Client-side JavaScript

Event Handling
AJAX

Tecnologie Internet

a.a. 2022/2023

Event Handling

JavaScript events

HTML events are **"things"** that happen to HTML elements.
When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

JavaScript allows for the execution of specific code when events are detected.

JavaScript events

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes:

```
<some-HTML-element some-event='some JavaScript'>
```

With double quotes:

```
<some-HTML-element some-event="some JavaScript">
```

In the following example, an `onclick` attribute (with code), is added to a button element:

```
<button onclick='getElementById("demo").innerHTML=Date()'>The
time is?</button>
```

JavaScript events

In the example above, the JavaScript code changes the content of the element with id="demo".

In the next example, the code changes the content of its own element (using **this.**innerHTML):

```
<button onclick="this.innerHTML=Date()">The time is?</button>
```

JavaScript code is often several lines long. It is more common to see event attributes calling functions:

```
<button onclick="displayDate()">The time is?</button>
```

JavaScript events

Here is a list of some common HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

More events are listed here:
http://www.w3schools.com/jsref/dom_obj_event.asp

Handling events

Event handlers can be used to handle and verify user input, user actions, and browser actions:

- Things that should be done every time a page loads
- Things that should be done when the page is closed
- Action that should be performed when a user clicks a button
- Content that should be verified when a user inputs data
- And more ...

Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly
- HTML event attributes can call JavaScript functions
- You can assign event handler functions to HTML elements from JavaScript code
- You can prevent events from being sent or being handled
- And more ...

Handling events

In this example, the content of the `<h1>` element is changed when a user clicks on it:

```
<!DOCTYPE html>
<html>
<body>

<h1 onclick="this.innerHTML='Ooops!'">Click on this text!</h1>

</body>
</html>
```

Handling events

In this example, a function is called from the event handler:

```
<!DOCTYPE html>
<html>
<body>

<h1 onclick="changeText(this)">Click on this text!</h1>

<script>
function changeText(id) {
    id.innerHTML = "Ooops!";
}
</script>

</body>
</html>
```

Handling events

The HTML DOM allows you to **assign event handler functions to HTML elements** from JavaScript code.

Example:

```
<script>
document.getElementById("myBtn").onclick = displayDate;
</script>
```

In the example above, a function named *displayDate* is assigned to an HTML element with the id="myBtn".
The function will be executed when the button is clicked.

Handling events

The **onload** and **onunload** events are triggered when the user enters or leaves the page.

The `onload` event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

The `onload` and `onunload` events can be used to deal with cookies.

```
<body onload="checkCookies()">
```

The **onchange** event is often used in combination with validation of input fields.
In the example below, the `upperCase()` function will be called when a user changes the content of an input field.

```
<input type="text" id="fname" onchange="upperCase()">
```

Handling events

The **onmouseover** and **onmouseout** events can be used to trigger a function when the user mouses over, or out of, an HTML element.

```
<!DOCTYPE html>
<html>
<body>

<div onmouseover="mOver(this)" onmouseout="mOut(this)"
style="background-color:#D94A38;width:120px;height:20px;padding:40px;">
Mouse Over Me</div>
<script>
function mOver(obj) {
    obj.innerHTML = "Thank You"
}
function mOut(obj) {
    obj.innerHTML = "Mouse Over Me"
}
</script>

</body>
</html>
```

Handling events

The **onmousedown**, **onmouseup**, and **onclick** events are all parts of a mouse-click. First when a mouse-button is clicked, the `onmousedown` event is triggered, then, when the mouse-button is released, the `onmouseup` event is triggered, finally, when the mouse-click is completed, the `onclick` event is triggered.

```
<div onmousedown="mDown(this)" onmouseup="mUp(this)"
style="background-color:#D94A38;width:90px;height:20px;padding:40px;">
Click Me</div>
<script>
function mDown(obj) {
    obj.style.backgroundColor = "#1ec5e5";
    obj.innerHTML = "Release Me";
}
function mUp(obj) {
    obj.style.backgroundColor="#D94A38";
    obj.innerHTML="Thank You";
}
</script>
```

Handling events

The **`addEventListener()`** method is used to add an event listener that fires when an event occurs.

```
element.addEventListener(event, function, useCapture);
```

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

Examples:

```
document.getElementById("myBtn").addEventListener("click",
displayDate);
```

```
element.addEventListener("click", function(){ alert("Hello
World!"); });
```

Note: we can define our own events!

Handling events

When using the `addEventListener()` method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.

The **`removeEventListener()`** method removes event handlers that have been attached with the addEventListener() method:

```
element.removeEventListener("mousemove", myFunction);
```

Handling events

The addEventListener() method allows for adding many events to the same element, without overwriting existing events:

```
element.addEventListener("click", myFunction);
element.addEventListener("click", mySecondFunction);
```

It is also possible to add events of different types to the same element:

```
element.addEventListener("mouseover", myFunction);
element.addEventListener("click", mySecondFunction);
element.addEventListener("mouseout", myThirdFunction);
```

Handling events

The `addEventListener()` method allows for adding event listeners on any HTML DOM object such as HTML elements, the HTML document, the window object, or other objects that supports events, like the XMLHttpRequest object.

In the following example, we add an event listener that fires when a user resizes the window:

```
window.addEventListener("resize", function(){
    document.getElementById("demo").innerHTML = sometext;
});
```

Handling events

When the function has parameters, use an **anonymous function** that calls the specified function with the arguments:

```
element.addEventListener("click", function(){ myFunction(p1, p2); });
```

Handling events

There are two ways of event propagation in the HTML DOM, **bubbling** and **capturing**.

Event propagation is a way of defining the element order when an event occurs. If you have a `<p>` element inside a `<div>` element, and the user clicks on the `<p>` element, which element's "click" event should be handled first?

In *bubbling* the inner most element's event is handled first and then the outer: the `<p>` element's "click" event is handled first, then the `<div>` element's "click" event.

In *capturing* the outer most element's event is handled first and then the inner: the `<div>` element's "click" event will be handled first, then the `<p>` element's "click" event.

Handling events

With the `addEventListener()` method you can specify the propagation type by using the `useCapture` parameter:

```
addEventListener(event, function, useCapture);
```

The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

```
document.getElementById("myP").addEventListener("click",
myFunction, true);
document.getElementById("myDiv").addEventListener("click",
myFunction, true);
```

AJAX

AJAX

AJAX = **Asynchronous JavaScript and XML**
AJAX is a misleading name. You don't have to understand XML to use AJAX.

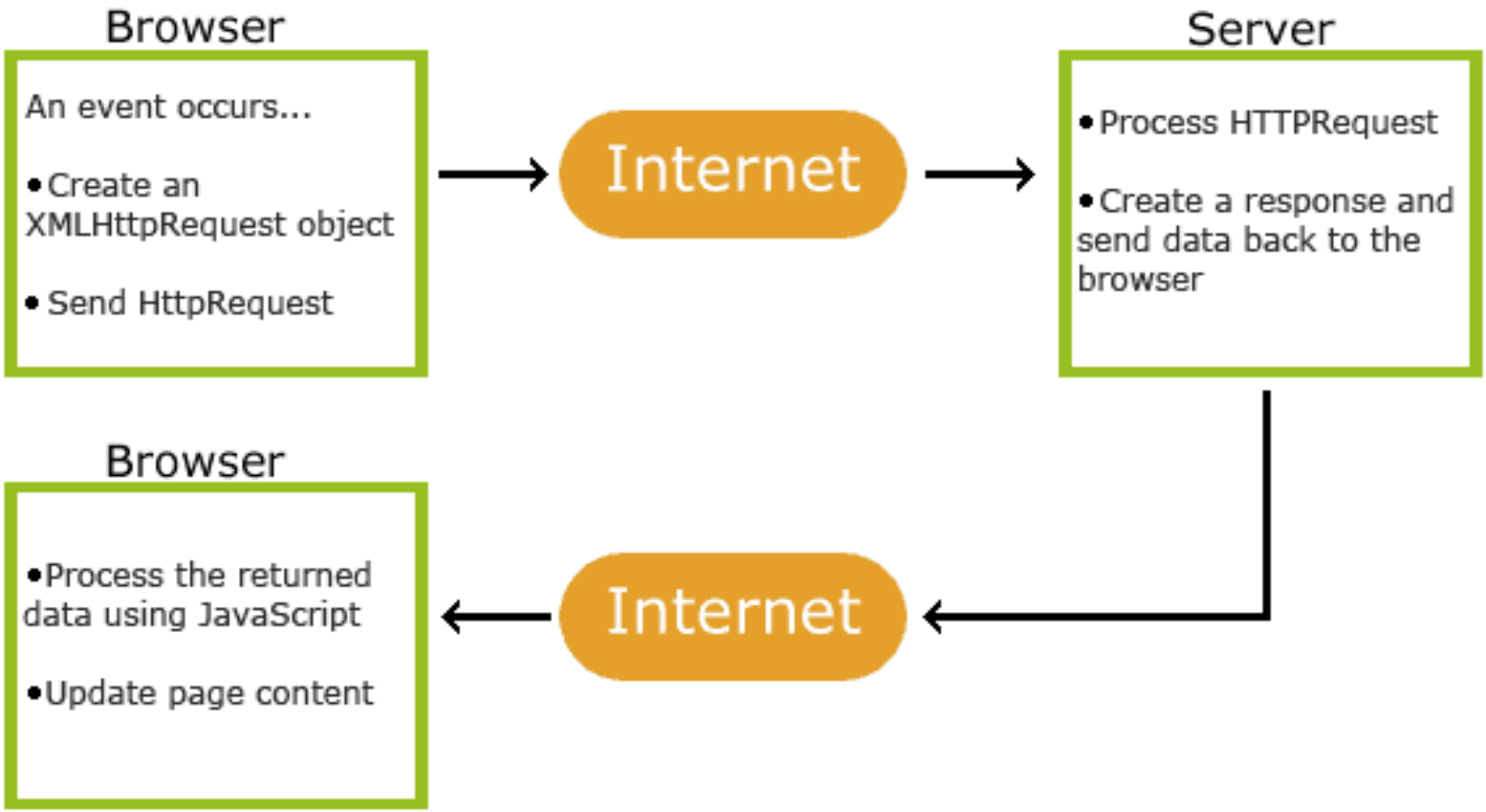
AJAX is a technique for creating fast and dynamic web pages.

AJAX allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

Classic web pages (which do not use AJAX) must reload the entire page if the content should change.

Examples of applications using AJAX: Google Maps, Gmail, YouTube, and Facebook.

AJAX


Browser
An event occurs...
• Create an XMLHttpRequest object
• Send HttpRequest
Internet
Server
• Process HTTPRequest
• Create a response and send data back to the browser
Browser
• Process the returned data using JavaScript
• Update page content
Internet

AJAX

The keystone of AJAX is the **XMLHttpRequest** object, which is supported by all modern browsers.

Syntax for creating an XMLHttpRequest object:

```
variable = new XMLHttpRequest();
```

To handle all browsers, including IE5 and IE6, check if the browser supports the XMLHttpRequest object. If it does, create an XMLHttpRequest object, if not, create an ActiveXObject:

```
let xhttp;
if (window.XMLHttpRequest) {
    xhttp = new XMLHttpRequest();
    } else {
    // code for IE6, IE5
    xhttp = new ActiveXObject("Microsoft.XMLHTTP");
}
```

AJAX

To send a request to a server, use the **open()** and **send()** methods of the XMLHttpRequest object:

Method	Description
open(*method, url, async*)	Specifies the type of request *method*: the type of request: GET or POST *url*: the server (file) location *async*: true (asynchronous) or false (synchronous)
send()	Sends the request to the server (used for GET)
send(*string*)	Sends the request to the server (used for POST)

AJAX

GET or POST?

GET is simpler and faster than POST, and can be used in most cases.

However, always use POST requests when:

- you need to update a file or database on the server;
- you need to send a large amount of data to the server (POST has no size limitations);
- you need to send user input (which can contain unknown characters).

POST is more robust and secure than GET.

AJAX

A simple GET request:

```
xhttp.open("GET", "demo_get.asp", true);
xhttp.send();
```

In the example above, you may get a cached result. To avoid this, add a unique ID to the URL:

```
xhttp.open("GET", "demo_get.asp?t=" + Math.random(), true);
xhttp.send();
```

If you want to send information with the GET method, add the information to the URL:

```
xhttp.open("GET", "demo_get2.asp?fname=Henry&lname=Ford", true);
xhttp.send();
```

https://www.w3schools.com/xml/ajax_xmlhttprequest_send.asp

AJAX

To POST data like an HTML form, add an HTTP header with **setRequestHeader()**.

Method	Description
setRequestHeader(*header, value*)	Adds HTTP headers to the request *header*: specifies the header name *value*: specifies the header value

Specify the data you want to send in the send() method:

```
xhttp.open("POST", "ajax_test.asp", true);
xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhttp.send("fname=Henry&lname=Ford");
```

https://www.w3schools.com/xml/tryit.asp?filename=tryajax_post2

AJAX

The *url* parameter of the open() method is an address to a resource on a server.

If an absolute URL is specified, the protocol, host, and port must match those of the document that calls the open() method (because of the **same-origin policy!**). Cross-origin HTTP requests normally cause an error.

The target resource can be any kind of
- file, like .txt and .xml
- server scripting file, like .asp and .php (which can perform actions on the server before sending the response back)
- resource made accessible by some RESTful service

AJAX

With AJAX, the browser **does not have to wait for the server response**, but can instead:

- execute other scripts while waiting for server response
- deal with the response when the response is ready

Using async=false is not recommended, but for a few small requests this can be ok.

With async=false, the JavaScript will NOT continue to execute, until the server response is ready. If the server is busy or slow, the application will hang or stop.

AJAX

To get the response from a server, use the **responseText** (to get the response data as a string) or **responseXML** (to get the response data as XML data) property of the XMLHttpRequest object.

The responseText property returns the response as a string, and you can use it accordingly:

```
document.getElementById("demo").innerHTML = xhttp.responseText;
```

https://www.w3schools.com/xml/tryit.asp?filename=tryajax_first

AJAX

If the response from the server is XML, and you want to parse it as an XML object, use the responseXML property:

(suppose we requested the file https://www.w3schools.com/xml/cd_catalog.xml)

```
let xmlDoc = xhttp.responseXML;
let txt = "";
let x = xmlDoc.getElementsByTagName("ARTIST");
for (i = 0; i < x.length; i++) {
  txt += x[i].childNodes[0].nodeValue + "<br>";
  }
document.getElementById("demo").innerHTML = txt;
```

http://www.w3schools.com/xml/tryit.asp?filename=tryajax_responsexml

AJAX

When a request to a server is sent, we want to perform some actions based on the response.
The **`onreadystatechange`** event is triggered every time the **`readyState`** property of the XMLHttpRequest object changes. The `readyState` property holds the status of the XMLHttpRequest.

Three important properties of the XMLHttpRequest object:

Property	Description
onreadystatechange	Stores a function (or the name of a function) to be called automatically each time the readyState property changes
readyState	Holds the status of the XMLHttpRequest. Changes from 0 to 4: 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
status	200: "OK" 404: Page not found

AJAX

When readyState is 4 and status is 200, the response is ready:

```
function loadDoc() {
  let xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    document.getElementById("demo").innerHTML = this.responseText;
  }
}
```

Note: The `onreadystatechange` event is triggered five times (0-4), one time for each change in readyState.

Note: `this` stands for `xhttp`

AJAX

A **callback function** is a function passed as a parameter to another function.

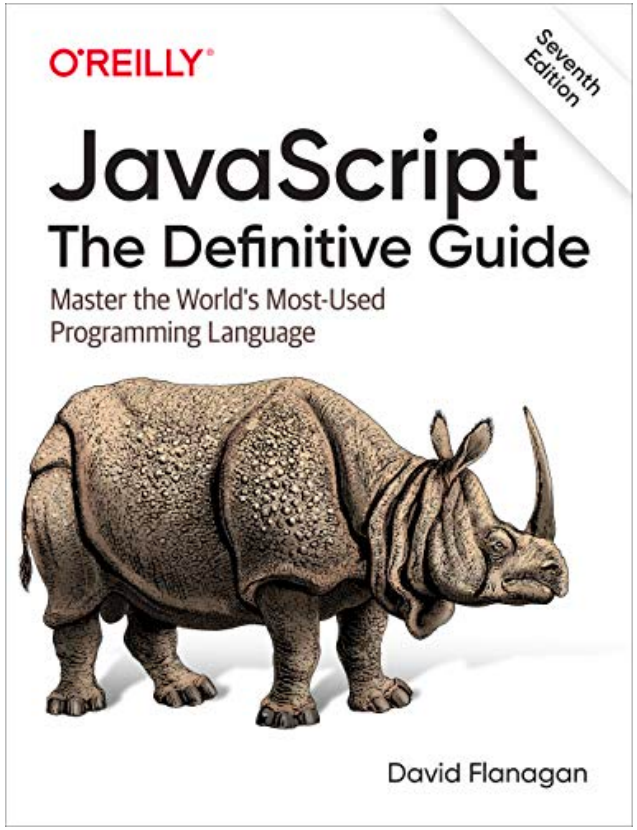
```
function loadDoc(url, cFunc) {
  let xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    cFunc(this);
  }
  xhttp.open("GET", url, true);
  xhttp.send();
}

function cFunc(xhttp) {
  ...
}
```

http://www.w3schools.com/xml/tryit.asp?filename=tryajax_callback

References

D. Flanagan
JavaScript - The Definitive Guide
ed. O'Reilly, 2020



https://www.w3schools.com/xml/ajax_intro.asp