Week 10 Notes

Client-side form validation

Ensure all required form controls are filled out in the correct format.

Should not be considered an exhaustive security measure!

What is form validation?

When you enter data, the browser and/or web server checks to see if the data is in the correct format.

Both client-side and server-side

If the info is formatted correctly, the data is submitted to the server and usually saved in a database;

Why validate?

To get the right data
To protect the users' data
To protect ourselves

Warning:: Never trust data passed to your server from the client. Even if your form is validating correctly and preventing malformed input on the client-side, a malicious user can still alter the network request.

Different types of client-side validation

Built in form validation:

Uses HTML5 form validation features. Not much JavaScript, but not as customizable.

JavaScript:

Uses JavaScript and is completely customizable. Create it or use a library.

Using built-in form validation

Uses validation attributes on form elements:

required
minlength and **maxlength**
min and **max**
pattern

If data entered into a form follows the rules specified, it is valid. If not, its invalid. Uses **required** attribute.

When valid: Element matches the **:valid** pseudo-class allowing us to style. If the user tries to send data, the browser will submit the form, if there is nothing else stopping it (like JavaScript).

When invalid: Element matches the **:invalid** pseudo-class allowing us to style. If the user tries to send data, the browser will block the form and display an error message.

Regular Expressions – uses **pattern** attribute.

a — Matches one character that is a (not b, not aa, and so on).

abc — Matches a, followed by b, followed by c.

ab?c—Matches a, optionally followed by a single b, followed by c. (ac or abc)

ab*c—Matches a, optionally followed by any number of bs, followed by c. (ac , abc, abbbbbc, and so on).

a|b — Matches one character that is a or b.

abc|xyz — Matches exactly abc or exactly xyz (but not abcxyz or a or y, and so on).

Example:

```
<form>

  <p>

    <fieldset>

      <legend>Do you have a driver's license?<abbr title="This field is mandatory"
aria-label="required">*</abbr></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?<abbr title="This field is mandatory"
aria-label="required">*</abbr></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>
  </p>
  <p>
    <label for="t2">What's your e-mail 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>
```

Copy to Clipboard
And now some CSS to style the HTML:

```
form {
  font: 1em sans-serif;
  max-width: 320px;
```

```
}


p > label {
  display: block;
}


input[type="text"],
input[type="email"],
input[type="number"],
textarea,
fieldset {
  width : 100%;
  border: 1px solid #333;
  box-sizing: border-box;
}


input:invalid {
  box-shadow: 0 0 5px 1px red;
}


input:focus:invalid {
  box-shadow: none;
```

```
}
```

Validating Forms Using Javascript

Constraint Validation API

Most browsers support, a set of methods and properties available on the following form element DOM interfaces.

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)

Properties available on above elements:

validationMessage: returns a localized message describing the validation constraints that the control doesn't satisfy (if any)

validity: returns a ValidityState object that has several properties describing the validity state of the element:

patternMismatch, tooLong, tooShort, rangeOverflow, rangeUnderflow, typeMismatch, valid, valueMissing

willValidate: true if element will be validated on submition.

Following methods are available on elements and form element:

checkValidity(), reportValidity(), setCustomValidity(message)

Implementing a customized error message

Two drawbacks:

-No standard way to change the look and feel with CSS

-They depend on the browser locale, so the page can be in one language, but the error message in another.

Example:

```
<form novalidate>

  <p>

    <label for="mail">

      <span>Please enter an email address:</span>

      <input type="email" id="mail" name="mail" required minlength="8">

      <span class="error" aria-live="polite"></span>

    </label>

  </p>

  <button>Submit</button>

</form>
```

```
body {

  font: 1em sans-serif;

  width: 200px;

  padding: 0;

  margin : 0 auto;

}



p * {

  display: block;

}
```

```
input[type=email]{
  -webkit-appearance: none;
  appearance: none;

  width: 100%;
  border: 1px solid #333;
  margin: 0;

  font-family: inherit;
  font-size: 90%;

  box-sizing: border-box;
}

/* This is our style for the invalid fields */
input:invalid{
  border-color: #900;
  background-color: #FDD;
}

input:focus:invalid {
  outline: none;
```

```
}


/* This is the style of our error messages */
.error {
  width  : 100%;
  padding: 0;

  font-size: 80%;
  color: white;
  background-color: #900;
  border-radius: 0 0 5px 5px;

  box-sizing: border-box;
}

.error.active {
  padding: 0.3em;
}
// There are many ways to pick a DOM node; here we get the form itself and the email
// input box, as well as the span element into which we will place the error message.
const form  = document.getElementsByTagName('form')[0];
```

```
const email = document.getElementById('mail');
const emailError = document.querySelector('#mail + span.error');

email.addEventListener('input', function (event) {
  // Each time the user types something, we check if the
  // form fields are valid.

  if (email.validity.valid) {
    // In case there is an error message visible, if the field
    // is valid, we remove the error message.
    emailError.textContent = ''; // Reset the content of the message
    emailError.className = 'error'; // Reset the visual state of the message
  } else {
    // If there is still an error, show the correct error
    showError();
  }
});

form.addEventListener('submit', function (event) {
  // if the email field is valid, we let the form submit

  if(!email.validity.valid) {
```

```
    // If it isn't, we display an appropriate error message
    showError();
    // Then we prevent the form from being sent by canceling the event
    event.preventDefault();
  }
});


function showError() {
  if(email.validity.valueMissing) {
    // If the field is empty,
    // display the following error message.
    emailError.textContent = 'You need to enter an e-mail address.';
  } else if(email.validity.typeMismatch) {
    // If the field doesn't contain an email address,
    // display the following error message.
    emailError.textContent = 'Entered value needs to be an e-mail address.';
  } else if(email.validity.tooShort) {
    // If the data is too short,
    // display the following error message.
    emailError.textContent = `Email should be at least ${ email.minLength }
characters; you entered ${ email.value.length }.`;
  }
```

```
  // Set the styling appropriately

  emailError.className = 'error active';

}
```

Example not using a built-in API

```
<form>

  <p>

    <label for="mail">

        <span>Please enter an email address:</span>

        <input type="text" id="mail" name="mail">

        <span class="error" aria-live="polite"></span>

    </label>

  </p>

  <!-- Some legacy browsers need to have the `type` attribute

       explicitly set to `submit` on the `button`element -->

  <button type="submit">Submit</button>

</form>

body {

  font: 1em sans-serif;

  width: 200px;

  padding: 0;

  margin : 0 auto;
```

```
}

form {
  max-width: 200px;
}

p * {
  display: block;
}

input.mail {
  -webkit-appearance: none;

  width: 100%;
  border: 1px solid #333;
  margin: 0;

  font-family: inherit;
  font-size: 90%;

  box-sizing: border-box;
}
```

```
/* This is our style for the invalid fields */
input.invalid{
  border-color: #900;
  background-color: #FDD;
}

input:focus.invalid {
  outline: none;
}

/* This is the style of our error messages */
.error {
  width  : 100%;
  padding: 0;

  font-size: 80%;
  color: white;
  background-color: #900;
  border-radius: 0 0 5px 5px;
  box-sizing: border-box;
}
```

```
.error.active {
  padding: 0.3em;
}
// There are fewer ways to pick a DOM node with legacy browsers
const form  = document.getElementsByTagName('form')[0];
const email = document.getElementById('mail');

// The following is a trick to reach the next sibling Element node in the DOM
// This is dangerous because you can easily build an infinite loop.
// In modern browsers, you should prefer using element.nextElementSibling
let error = email;
while ((error = error.nextSibling).nodeType != 1);

// As per the HTML5 Specification
const emailRegExp = /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-
]+)*$/;

// Many legacy browsers do not support the addEventListener method.
// Here is a simple way to handle this; it's far from the only one.
function addEvent(element, event, callback) {
  let previousEventCallBack = element["on"+event];
  element["on"+event] = function (e) {
```

```
    const output = callback(e);


    // A callback that returns `false` stops the callback chain
    // and interrupts the execution of the event callback.
    if (output === false) return false;


    if (typeof previousEventCallBack === 'function') {
      output = previousEventCallBack(e);
      if(output === false) return false;
    }
  }
};


// Now we can rebuild our validation constraint
// Because we do not rely on CSS pseudo-class, we have to
// explicitly set the valid/invalid class on our email field
addEvent(window, "load", function () {
  // Here, we test if the field is empty (remember, the field is not required)
  // If it is not, we check if its content is a well-formed e-mail address.
  const test = email.value.length === 0 || emailRegExp.test(email.value);


  email.className = test ? "valid" : "invalid";
```

```
});


// This defines what happens when the user types in the field
addEvent(email, "input", function () {
  const test = email.value.length === 0 || emailRegExp.test(email.value);
  if (test) {
    email.className = "valid";
    error.textContent = "";
    error.className = "error";
  } else {
    email.className = "invalid";
  }
});


// This defines what happens when the user tries to submit the data
addEvent(form, "submit", function () {
  const test = email.value.length === 0 || emailRegExp.test(email.value);


  if (!test) {
    email.className = "invalid";
    error.textContent = "I expect an e-mail, darling!";
    error.className = "error active";
```

```
    // Some legacy browsers do not support the event.preventDefault() method
    return false;
  } else {
    email.className = "valid";
    error.textContent = "";
    error.className = "error";
  }
});
```

Using Fetch

Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline like requests and responses. Provides global fetch() method to give easy, logical way to fetch resources asynchronously across the network.

Basic fetch request:

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

This fetches a JSON file across the network and prints it to the console.

To extract the JSON body content from the Response object, we use json().

Supplying request options

fetch90 method can accept a second parameter: **init** object that allows you to control a number of different settings.

Example:

```
// Example POST method implementation:

async function postData(url = '', data = {}) {

  // Default options are marked with *

  const response = await fetch(url, {

    method: 'POST', // *GET, POST, PUT, DELETE, etc.

    mode: 'cors', // no-cors, *cors, same-origin

    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached

    credentials: 'same-origin', // include, *same-origin, omit

    headers: {

      'Content-Type': 'application/json'

      // 'Content-Type': 'application/x-www-form-urlencoded',

    },

    redirect: 'follow', // manual, *follow, error

    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade,
origin, origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-
cross-origin, unsafe-url

    body: JSON.stringify(data) // body data type must match "Content-Type" header

  });

  return response.json(); // parses JSON response into native JavaScript objects

}



postData('https://example.com/answer', { answer: 42 })
```

```
  .then(data => {
    console.log(data); // JSON data parsed by `data.json()` call
  });
```

Sending a request with credentials included

add **credentials: 'include** to the **init** object you pass to the fetch() method.

Example:

```
fetch('https://example.com', {
  credentials: 'include'
});
```

Uploading JSON data

Use fetch() to POST JSON-encoded data

Example:

```
const data = { username: 'example' };

fetch('https://example.com/profile', {
  method: 'POST', // or 'PUT'
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data),
})
.then(response => response.json())
.then(data => {
```

```
  console.log('Success:', data);
})
.catch((error) => {
  console.error('Error:', error);
});
```

Uploading a file

files uploaded using HTML <input type="file" multiple /> input element, FormData() and fetch().

const formData = new FormData()

```
const formData = new FormData();
const fileField = document.querySelector('input[type="file"]');

formData.append('username', 'abc123');
formData.append('avatar', fileField.files[0]);

fetch('https://example.com/profile/avatar', {
  method: 'PUT',
  body: formData
})
.then(response => response.json())
.then(result => {
  console.log('Success:', result);
```

```
})

.catch(error => {

  console.error('Error:', error);

});
```

Uploading multiple files

Files can be uploaded using an HTML <input type="file" multiple /> input element, FormData() and fetch().

```
const formData = new FormData();

const photos = document.querySelector('input[type="file"][multiple]');



formData.append('title', 'My Vegas Vacation');

for (let i = 0; i < photos.files.length; i++) {

  formData.append(`photos_${i}`, photos.files[i]);

}



fetch('https://example.com/posts', {

  method: 'POST',

  body: formData,

})

.then(response => response.json())

.then(result => {

  console.log('Success:', result);

})
```

```
.catch(error => {

  console.error('Error:', error);

});
```

Process Text line by line

Chunks read from response are **Uint8Arrays, not strings.**

Example using a line iterator:

```
async function* makeTextFileLineIterator(fileURL) {

  const utf8Decoder = new TextDecoder('utf-8');

  const response = await fetch(fileURL);

  const reader = response.body.getReader();

  let { value: chunk, done: readerDone } = await reader.read();

  chunk = chunk ? utf8Decoder.decode(chunk) : '';



  const re = /\n|\r|\r\n/gm;

  let startIndex = 0;

  let result;



  for (;;) {

    let result = re.exec(chunk);

    if (!result) {

      if (readerDone) {
```

```
        break;
      }
      let remainder = chunk.substr(startIndex);
      ({ value: chunk, done: readerDone } = await reader.read());
      chunk = remainder + (chunk ? utf8Decoder.decode(chunk) : '');
      startIndex = re.lastIndex = 0;
      continue;
    }
    yield chunk.substring(startIndex, result.index);
    startIndex = re.lastIndex;
  }
  if (startIndex < chunk.length) {
    // last line didn't end in a newline char
    yield chunk.substr(startIndex);
  }
}

async function run() {
  for await (let line of makeTextFileLineIterator(urlOfFile)) {
    processLine(line);
  }
}
```

```
run();
```

Was Request Successful?

```
fetch('flowers.jpg')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not OK');
    }
    return response.blob();
  })
  .then(myBlob => {
    myImage.src = URL.createObjectURL(myBlob);
  })
  .catch(error => {
    console.error('There has been a problem with your fetch operation:', error);
  });
```

Supplying your own request Object

Instead of passing a path to the resource you want to request, you can create a request object using **Request()** constructor and pass it in as a fetch() method argument:

```
const myHeaders = new Headers();
```

```
const myRequest = new Request('flowers.jpg', {
  method: 'GET',
  headers: myHeaders,
  mode: 'cors',
  cache: 'default',
});

fetch(myRequest)
  .then(response => response.blob())
  .then(myBlob => {
    myImage.src = URL.createObjectURL(myBlob);
  });
```

Headers

The Headers interface allows you to create a headers object with the Headers() constructor.

```
const content = 'Hello World';

const myHeaders = new Headers();

myHeaders.append('Content-Type', 'text/plain');

myHeaders.append('Content-Length', content.length.toString());

myHeaders.append('X-Custom-Header', 'ProcessThisImmediately');
```

Guard

Header objects have a guard property.

Possible guard values are:

none: default.

request: guard for a headers object obtained from a request (Request.headers).

request-no-cors: guard for a headers object obtained from a request created with Request.mode no-cors.

response: guard for a headers object obtained from a response (Response.headers).

immutable: guard that renders a headers object read-only; mostly used for ServiceWorkers.

Response objects

Response objects are returned when fetch() promises are resolved.

Most common response properties:

Response.status — An integer (default value 200) containing the response status code.

Response.statusText — A string (default value ""), which corresponds to the HTTP status code message. Note that HTTP/2 does not support status messages.

Response.ok — seen in use above, this is a shorthand for checking that status is in the range 200-299 inclusive. This returns a boolean value.

Body

Requests and reponses contain body data. A body is an instance of the following types:

ArrayBuffer

ArrayBufferView (Uint8Array and friends)

Blob/File

string

URLSearchParams

FormData

Request bodies can be set by passing body parameters:

```
const form = new FormData(document.getElementById('login-form'));

fetch('/login', {

  method: 'POST',

  body: form

});
```