Cross-Browser Implementation Strategies

Knowing the cross-browser issues is only half the battle—figuring out effective management strategies and using them to implement robust cross-browser code is another matter.

There are a wide a range of strategies that we can use, and, while not every strategy will work in every situation, the range that we'll examine should provide a good set of tools for covering most of the concerns that we need to address within our robust code bases.

We'll start with something that's easy and almost trouble free.

Safe cross-browser fixes

The simplest (and safest) classes of cross-browser fixes are those that exhibit two important traits:

- They have no negative effects or side effects on other browsers.
- They use no form of browser or feature detection.

The instances in which we can apply such fixes may be rather rare, but they're a tactic that we should always strive for in our applications.

Let's look at an example. The following code snippet represents a change (plucked from jQuery) that came about when working with Internet Explorer:

```
// ignore negative width and height values
if ((key == 'width' || key == 'height') && parseFloat(value) < 0)
  value = undefined;
```

Some versions of IE throw an exception when a negative value is set on the `height` or `width` style properties. All other browsers ignore negative input. This workaround simply ignores all negative values in *all* browsers. This change prevented an exception from being thrown in Internet Explorer and had no effect on any other browser. This was a painless change that provided a unified API to the user (because throwing unexpected exceptions is never desired).

Another example of this type of fix (also from jQuery) appears in the attribute manipulation code. Consider this:

```
if (name == "type" &&
    elem.nodeName.toLowerCase() == "input" &&
    elem.parentNode)
  throw "type attribute can't be changed";
```

Internet Explorer doesn't allow us to manipulate the `type` attribute of input elements that are already part of the DOM—attempts to change this attribute result in a proprietary exception being thrown. jQuery came to a middle-ground solution: it disallows *all* attempts to manipulate the `type` attribute on injected input elements in all browsers, throwing a uniform informational exception.

This change to the jQuery code base required no browser or feature detection; it unified the API across all browsers. The action still results in an exception, but that exception is uniform across all browser types.

This particular approach could be considered quite controversial—it purposefully limits the features of the library in all browsers because of a bug that exists in only one. The jQuery team weighed this decision carefully and decided that it was better to have a unified API that worked consistently than an API that would break unexpectedly when developing cross-browser code. It's very possible that you'll come across situations like this when developing your own reusable code bases, and you'll need to consider carefully whether a limiting approach such as this is appropriate for your audience.

The important thing to remember for these types of code changes is that they provide a solution that works seamlessly across browsers without the need for browser or feature detection, effectively making them immune to changes going forward. You should always strive for solutions that work in this manner, even if the applicable instances are few and far between.

Object detection

Object detection is a commonly used approach when writing cross-browser code, being not only simple but also generally quite effective. It works by determining if a certain object or object property exists, and if so, assuming that it provides the implied functionality. (In the next section, we'll see what to do about cases where this assumption fails.)

Most commonly, object detection is used to choose between multiple APIs that provide duplicate pieces of functionality. For example, the following code shows object detection used to choose the appropriate event-binding APIs provided by the browser.

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false); }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle); }
}
```

In this example, we checked to see if a property named `addEventListener` exists; if so, we assume that it's a function that we can execute and that it'll bind an event listener to that element. We then proceed to test other APIs, such as `attachEvent`, for existence.

Note that we tested for `addEventListener`, the *standard* method provided by the W3C DOM Events specification, first. This is intentional.

Whenever possible, we should default to the standard way of performing any action. This will help to make our code as future-proof as possible. Moreover, pressure from mass-adoption libraries, as well as very vocal and influential voices in the Twitter-verse and other social media, can encourage browser vendors to work toward providing the standard means of performing actions.

An important use of object detection is discovering the facilities provided by the browser environment in which the code is executing. This allows us to provide features that use those facilities in our code or to determine whether we need to provide a fallback.

The following code snippet shows a basic example of detecting the presence of a browser feature using object detection, to determine whether we should provide full application functionality or a reduced-experience fallback:

```
if (typeof document !== "undefined" &&
    (document.addEventListener || document.attachEvent) &&
    document.getElementsByTagName &&
    document.getElementById) {
  // We have enough of an API to work with to build our application
}
```

```
else {
  // Provide Fallback
}
```

Here, we test whether:

- The browser has a document loaded.
- The browser provides a means to bind event handlers.
- The browser can find elements given a tag name.
- The browser can find elements by ID.

Failing any of these tests causes us to resort to a fallback position. What is done in the fallback is up to the expectations of the consumers of the code, and the requirements placed upon the code. There are a couple of options that can be considered:

- We could perform further object detection to figure out how to provide a reduced experience that still uses some JavaScript.
- We could opt to not execute any JavaScript, falling back to the unscripted HTML on the page.
- We could redirect the user to a plainer version of the site. Google does this with Gmail, for example.

Because object detection has very little overhead associated with it (it's just a simple property/object lookup) and is relatively simple in its implementation, it makes for a good way to provide basic levels of fallback, both at the API and application levels. It's a good choice for the first line of defense in your reusable code authoring.

But what if our assumption about an API working correctly just because it *exists* proves to be overly optimistic? Let's see what we can do about that.

Feature simulation

Another means of dealing with regressions, and the most effective means of detecting fixes to browser bugs, is *feature simulation*. In contrast to object detection, which is simply an object/property lookup, feature simulation performs a complete run-through of a feature to make sure that it works as we'd expect it to.

While object detection is a good way to check that a feature *exists*, it doesn't guarantee that the feature will *behave* as intended. But if we know of specific bugs, we can quickly build tests to check when the feature bug is fixed as well as write code to work around the bug until that time.

As an example, Internet Explorer 8 and earlier versions will erroneously return both elements *and* comments if we execute `getElementsByTagName("*")`. No amount of object detection is going to determine if this will happen or not. As we hope often happens, this bug has been fixed by the Internet Explorer team in the IE 9 release of the browser.

Let's write a feature simulation to determine if the `getElementsByTagName()` method will work as we expect it to:

```
window.findByTagWorksAsExpected = (function(){
  var div = document.createElement("div");
  div.appendChild(document.createComment("test"));
  return div.getElementsByTagName("*").length === 0;
})();
```

In this example, we've written an immediate function that returns `true` if a call to `getElementsByTagName("*")` functions as expected and `false` otherwise. The steps of this test function are fairly simple:

- Create a detached `<div>` element.
- Add a comment node to the `<div>`.
- Call the function, see how many values are returned, and return `true` or `false` depending upon the result.

Well, knowing that there's a problem is only half the battle. What can we do with this knowledge to make things better for our code? The following listing shows a use of the preceding feature-simulation snippet in a useful context: working around the bug.

Listing 1 Putting feature simulation into practice to work around a browser bug

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 11.3</title>
    <script type="text/javascript" src="../scripts/assert.js"></script>
    <link href="../styles/assert.css" rel="stylesheet" type="text/css">
  </head>
  <body>

    <div><!-- comment #1--></div>
    <div><!-- comment #2--></div>

    <script type="text/javascript">

      function getAllElements(name) {

        if (!window.findByTagWorksAsExpected) {                  #1

          window.findByTagWorksAsExpected = (function(){         #2
            var div = document.createElement("div");            #2
            div.appendChild(document.createComment("test"));     #2
            return div.getElementsByTagName("*").length === 0;  #2
          })();                                                  #2
        }

        var allElements = document.getElementsByTagName('*');   #3

        if (!window.findByTagWorksAsExpected) {                   #4
          for (var n = allElements.length - 1; n >= 0; n--) {    #4
            if (allElements[n].nodeType === 1)                   #4
              allElements.splice(n,1);                           #4
          }                                                      #4
        }                                                        #4

        return allElements;

      }

      var elements = getAllElements();                           #5
      var elementCount = elements.length;                        #5

      for (var n = 0; n < elementCount; n++) {                   #6
        assert(elements[n].nodeType === 1,                       #6
               "Node is an element node");                       #6
      }                                                          #6

    </script>

  </body>
</html>
```

#1 Tests if we already know whether browser works as expected
#2 If not, determines if the feature works as expected in the browser or is broken
#3 Calls suspect feature and stores result
#4 Fixes things up if we know that browser is buggy
#5 Sets up for testing
#6 Tests feature with workaround

In this code, we set up some `<div>` elements containing comment nodes that we'll later use for testing. Then we get down to business with some script.

Because using `document.getElementsByTagName('*')` directly is suspect, we define an alternate method, `getAllElements()`, to use in its place. We want this method to just factor down into a call to `document.getElementsByTagName('*')` on browsers that implement it correctly, but to use a fallback that produces the correct results on browsers that don't.

The first thing that our method does is to use the immediate function that we developed previously to determine if the feature works as expected (#2). Note that we store the result in a window-scoped variable so that we can refer to it later, and we check to see if it's already been set, so that we only run the (relatively expensive) feature-simulation check once (#1).

After the check, we run the call to `document.getElementsByTagName('*')` and store the result in a variable (#3).

At this point, we have the node list of all elements, and we know whether we're operating in a browser that has the comment node problem or not. If we had determined that the problem exists, we run through the nodes, stripping out any that aren't element nodes (#4). This process is skipped for browsers that don't have that problem.

NOTE The `nodeType` of element nodes is `1`, while that of comment nodes is `8`. Modern browsers (including versions 8 and 9 of IE) define a set of constants on the `Node` object, such as `Node.ELEMENT_NODE` and `Node.COMMENT_NODE`. As our fix will be triggered in older browsers, we can't assume that these constants exist, so we've used hard-coded values. You can find a complete list of node type values at https://developer.mozilla.org/en/nodeType.

Finally, we test our new method by using it (#5) and asserting that the returned node list only contains element nodes (#6).

This example demonstrates how feature simulation works in two phases.

First, a simple test is run to determine if a feature works as we expect it to. It's important to verify the integrity of a feature (making sure it works correctly) rather than explicitly testing for the presence of a bug. While that may be a semantic distinction, it's one that's important to keep in mind.

Second, the results of the test are later used in our program to speed up looping through an array of elements. Because a browser that works correctly (one that returns only elements) doesn't need to perform the element checks on every stage of the loop, we can completely skip it and not pay any performance penalties in browsers that work correctly.

That's the most common idiom used in feature simulation: making sure a feature works as expected and providing fallback code in non-working browsers.

The most important point to take into consideration when using feature simulation is that it's a trade-off. Paying the extra performance overhead of the initial simulation, along with the extra lines of code in our programs, gives us the benefit of knowing that a suspect feature will work as expected in all supported browsers and makes our code immune to breaking upon future bug fixes. This immunity can be absolutely priceless when creating reusable code bases.

Feature simulation is great when we can test whether a browser is broken or not, but what can we do about browser problems that stubbornly resist being tested?

Untestable browser issues

Unfortunately, a number of problem areas in JavaScript and the DOM are either impossible or prohibitively expensive to test for. These situations fortunately are rather rare, but, when we encounter them, it always pays to spend some time investigating the matter to see if there's something we can do about it.

The following sections discuss some known issues that are impossible to test using any conventional JavaScript interactions.

Event handler bindings

One of the infuriating lapses in the browsers is the inability to determine if an event handler has been bound. The browsers don't provide any way of determining if any functions have been bound to an event listener on an element. Because of this, there's no way to remove all bound event handlers from an element unless we've maintained references to all bound handlers as we create them.

Event firing

Another aggravation is determining if an event will fire. While it's possible to determine if a browser supports a means of binding an event, it's *not* possible to know if a browser will actually fire an event. There are a couple places where this becomes problematic.

First, if a script is loaded dynamically after the page itself has already loaded, it may try to bind a listener to wait for the window to load when, in fact, that event already happened. As there's no way to determine if the event has already occurred, the code may wind up waiting forever to execute.

The second situation occurs if a script wishes to use custom events provided by a browser as an alternative. For example, Internet Explorer provides `mouseenter` and `mouseleave` events, which simplify the process of determining when a user's mouse enters or leaves an element's boundaries. These are frequently used as alternatives to the `mouseover` and `mouseout` events because they act slightly more intuitively than the standard events. But because there's no way of determining if these events will fire without first binding the events and waiting for some user interaction against them, it's hard to use them in reusable code.

CSS property effects

Yet another pain point is determining whether changing certain CSS properties actually affects the presentation. A number of CSS properties only affect the visual representation of the display and nothing else; they don't change surrounding elements or affect other properties on the element. Examples are `color`, `backgroundColor`, and `opacity`.

Because of this, there's no way to programmatically determine if changing these style properties will generate the effects desired. The only way to verify the impact is through a visual examination of the page.

Browser crashes

Testing script that causes the browser to crash is another annoyance. Code that causes a browser to crash is especially problematic because, unlike exceptions that can be easily caught and handled, these will always cause the browser to break.

For example, in older versions of Safari, creating a regular expression that uses Unicode character ranges would always cause the browser to crash, as in the following example:

```
new RegExp("[\\w\u0128-\uFFFF*_-]+");
```

The problem with this is that it's not possible to test using feature simulation because the test itself will always produce a crash in that older browser.

Additionally, bugs that cause crashes to occur forever become embroiled in difficulty because, while it may be acceptable for JavaScript to be disabled in some segment of the population using your browser, it's never acceptable to outright crash the browser of those users.

Incongruous APIs

Earlier we saw how jQuery decided to disallow the ability to change the `type` attribute in all browsers due to a bug in Internet Explorer. We *could* test this feature and only disable it in Internet Explorer, but that would set up an incongruity in which the API would work differently from browser to browser. In situations such as when a bug is so bad that it causes an API to break, the only option is to work around the affected area and provide a different solution.

In addition to impossible-to-test problems, there are issues that are *possible* to test, but that are prohibitively difficult to test effectively. Let's look at some of them.

API performance

Sometimes specific APIs are faster or slower in different browsers. When writing reusable and robust code, it's important to try to use the APIs that provide good performance. But it's not always obvious which API that is.

Effectively conducting performance analysis of a feature usually entails throwing a large amount of data at it, and that usually takes a relatively long time. Therefore, it's not something we can do in our code in the same way that we use feature simulation.

Ajax issues

Determining if Ajax requests work correctly is another thorn in our sides. Internet Explorer broke the ability to request local files via the `XMLHttpRequest` object in Internet Explorer 7. We could test to see if this bug has been fixed, but, to do so, we'd have to perform an extra request on every page load that attempted to perform a request. That's not optimal. And not only that, but an extra file would have to be included with the library for the sole reason of serving as a target for these extra requests. The overhead of both of these matters is prohibitive and would certainly not be worth the extra time and resources.

Untestable features are a significant hassle that hinders our writing of reusable JavaScript, but they can frequently be worked around with a bit of effort and cleverness. By utilizing alternative techniques, or constructing our APIs in such a manner as to obviate these issues in the first place, we'll likely be able to build effective code, despite the odds stacked against us.

Summary

Reusable cross-browser development involves juggling three factors:

- Code size—Keeping the file size small.
- Performance overhead—Keeping the performance level above a palatable minimum.
- API quality—Making sure that the APIs provided work uniformly across browsers.

There's no magic formula for determining the correct balance of these factors. The development factors are something that will have to be balanced by every developer in their individual development efforts. Thankfully, by using smart techniques like object detection and feature simulation it's possible to defend against the numerous directions from which reusable code will be attacked, without making any undue sacrifices.

Last updated: November 18, 2012