DIG 4104C: WEB DESIGN WORKSHOP

Lecture 06: Messaging: Communicating to and from scripts in HTML5, Server side messaging

Summer 2015

D. Novatnak

Outline

Objectives

- Expose students to methods in HTML5 messaging
- See how client side document messaging is handled
- Gain information on the language and syntax of document messaging
- See how server side messaging is handled
- Gain information on the language and syntax of server side messaging
- Gain understanding of overall messaging concepts

At the end of this lecture, you should be able to….

- Display gained knowledge through the generation of functional code with document messaging
- Display gained knowledge through the generation of functional code with server side messaging
- Explain the similarity and differences between the messaging systems

HTML5 MESSAGING

PSSST…YOU…YEA YOU… HI!

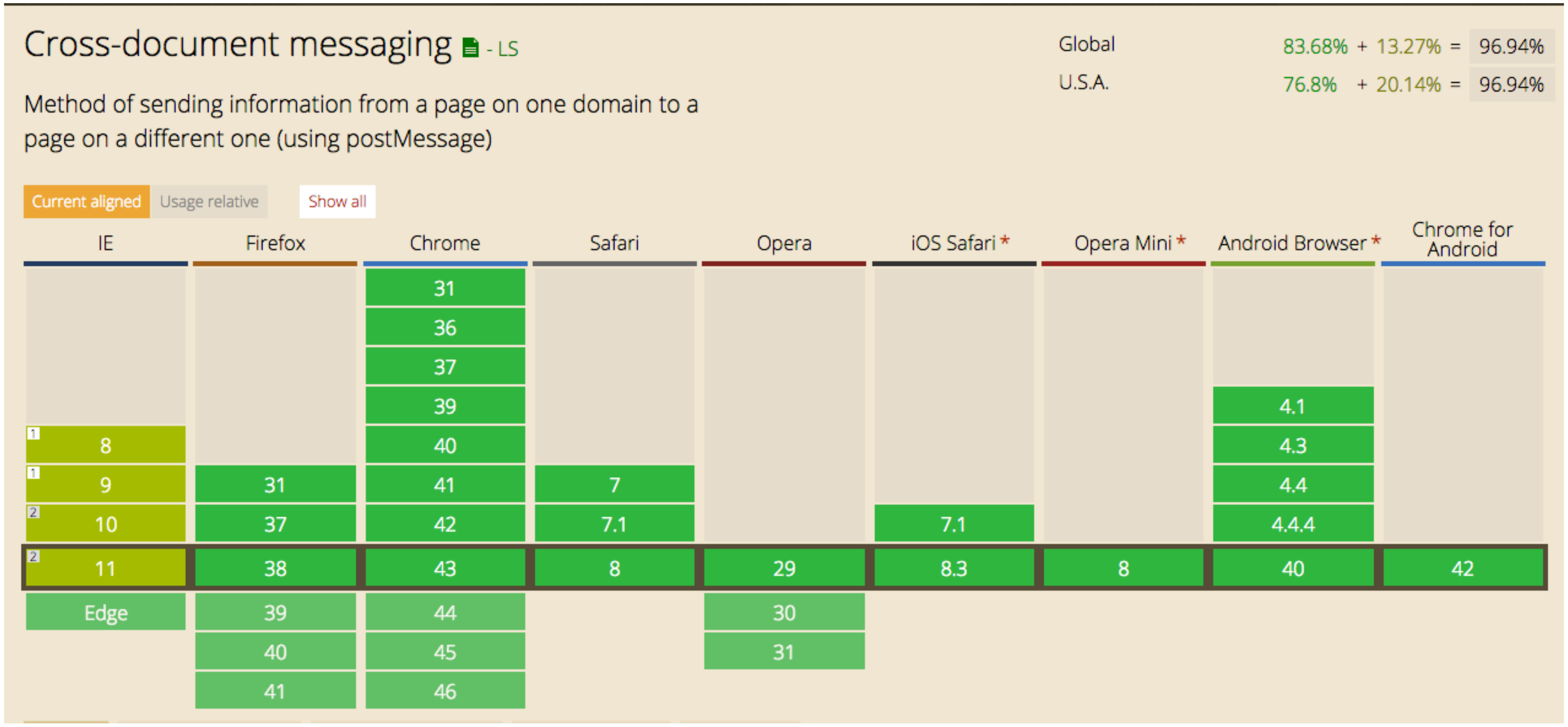
Introduction

- Web messaging is a way for documents in separate browsing contexts to share data without the DOM being exposed to malicious cross-origin scripting.
- Unlike other forms of cross-site communication (cross-domain XMLHttpRequest, or dynamic script insertion), web messaging never directly exposes the DOM.

Web messaging

- When we talk about web messaging, we're actually talking about two slightly different systems: cross-document messaging and channel messaging.
  - Cross-document messaging is often referred to by its syntax as window.postMessage(),
  - Channel messaging is also known as MessageChannel.
- Along with server-sent events and web sockets, cross-document and channel messaging are a valuable part of the HTML5 "suite" of communication interfaces.

Browser compatibility



Cross-document messaging - LS

Method of sending information from a page on one domain to a page on a different one (using postMessage)

Global	83.68%	+	13.27%	=	96.94%
U.S.A.	76.8%	+	20.14%	=	96.94%

IE	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
		31						
		36						
		37						
		39					4.1	
[1] 8		40					4.3	
[1] 9	31	41	7				4.4	
[2] 10	37	42	7.1		7.1		4.4.4	
[2] 11	38	43	8	29	8.3	8	40	42
Edge	39	44		30				
	40	45		31				
	41	46						

MessageEvent interface

- Cross-document messaging, channel messaging, server-sent events and web sockets all fire message events, so understanding it is helpful.
- Message events contain five read-only attributes:
  - data:
    - Contains an arbitrary string of data, sent by the originating script.
  - origin:
    - A string containing the originating document's scheme, domain name, and port (for example: https://domain.example:80)
  - lastEventId:
    - A string containing a unique identifier for the current message event.
  - Source:
    - A reference to the originating document's window. More accurately, it's a WindowProxy object.
  - Ports:
    - An array containing any MessagePort objects sent with the message.

MessageEvent interface

- In the case of cross-document messaging events and channel messaging, the value of lastEventId is always an empty string.
- lastEventId applies to server-sent events.
- If no ports are sent with the message, the value of the ports attribute will be an array whose length is zero.
- MessageEvent inherits from the DOM Event interface and shares its properties.
- Message events however do not bubble, are not cancelable, and have no default action.

Cross-document messaging

- Sending a cross-document message requires a new browsing context either by creating a new window or referencing an iframe.
- Then we can send a message from it using the postMessage() method.

- For cross-document messaging, postMessage() requires two arguments.
  - message: The message to send.
  - targetOrigin: The origin to which the message will be sent.

  - Note: There is also an optional third argument, transfer

Cross-document messaging

- The message parameter is not limited to strings.
  - Structured objects, data objects (such as File and ArrayBuffer), or arrays can also be sent as messages.
- The targetOrigin is the origin of the receiving document.
  - Browsers will not send the message unless the origin of the receiving browsing context matches the one provided in targetOrigin.
  - You can circumvent this restriction using the * wild card character. Doing so however can lead to information leakage, so it’s best to set a specific target origin.
- You can also limit message sending to the same origin by setting the targetOrigin argument to /

Example

- Send a message from our parent document to a document contained within an iframe.

```
var iframe = document.querySelector('iframe');
var button = document.querySelector('button');

var clickHandler = function(){
// iframe.contentWindow refers to the iframe’s window object.
iframe.contentWindow.postMessage('The message to send.','http://
www.example.com');
}

button.addEventListener('click',clickHandler,false);
```

Example

- Even though our documents share the same origin, for cross-browser compatibility set the value of targetOrigin to https://www.example.com instead of /.
- If our document lived on another domain, we could send a message using its origin as the target.
- Note that origins do not contain a trailing slash.

Receiving a cross-document message

- Sending an event is only half of the process.
- We also need to handle these events in the receiving document.
- Each time postMessage() is called, a message event is dispatched in the receiving document.

Example

- We can then listen for the message event as shown:

```
var messageEventHandler = function(event){
// check that the origin is one we want.
if(event.origin == 'http://www.example.com'){
        alert(event.data);
}
}
window.addEventListener('message', messageEventHandler,false);
```

In action: crossdocmessaging.html

Detecting when the receiving document is ready

- In the examples, window.postMessage() is being invoked inside an event handler that requires user interaction. For a simple demo, this is fine.
- A better way to handle this in the real world, however, is to ensure that scripts in the target browsing context have had time to set up listeners and that they are ready to receive our messages.
- To check that, we can send a message event to our parent document when the new document is loaded.

Example

- In this example, we are going to open a new window. When the document in that window loads, it will post a message to the opening window. Let’s also assume that our markup has a button element, which is how we will open the new window.

Example

```
var clickHandler, messageHandler, button;

button = document.querySelector('button');

clickHandler = function(){

window.open('otherpage.html','newwin','width=500,height=500');
}

button.addEventListener('click',clickHandler,false);

messageHandler = function(event){
        if(event.origin == 'http://foo.example'){
                event.source.postMessage('This is the
message.','http://foo.example');
        }
}

window.addEventListener('message',messageHandler, false);
```

Receiving a cross-document message

- When our button is clicked
  - the clickHandler function will open a new window
  - messageHandler function will listen for the message from the opened window.
  - Note that event.source is a WindowProxy object that represents our opened window.

Receiving a cross-document message

- In our opened window, we will listen for the DOMContentLoaded event

```
var loadHandler = function(event){
		event.currentTarget.opener.postMessage('ready','http://foo.example');
}
window.addEventListener('DOMContentLoaded', loadHandler, false);
```

- When it is fired, it will use window.postMessage() to “notify” the opening document that it is ready to receive messages

- Demo: webmessaging-tellopener.html

Channel messaging

- Channel messaging provides a means of direct, two-way communication between browsing contexts.
- As with cross-document messaging the DOM is not directly exposed.
- Instead, at each end of our pipe is a port; the data sent from one port becomes input in the other (and vice-versa).
- Channel messaging is particularly useful for communication across multiple origins.
- Consider the following scenario:
- We have a document at http://socialsite.example containing content from http://games.example embedded in one iframe, and content from http://addressbook.example in another.

Channel messaging

- Consider the following scenario:
- We have a document at http://socialsite.example containing content from http://games.example embedded in one iframe, and content from http://addressbook.example in another.

Channel messaging

- Now let’s say that we want to send a message from our address book site to our games site.
- We could use the social site as a proxy.
- That, however, means the address book gains the same level of trust as the social site.
- Our social site either has to trust every request, or filter them for us.

- With channel messaging, however, http://addressbook.example and http://games.example can communicate directly.

MessageChannel and MessagePort Objects

- When we create a MessageChannel object, we’re really creating two interrelated ports. One port stays open on our sending side. The other is forwarded to another browsing context.

- Each port is a MessagePort object with three available methods.

  - postMessage(): Posts a message through the channel.
  - start(): Begins the dispatch of messages received on the port.
  - close(): Closes and deactivates the port.

- MessagePort objects also have an onmessage event attribute, which can be used to define an event handler function instead of adding an event listener.

Example of communicating with channel messaging

- We'll use a scenario similar to what's described above: a document containing two iframes. We will send messages from one iframe to the other, using a MessageChannel object and ports.

- All of the documents in the examples linked above have the same origin. However, the process is the same for cross-origin communication.

- In our first iframe, we will do the following.
  - Create a new MessageChannel object.
  - Transfer one MessageChannel port to our parent document where it will be forwarded to our other iframe.
  - Define an event listener for our remaining port to handle the message sent from our other iframe
  - Open our port so that we can receive messages.
  - We will also wrap everything in a function that will be invoked when the DOM is ready.

Example

```
var loadHandler = function(){
  var mc, portMessageHandler;
  mc = new MessageChannel();
  // Send a port to our parent
  document.window.parent.postMessage('documentAHasLoaded','http://foo.example',
[mc.port2]);
  // Define our message event handler.
  portMessageHandler = function(portMsgEvent){
          alert( portMsgEvent.data );
  }

  // Set up our port event listener.
  mc.port1.addEventListener('message', portMessageHandler, false);
  // Open the port
  mc.port1.start();
}

window.addEventListener('DOMContentLoaded', loadHandler, false);
```

Example

- Now in our parent document, we will listen for this incoming message and associated port. When it’s received, we will post a message to our second iframe, and forward our port with that message.

```
var loadHandler = function(){
          var iframes, messageHandler;

          iframes = window.frames;

          // Define our message handler.
          messageHandler = function(messageEvent){
                    if( messageEvent.ports.length > 0 ){
                              // transfer the port to iframe[1]
                              iframes[1].postMessage('portopen','http://
foo.example',messageEvent.ports);
                    }
          }

          // Listen for the message from iframe[0]
          window.addEventListener('message',messageHandler,false);
}

window.addEventListener('DOMContentLoaded',loadHandler,false);
```

Example

- Finally, in our second iframe, we can handle the message from our parent document, and post a message to the port.
- The message sent from this port will be handled by the portMsgHandler function in our first document.

```
var loadHandler(){
          // Define our message handler function
          var messageHandler = function(messageEvent){

                    // Our form submission handler
                    var formHandler = function(){
                              var msg = 'add <foo@example.com> to game circle.';
                              messageEvent.ports[0].postMessage(msg);
                    }
                    document.forms[0].addEventListener('submit',formHandler,false);
          }
          window.addEventListener('message',messageHandler,false);
}

window.addEventListener('DOMContentLoaded',loadHandler,false);
```

For more on messaging

- HTML5 Specification on messaging
  - http://dev.w3.org/html5/postmsg/

- So much good info here.

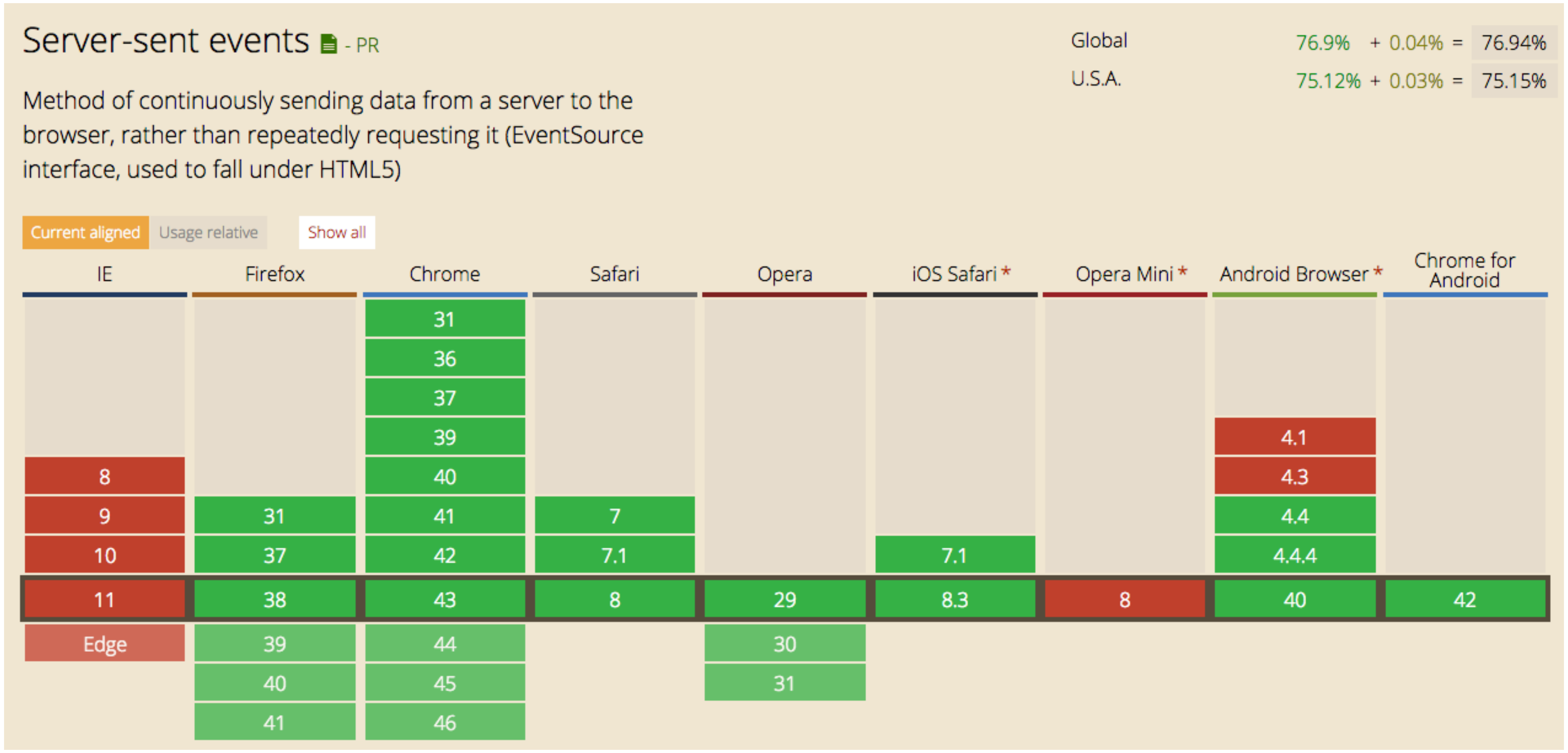
SERVER SIDE MESSAGING

Server side messaging

- HTML5 server-sent events are one way for the web pages to communicating with the web server.

- But: It is also possible with the XMLHttpRequest object that lets your JavaScript code make a request to the web server. It is a one-for-one exchange: once the web server provides its response, the communication is over.

Browser Support



Server-sent events - PR

Method of continuously sending data from a server to the browser, rather than repeatedly requesting it (EventSource interface, used to fall under HTML5)

Global	76.9%	+ 0.04%	=	76.94%
U.S.A.	75.12%	+ 0.03%	=	75.15%

Current aligned | Usage relative | Show all

IE	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
		🟩 31						
		🟩 36						
		🟩 37						
		🟩 39					🟥 4.1	
🟥 8		🟩 40					🟥 4.3	
🟥 9	🟩 31	🟩 41	🟩 7				🟩 4.4	
🟥 10	🟩 37	🟩 42	🟩 7.1		🟩 7.1		🟩 4.4.4	
🟥 11	🟩 38	🟩 43	🟩 8	🟩 29	🟩 8.3	🟥 8	🟩 40	🟩 42
🟥 Edge	🟩 39	🟩 44		🟩 30				
	🟩 40	🟩 45		🟩 31				
	🟩 41	🟩 46						

When to use Server side

- There are some situations where web pages require a longer-term connection to the web server.
- Stock quotes on finance websites where price updated automatically.
- Another example is a news ticker running on various media websites.

- But how?

HTML5 server-sent events feature.

- Allows a web page to hold an open connection to the web server so that the web server can send a new response automatically at any time
- There's no need to reconnect, and run the same server script from scratch over and over again.

Example

- Sending Messages with a Server Script
  - Report the current time of the web server's built-in clock in regular intervals.

```php
<?php
header("Content-Type: text/event-stream");
header("Cache-Control: no-cache");

// Get the current time on server
$currentTime = date("h:i:s", time());

// Send it in a message
echo "data: " . $currentTime . "\n\n";
flush();
?>
```

Example Explained

- PHP script sets two important headers.
    - First, it sets the MIME type to text/event-stream, which is required by the server-side event standard.
    - The second line tells the web server to turn off caching otherwise the output of your script may be cached.
- Every message send through HTML5 server-sent events must start with the text "data:" followed by the actual message text and the new line character sequence ("\n \n").
- Finally, the PHP flush() function makes sure that the data is sent right away, rather than buffered until the PHP code is complete.

Processing Messages in a Web Page

- The EventSource object is used to receive server-sent event messages.
- Let's extend the previous example and create an HTML document simply receives the current time reported by the web server and display it to the user.

Example

- Processing Messages in a Web Page
  - Leverage previous code to display the formatted time from previous example

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>HTML5 Server-Sent Events</title>
<script type="text/javascript">
    window.onload = function(){
        var source = new EventSource("server_time.php");
        source.onmessage = function(event){
            document.getElementById("result").innerHTML += "New time received from web
server: " + event.data + "<br>";
        };
    };
</script>
</head>
<body>
    <div id="result">
        <!--Server response will be inserted here-->
    </div>
</body>
</html>
```

DISCUSSION

How would you use messaging?

Summary

- Messaging is useful on both the client and server side
- Messaging opens possibilities for additional interactions between documents
- Cross document messaging permits content to be passed securely between documents in the same domain
- Server side messaging permits the reception of data that is updated on a regular basis without refreshing the connection to the server

Questions?

References

- Crowther, R., & Lennon, J. (n.d.). HTML5 in action.
- HTML5 Web Messaging (n.d.). Retrieved May 25, 2015, from http://www.w3.org/TR/webmessaging/
- https://dev.opera.com/