not. It is a boolean property, so can only have the values true or false. The value property is used to set the name of the power that can be used if the checkbox has been checked.

hero.powers = [];*for* (*let* i=0; i < form.powers.length; i++) { *if* (form.powers[i].checked) { hero.powers.push(form.powers[i].value); }}

We can refactor this code to be much more succinct by using the array iterators we saw in Chapter 4. The following code will achieve the same result:

hero.powers = [...form.powers].filter(box => box.checked).map(box => box.value);

Radio buttons are created using input fields with type='radio'. Like checkboxes they allow users to check an option as true, but they provide an exclusive choice of options, so only one option can be selected.

This type of mutually exclusive option could be whether a superhero is a hero or a villain... or even an antihero (you know, those who are unable to decide whether to be good or bad!).

<p>What type *of* hero are you?</p><label *for*='hero'>Hero: <input type='radio' name='category' value='Hero' id='hero'></label><label *for*='villain'>Villain: <input type='radio' name='category' value='Villain' id='villain'></label><label *for*='anti-hero'>Anti-Hero: <input type='radio' name='category' value='Antihero' id='anti-hero'></label>

All these radio buttons have the same 'name' attribute of 'category'. This is used to group them together only one radio button can be checked in a group that has the same name attribute. It also means we can access an HTML collection of all the radio buttons in that group using the property of the same name as can be seen in this line of code:

form.category;

Hidden fields can be created using input fields with type='hidden'. These are not displayed by the browser, but have a 'value' attribute that can contain information that is submitted with the form. They are often used to send information such as settings or information that the user has already provided. Note that the information in these fields is in no way secret, as it’s visible in the HTML, so shouldn’t be used for sensitive data.

A file input field can be created using input fields with type='file'. These are used to upload files, and most browsers will provide a browse button or similar that lets users select a file from their file system.

Number input fields also have optional 'min' and 'max' attributes that can be used to limit the input given. The 'step' attribute is used to specify how much the value changes by each click. Most modern browsers will add controls at the side of the input field so the value can be increased or decreased.

These new input types are yet to be supported, but the good news is that you can start using them now because they will still work; the browser will just display a normal text input field if it doesn’t support a particular type.

Select drop-down lists can be used to select one or more options from a list of values. The 'multiple' attribute is required if more than one option is to be selected.

<label *for*='City'>Base *of* Operations: <select name='city' id='city'> <option value='' selected>Choose a City</option> <option value='Metropolis'>Metropolis</option> <option value='Gotham City'>Gotham City</option> <option value='Keystone City'>Keystone City</option> <option value='Coast City'>Coast City</option> <option value='Star City'>Star City</option> </select></label>

Note that the 'selected' attribute can be used to set the initial value in the HTML. In this example, the blank option that provides the instructional message 'Choose a City' has this attribute, so it’s shown when the page loads.

If only one item was selected, this will return a reference to that selection; otherwise a collection will be returned containing each selection.

Each selection object has a value property that’s equal to the 'value' attribute of the <option> tag that was selected.

hero.city = form.city.value;

You can access the text of any option using index notation. For example, the following code returns the text from the first option:

form.city.options[0].text<< "Choose a City"

There is also a type of 'menu' that can be combined with <menu>, <menuitem> and <li> tags to create a dropdown menu when it’s clicked on, although support for this is fairly patchy at present.

Validation can occur on the client side using JavaScript, and on the server side. It is advisable to use both client-side and server-side validation. JavaScript should not be relied upon to validate any data before it’s saved to a database. This is because it’s possible for a user to modify the JavaScript code and bypass the validation rules. It's also very easy to bypass the front-end completely and send arbitrary data to the application's backend. For these reasons, JavaScript validation should be used to enhance the user experience when filling in a form by giving feedback about any errors before it’s submitted. This should then be backed up with more validation performed on the server before the data is eventually saved to a database. Having said that, it’s still useful to validate on the client side even if the data will be validated again on the server side. This is because it will ensure that more valid data is sent to the server, which helps to cut down the number of HTTP requests required to send the form back and forward from the server to be corrected.

HTML5 has its own validation API that can be used, although it lacks the full support from all browsers at the moment. The error messages that it produces can look inconsistent across browsers and are difficult to style.

The API works by simply adding relevant attributes to the form fields. For example, if a field is a required field that must be filled in, all you need to do is add a 'required' attribute to that field and the browser will take care of the rest.

<input type='text' id='heroName' name='heroName' autofocus placeholder='Your Super Hero Name' maxlength=32 required>

It is also possible to implement custom form validation using JavaScript. For example, say we wanted to exclude any superhero names that begin with an 'X'. This is not a standard form of validation, so we’d have to write our own.

form.addEventListener('submit',validate,false);

*function* validate(event) { *const* firstLetter = form.heroName.value[0]; *if* (firstLetter.toUpperCase() === 'X') { event.preventDefault(); alert('Your name is not allowed to start with X!'); }}

We can improve the usability of the form further by giving instant feedback, instead of waiting for the form to be submitted. This can be achieved by adding the event listener directly to the input field that will fire when the user presses a key (using the keyup event). The feedback can then be inserted inside the label element of the input field, along with a class of error for more direct feedback.

*const* label = form.querySelector('label');*const* error = document.createElement('div');error.classList.add('error');error.textContent = '! Your name is not allowed to start with X.';label.append(error);

*function* validateInline() { *const* heroName = *this*.value.toUpperCase(); *if*(heroName.startsWith('X')){ error.style.display = 'block'; } *else* { error.style.display = 'none'; }}

Another useful technique that can aid usability is to disable the submit button if there are errors on the form. If the submit button is disabled then no action is taken when it’s clicked. Most browsers will also display it in a lighter color to indicate that it cannot be clicked on. This prevents users from submitting a form containing any errors.

A submit button can be disable by added the disabled attribute to the <input> element:

<button type='submit' id='submit' disabled>Submit</button>

This can be changed programmatically using the disabled property of the <button> element. The following function will disable the button if an input field is empty:

*function* disableSubmit(event) { *if*(event.target.value === ''){ document.getElementById('submit').disabled = true; } *else* { document.getElementById('submit').disabled = false; }}