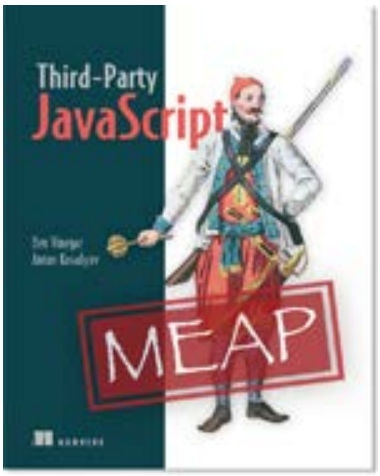
MANNING PUBLICATIONS



Third-Party JavaScript
By Ben Vinegar and Anton Kovalyov

Legacy browsers—including Internet Explorer versions 6 and 7—have absolutely no support for the window.postMessage API so you have to fall back to some other approach. In this article based on chapter 5 of Third-Party JavaScript, the authors discuss four hacks: using window.name property, using window.hash property, using window.opener property, and using a Flash object.

window.postMessage Alternatives for Legacy Browsers

All modern browsers support the `window.postMessage` API in their stable versions while older browsers—such as Internet Explorer 6 or 7—don't have that support. For these browsers, you will have to employ a few hacks that allow you to use undocumented browser features to pass small strings back and forth between an iframe and its parent window. All of the techniques described in this article transmit messages much more slowly than `window.postMessage`, and you should only use them as your backup plan.

In this article, we will go over four techniques: using `window.name` property, using `window.hash` property, using `window.opener` property, and, finally, using a Flash object. All those techniques base themselves on a fact that some browsers—intentionally or due to bugs—skip the Same-Origin Policy checks when dealing with respective components.

Since you can't simply broadcast your message or subscribe to an event without `window.postMessage`, you will emulate the broadcast by writing data to a special property. As for subscribing, you will have to constantly poll these property values to check if they have been changed by another window. It is not fun, but it is necessary. Let's start with the first technique, the one that uses `window.name` property to exchange messages.

Sending messages using `window.name` property

First of all, what is the `name` property? The `name` property belongs to the global `window` object and stores the name of the current window so that other windows—such as parent windows that opened the current one as a popup—can get a reference to it by its name. This property has one peculiarity: once set, its value doesn't change when the window is redirected to a new URL. You can use this behavior to bypass the Same-Origin Policy check that kicks in every time you try to read a `window.name` property from a page with a different origin.

You'll start by creating a new, hidden iframe and redirecting it to your target page. This target page must set the `window.name` property for that frame to whatever value it wants to pass back to the host page. Since the `window.name` property is accessible only within the same domain, you will need to listen to the `onload` event for your newly created iframe and then redirect it to a different page but on the same domain as a host page. After that, you should be able to read the message from the name property. Listing 1 shows an example of a client creating a new iframe and polling for new messages.

Listing 1 Client listening to new messages using the window.name workaround

```
<!DOCTYPE html>

<html>
  <head>
    <script>
      function log(msg) {
        var el = document.getElementById('log');
        el.innerHTML += '<br>' + msg;
      }

      function start() {
        log('Creating iframe');

        var el = document.createElement('iframe');      #1
        var bd = document.getElementsByTagName('body')[0];

        el.style.display = 'none';
        el.src = 'http://camerastork.com/nametransport/server.html';

        var done = false;
        el.onreadystatechange = function () {       #2
          if (!el.readyState == 'complete' || done)
            return;

          log('Listening');
          var name = el.contentWindow.name;       #3
          if (name) {
            log('Data: ' + el.contentWindow.name);
            done = true;
          }
        };

        bd.appendChild(el);
      }
    </script>
  </head>

  <body>
    <button onclick="start()">Start</button>
    <pre id="log"></pre>
  </body>
</html>
```

#1 Creates an iframe and set necessary attributes
#2 Function to be called when iframe is loaded
#3 If there is a value in el.contentWindow.name, it prints it

The code on the server is also not very complicated. It simply changes the property and then redirects its host `window.name Window` object to an empty HTML page hosted on the same domain as the client:

```
<!DOCTYPE html>

<html>
  <head>
    <script>
      function init() {
        window.name = 'Hello, World!';
        window.location = 'http://example.com/empty.html';
      }
    </script>
  </head>

  <body onload="init();"></body>
</html>
```

This empty.html file is exactly what you think it is, an empty page stub, because browsers need to load something.

```
<!DOCTYPE html>
<html></html>
```

One disadvantage of this approach is that you have to make a network request every time you want to retrieve a new message. In addition, most of the time, widget developers don't have access to their customers' websites so you don't have an empty page you can redirect to in the last step. That means that you either have to ask your

users to host such a page on their website or you will have to redirect to some other random URL on the client's website—like a 404 page. Using a random UR like this, however, can significantly alter that site's traffic statistics by increasing the number of requests to their pages.

Security implications

In theory, other frames loaded on the page might attempt to access the loading frame and navigate it to their own URLs in order to get hold of the data you placed in the property. In practice, navigating a `window.name` frame from another frame that is neither a child nor a parent of that frame is prohibited in most browsers with one exception: Firefox 2. But that version of Firefox has almost no market share so you shouldn't worry about it too much.

Clicking sound problem when changing the location

The `window.name` trick works pretty well in legacy browsers with one exception: older versions of Internet Explorer—such as 6 and 7—accompany every location change with an annoying clicking sound. As you can imagine, if you are making a lot of cross-domain requests with this method, your computer will sound like a toy machine gun.

To solve this problem in Internet Explorer 6, you just need to add a `<bgsound>` element to the page. The browser will think that there is some background music playing and not play the clicking sound when navigating from the page. Unfortunately, that trick doesn't work in Internet Explorer 7, so we have to utilize some ActiveX magic as shown in listing 2.

Listing 2 Using a detached document to silence the clicking sound in IE7

```
var doc, iframe, html;

html =
  "<html><body>" +
    "<iframe id='iframe'></iframe>" +
  "</body></html>";

if ("ActiveXObject" in window) {
  doc = new ActiveXObject("htmlfile");     #1

  doc.open();
  doc.write(html);
  doc.close();      #2

  iframe = doc.getElementById('iframe');
} else {
  iframe = document.createElement('iframe');
  document.body.appendChild(iframe);
}
```

#1 Place your iframe inside of ActiveX object
#2 Don't forget to close opened object

As you can see, we simply create a new ActiveX object "htmlfile" that acts as an HTML document but disconnected from the UI. The fact that it is disconnected from the UI gives the browser enough of a reason to save resources—and our nerve cells—and not play the clicking sound.

Sending messages using window.hash property

In browser environments, all `Window` objects have a property called `location`. This property can be used to get information about the current document and its URL as well as to redirect the current document to another URL.

```
console.log(window.location.href);
> http://thirdpartyjs.com/

window.location = 'http://example.com/';

console.log(window.location.href);
> http://example.com/
```

When working with iframes and other child windows—such as popup windows—this property is also accessible but exclusively in a write-only mode.

That means that you still can redirect a window to another URL but you don't have any way to find out the current URL of a child window. This policy was implemented as a security measure so that a malicious website couldn't simply open your favorite webmail client in a hidden iframe and gather information by logging redirects.

Another interesting detail is that not all URL changes result in network requests. HTML defines a special portion of a URL—called the anchor portion or fragment identifier—that is designed to point to a location inside the current document. This portion always comes last in a URL string and is preceded by the hash (#) sign.

```
http://camerastork.com/#products
```

The fallback technique you are about to learn transfers small chunks of data across domain by changing each other's fragment identifiers. Since changing fragment identifiers doesn't reload the page, you can maintain state in each frame.

Always keep in mind that you can't read an iframe's URL; you can only write to it. So when loading a document into an iframe, you have to pass your current URL into it so that it doesn't accidentally redirect your publisher's page to some other address. You can then start sending information by changing iframe's `location` property to its original URL plus your data in the anchor portion. And when the child window wants to send you some data back, it can change its parent window's location to the URL you sent with your initial request, plus the data. Listing 3 shows an example of the client creating a new iframe and listening to new messages using the `window.hash` technique.

Listing 3 Client listening to new messages using the window.hash workaround

```
<!DOCTYPE html>

<html>
  <head>
    <script>
      function log(msg) {
        var el = document.getElementById('log');
        el.innerHTML += '<br>' + msg;
      }

      function start() {
        log('Creating iframe');

        var el = document.createElement('iframe');
        var bd = document.getElementsByTagName('body')[0];
        var url = 'http://example.com/client.html';

        el.style.display = 'none';
        el.src = 'http://camerastork.com/hashtransport/server.html?'
            + encodeURIComponent(url);

        bd.appendChild(el);

        var listener = function () {
          var hash = location.hash;

          if (hash && hash != '#') {     #1
            log("Incoming: " + hash.replace('#', ''));
            window.location.href = url + '#';
          }

          setTimeout(listener, 100);
        };

        listener();
      }
    </script>
  </head>

  <body>
    <button onclick="start()">Start</button>
    <pre id="log"></pre>
  </body>
</html>
```

#1 Checks for changes in location.hash value

Listing 4 shows an example of a page hosted on the server that sends a message to the parent page by modifying the parent's fragment identifier.

Listing 4 Example of sending messages to the client by modifying parent window's fragment identifier

```
<!DOCTYPE html>

<html>
  <head>
    <script>
      function init() {
        var url = window.location.href;
        url = url.split('?')[1].replace('?', '');
        window.parent.location = decodeURIComponent(url)
            + '#helloworld';
        }
      </script>
    </head>

    <body onload="init()"></body>
</html>
```

Note that to receive data you have to monitor the current URL in order to know when it has been changed. You can do that by polling the URL every *n* milliseconds and comparing it with the previous version. In addition to that, HTML5 defined a hashchange event that is triggered whenever the hash portion of URL is changed. However, it is supported only by modern browsers where you can already use the API. Listing 5 shows `window.postMessage` an example of code that polls for hash changes every 100 milliseconds.

Listing 5 Polling for hash changes every 100 milliseconds

```
var hash;

function listener() {
  var cur = location.hash;

  if (cur && hash != '#' && cur != hash) {
    log("Incoming: " + cur.replace('#', ''));
    hash = cur;
  }

  setTimeout(listener, 100);
};

listener();
```

As you can see, this is a pretty straightforward technique that doesn't require much effort to implement. Unfortunately, as almost everything in life, its simplicity comes with some serious limitations.

URL size limitation when using fragment transport

The first issue that most implementors have to deal with is the URL size limit in Internet Explorer. Microsoft requires all URLs to be limited to 2083 characters and, since you're sending your data by changing the URL, you will most probably have to break your message into smaller consecutive chunks before sending it. And while not particularly difficult, the whole splitting message into packets part adds to the overall complexity of this transport protocol.

The fragment transport technique is a pretty good one since it works reliably in older browsers. Now let's go over a couple of other workarounds that allow us to emulate the `window.postMessage` API. We will start with the one bug we didn't want Microsoft to fix: a bug with the `window.opener` property behavior.

Sending messages using window.opener property

We first found out about the so-called Native IE XDC (NIX) technique from John Hjelmstad and Joey Schorr of Apache Shindig project.[1] They discovered that in Internet Explorer 6 and 7 any party can set the `window.opener` property on a Window object but only the controlling window can read from it. They then introduced a technique that uses that behavior to create a bidirectional communication channel between frames. However, due to security reasons, you can't pass a JavaScript object through `window.opener` so they had to write a special VBScript (COM) wrapper since COM objects don't have that restriction.

APACHE SHINDIG PROJECT

If you are interested in third-party widgets development, check out the Shindig project from the Apache Foundation. Apache Shindig is an OpenSocial container that provides utilities to render gadgets, proxy requests and handle REST/RPC requests. Even if you don't plan to use it, the project page contains tons of interesting information on cross-domain communication and third-party JavaScript development.

Unfortunately for all third-party JavaScript developers who employed this bug, Microsoft fixed it and by doing so made this technique practically obsolete. The fix went out pretty dramatically without any prior notification so developers literally had to find an alternative—the one that didn't require an intermediate page—overnight. And that is when Adobe Flash came to rescue.

Sending messages using Flash

Adobe's Flash plugin doesn't follow the SOP implemented by browsers. Instead, it assumes that if you let a user upload a Flash object on your website, you implicitly trust everything they do. Third-party JavaScript widgets can load anything on the website, including a special Flash object that acts as a tunnel between the host site and an iframe. It doesn't matter which host you load this Flash object from—you can load it from any of the two participating domains, or you can use a common host—such as CDN—to load the file faster.

The actual Flash object should be written in ActionScript—a dialect of ECMAScript developed by Macromedia Inc. back in 1999 specifically for Flash scripting. Since it is a dialect of ECMAScript, it has the same syntax and semantics as JavaScript so you shouldn't have any difficulties in understanding the examples we are about to show you.

To establish a connection between the Flash object and its container, you can define public interfaces using the `flash.external.ExternalInterface` class. The class has a public `addCallback` method that can be used to register an ActionScript method as callable from the container. For example, to define a public `postMessage` method on the Flash object, you just need to call the `ExternalInterface.addCallback` and provide your method's implementation.

```
import flash.external.ExternalInterface;

class Main {
  private static function main(swfRoot:MovieClip):Void {
    ExternalInterface.addCallback("postMessage", {},
        function(channel:String, message:String) {
            /* ... */
        }
    );
  }
}
```

The `ExternalInterface` class also exposes a public `call` method that can be used to access JavaScript properties and methods from the host page. You can use that method to notify the host page about new messages coming their way.

The example in listing 6 shows an implementation of the `onMessage` function that notifies an external JavaScript method on the host page. We took this example from the source code of a popular cross-domain messaging library called easyXDM.

---

[1] Implement window.opener-based IE transport (NIX) in gadgets.rpc—https://issues.apache.org/jira/browse/SHINDIG-416

Listing 6 Example of the onMessage implementation from easyXDM JavaScript library

```
listeningConnection.onMessage =
  function(message, fromOrigin, remaining) {
    if (fromOrigin !== remoteOrigin) {
      return;
    }

    incommingFragments.push(message);     #1

    if (remaining <= 0) {
      // escape \\ and pass on
      ExternalInterface.call(prefix +     #2
        "easyXDM.Fn.get(\"flash_" + channel +
        "_onMessage\")",
      incommingFragments.join("").split("\\").join("\\\\"),
        remoteOrigin);

      incommingFragments = [];
    } else {
      log("received fragment, length is " +
        message.length + " remaining is " + remaining);
    }
};
```

#1 Queues message parts

#2 Passes the complete message to the provider

The ActionScript implementation of the cross-domain channel is not the most fun program to write so usually developers—including authors of this book—simply reuse an existing open source implementation that was written and tested by somebody else. If you are interested in the specifics of the ActionScript implementation, we encourage you to check out the source code of easyXDM. It is available on GitHub: https://github.com/oyvindkinsey/easyXDM/. Now let's talk a bit about a couple of gotchas that quite often bite developers who decide to rely on Flash for their cross-domain communication needs.

Problems with cross-domain channels based on Flash

Most of the time, the iframe that you use to communicate with your domain is hidden from the page. The first thing that comes to mind when you think about hiding an element on the page is to set its CSS property `display` to `none`.

Another approach would be to set its CSS property `visibility` to `hidden`. However, Internet Explorer 7 makes a rather odd optimization where it doesn't initialize any Flash objects that are being loaded from hidden iframes. That means that you will see your Flash object requested and loaded from the server but, after that, nothing will happen. To solve this problem, you need to hide your iframe in a different way, by setting its `position` property to `absolute` and its `top` property to some ridiculous value such as `-10000px`. That will trick Internet Explorer into thinking that your iframe is still visible (despite being outside the viewport) so the browser will initialize the Flash object inside that iframe right away.

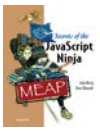
Another problem occurs with newer versions of Flash, which throttle access to hidden Flash objects. We don't know any good workarounds for this problem with an exception of not hiding your object. Usually, placing it at the top-right corner with width and height being equal to 20px is enough.

Aside from these two problems and occasional slowness, Flash is a pretty decent technique when everything else fails. On the other hand, this technique is not pure JavaScript, so we encourage you to try using `window.name` or `window.hash` before resorting to Adobe Flash. Or, you can reuse the easyXDM library that was written and tested by other developers and companies such as Disqus, Twitter, and LinkedIn.

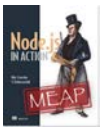
Summary

Legacy browsers—including Internet Explorer versions 6 and 7—have absolutely no support for the `window.postMessage` API, so you have to fall back to some other approach. We discussed iframe messaging fallbacks, going over several hacks you can use to send messages between iframes.

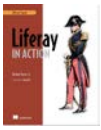
Here are some other Manning titles you might be interested in:



Secrets of the JavaScript Ninja
John Resig and Bear Bibeault



Node.js in Action
Mike Cantelon and TJ Holowaychuk



Liferay in Action
The Official Guide to Liferay Portal Development
Richard Sezov, Jr

Last updated: January 14, 2012