Debugging AJAX in Production

Lacking proper browser support, what steps can we take to debug production AJAX code?

Eric Schrock, Sun Microsystems

The JavaScript language has a curious history. What began as a simple tool to let Web developers add dynamic elements to otherwise static Web pages has since evolved into the core of a complex platform for delivering Web-based applications. In the early days, the language's ability to handle failure silently was seen as a benefit. If an image rollover failed, it was better to preserve a seamless Web experience than to present the user with unsightly error dialogs.

This tolerance of failure has become a central design principle of modern browsers, where errors are silently logged to a hidden error console. Even when users are aware of the console, they find only a modicum of information, under the assumption that scripts are small and a single message indicating file and line number should be sufficient to identify the source of a problem.

This assumption no longer holds true, however, as the proliferation of sophisticated AJAX applications has permanently changed the design center of the JavaScript environment.

Scripts are large and complex, spanning a multitude of files and making extensive use of asynchronous, dynamically instantiated functions. Now, at best, script execution failure results in an awkward experience. At worst, the application ceases to work or corrupts server-side state. Tacitly accepting script errors is no longer appropriate, nor is a one-line number and message sufficient to identify a failure in a complex AJAX application. Accordingly, the lack of robust error messages and native stack traces has become one of the major difficulties with AJAX development today.

The severity of the problem depends on the nature of the debugging environment. During development, engineers have nearly unlimited freedom. They can re-create problems at will, launch an interactive debugger, or quickly modify and deploy test code, providing the ability to form and test hypotheses rapidly in order to determine the root cause of a problem. Everything changes, however, once an application leaves this haven for the production environment. Problems can be impossible to reproduce outside the user's environment, and gaining access to a system for interactive debugging is often out of the question. Running test code, even without requiring downtime, can prove worse than the problem itself. For these environments, the ability to debug problems after the fact is a necessity. When a bug is encountered in production, enough information must be preserved such that the root cause can be accurately determined, and this information must be made available in a form that can be easily transported from the user to engineering.

Depending on the browser, JavaScript has a rich set of tools for identifying the bugs at the root of problems during the development phase. Tools such as Firebug, Venkman, and built-in DOM (document object model) inspectors are immensely valuable. As with most languages, however, things become more difficult in production. Ideally, we would like to be able to obtain a complete dump of the JavaScript execution context, but no browser can support such a feature in a safe or practical manner. This leaves error messages as our only hope. These error messages have to provide sufficient context to identify the root cause of an issue, and they need to be integrated into the application

experience such that the user can manage streams of errors and understand how to get the required information to developers for further analysis.

The first step in this process is to provide a means for displaying errors within the application. Although it is tempting simply to rely on alert() and its simple pop-up message, the visual experience associated with that is quite jarring. Large amounts of text do not scale well to pop-ups, and a flurry of such errors can require repeatedly dismissing the dialogs in rapid succession—sometimes making forward progress impossible. Many frameworks provide built-in consoles for this purpose, but a very simple hidden DOM element that allows us to expand, collapse, clear, and hide the console does the job nicely. With this integrated console, we can catch and display errors that would normally be lost to the browser error console. On most browsers, errors can be caught by a top-level window.onerror() handler that provides a browser-specific message, file, and line number.

Simply dumping these messages to a user-visible console represents a major step forward, but even an accurate message, file, and line number can be worthless when debugging a problem in an AJAX application. Unless the bug is a simple typo, we need to better understand the context in which the error was encountered.

Faced with an unexpected error, the next question is almost always: "How and why are we here?" If we're lucky, we can just look at the source code and make some educated guesses. The most common method of improving this process is through stack traces. The ability to generate stack traces is the hallmark of a robust programming environment, but unfortunately this is also one feature that is often overlooked. Stack traces are often viewed as too difficult to construct, too expensive to make available in production, or simply not worth the effort to implement. Because they are commonly viewed as something that's required only in exceptional circumstances, stack traces can often be expensive to calculate. As the complexity of a system grows and as asynchrony is employed to a larger extent, however, this view becomes less tenable. In a message-passing system, for example, the context in which the original message was enqueued is often more important than the context of the failure once the message has been dequeued. In an AJAX environment (where *asynchronous* was worthy of a spot in the acronym), the need for closures often makes the context in which they have been instantiated more useful than the closures themselves.

Sadly, JavaScript support for stack traces is sorely lacking. The browsers that do support stack traces make them available only via thrown exceptions, and most browsers don't provide them at all. Stack traces are never available within global handlers such as window.onerror(), as the arguments are defined by a DOM that optimizes for the lowest common denominator. A window.onexception() handler that's passed as an exception object would be a welcome addition. Instead, we're forced to catch all exceptions explicitly. On the surface, this seems like a daunting task—we don't want to wrap every piece of code in a try/catch block. In an AJAX application, however, all JavaScript code is executed in one of four contexts:

- Global context while loading scripts
- From an event handler in response to user interaction
- From a timeout or interval
- From a callback when processing an XMLHTTPRequest

The first case we must defer to window.onerror(), but since it happens while scripts are loading, it would be hard for such bugs to escape development. For the remaining cases, we can automatically wrap callbacks in try/catch blocks through our own registration function:

```
function mySetTimeout(callback, timeout)
{
        var wrapper = function () {
                try {
                        callback();
                } catch (e) {
                        myHandleException(e);
                }
        };

        return (setTimeout(wrapper, timeout));
}
```

For event listeners, things get slightly more complicated because the API requires the original function in order to remove the handler at a later point:

```
function myAddEventListener(obj, event, callback, capture)
{
        var wrapper = function (evt) {
                try {
                        callback(evt);
                } catch (e) {
                        myHandleException(e);
                }
        };
        if (!obj.listeners)
                obj.listeners = new Array();
        obj.listeners.push({
                event: event,
                wrapper: wrapper,
                capture: capture,
                callback: callback
        });
        obj.addEventListener(event, wrapper, capture);
}
```

Table 1 describes the information that is available from a global context and when catching particular types of exceptions for different browsers. The table demonstrates the limits of integrated browser support. Without reliable stack traces on every exception, we are forced to generate programmatic stack traces for better coverage. Thankfully, the semantics of the arguments object allows us to write a function to generate a programmatic stack trace:

```
function myStack()
{
        var caller, depth;
        var stack = new Array();
        for (caller = arguments.callee, depth = 0;
           caller && depth < 12;
           caller = caller.caller, depth++) {
                var args = new Array();
                for (var i = 0; i < caller.arguments.length; i++)
                        args.push(caller.arguments[i]);
                stack.push({
                   caller: caller,
                   args: args
                });
        }
        this.stack = stack;
}
```

Table 1 **Browser Support**

Browser	Event	Message	File	Line	Stack
Firefox 3.0.5	window.onerror	X	X[1]	X[1]	
	DOM exception	X	X	X	
	runtime exception	X	X	X	X
	user exception				X[2]
IE 7.0.5730.13	window.onerror	X	X	X	
	DOM exception	X			
	runtime exception	X			
Safari 3.2.1	window.onerror				
	DOM exception	X	X	X	
	runtime exception	X	X	X	
	user exception		X	X	
Chrome 1.0.154.36	window.onerror				
	DOM exception	X			
	runtime error	X			
	user exception				
Opera 9.63	window.onerror				
	DOM exception	X			X[3]
	runtime exception	X			X
	user exception				X[3]

1. DOM errors in Firefox do not have explicit file and line numbers, but the information is contained within the message.

2. Arbitrary exceptions do not have stack traces in Firefox, but those that use the Error() *constructor do.*

3. Opera can be configured to generate stack traces for exceptions, but it is not enabled by default.

A full implementation would provide a means for skipping uninteresting frames, including native stack traces (via a try/catch block), and providing a toString() method for converting the results. We don't have file and line numbers, but we do have function names and arguments. Sadly, the proliferation of anonymous functions in JavaScript makes it difficult to get the canonical name of a function. The toString() method can give us the source for a particular function, but when printing a stack trace we need a name. The only effective way to accomplish this is to search the global namespace of all objects while constructing a human-readable name for the function along the way. This seems expensive, but we need to print the stack trace only in case of error. Most functions are either in the global namespace, one level deep, or two levels deep in the prototype of a particular object. To get a function's name, we simply need to search the members of the window object, all of their children, and all children of their prototype objects. If we find a match, then we can construct the name using this lineage.

With the function name and the arguments, we can display a reasonable facsimile of a stack trace, even on browsers without native support for stack traces. One caveat, however, is that getting function names doesn't work with Internet Explorer 7. For reasons that are not well understood, global functions are not included when iterating over members of the window object.

Careful construction of the global exception handler allows us to handle both native browser and dynamically generated exceptions. Although having stack traces attached to our custom exceptions is useful, the true power of this mechanism is evident when dealing with asynchronous closures in a complex environment, particularly asynchronous XMLHTTPRequest objects. In a complicated AJAX application, all server activity must happen asynchronously; otherwise, the browser will hang while waiting for a response. A typical service model will do something like:

```
function dosomething(a, b)
{
        service.dosomething(a + b, function (ret, err) {
                if (err)
                        throw (err);
                process(ret);
        });
}
```

If an exception occurs in the process() function, then a wrapper embedded in the service implementation will catch the result and hand it off to our exception handler. But the stack trace will end at process(), when what we really want is the stack trace at the point when dosomething() was called. Because our stack traces are generated on demand and are cheap to assemble, we can achieve this by recording the stack trace before dispatching every asynchronous call and then chaining it to any caught exception. The global exception handler will print all members of the exception, displaying both stack traces in the process. Our core dispatch routine would look something like this:

```
function dispatch(func, args, callback)
        {
                var stack = new myStack();
                dodispatch(func, args, function (ret, err) {
                        try {
                                callback(ret, err);
                        } catch (e) {
                                e.linkedStack = stack;
                                myHandleException(e);
                        }
                });
        }
```

This allows transparent handling of server-side failures using the same exception handler. If an asynchronous closure generates an unanticipated exception, we can include the context in which the original XMLHTTPRequest was made.

By carefully following these design principles, we can construct an environment that dramatically improves our ability to debug issues by enabling users to provide developers with richer information that will allow for further analysis. Unfortunately, this environment is required to overcome the inadequacies of current JavaScript runtime environments. Without a single point to handle all uncaught exceptions, we are forced to wrap all callbacks in a try/catch block; and without reliable stack traces, we are forced to generate our own debugging infrastructure. It seems clear that a browser that implements these two features would soon become the preferred development environment for AJAX applications. Until that happens, careful design of the AJAX environment can still yield dramatic improvements in debuggability and serviceability for users of an application. Q

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ERIC SCHROCK has been a staff engineer at Sun Microsystems since 2003. After starting in the Solaris kernel group—where he worked on ZFS, among other things—Schrock spent the past few years helping to develop the Sun Storage 7000 series of appliances as part of the company's Fishworks engineering team.

The complete source code for the examples included here, as well as the latest version of the browser support table, can be found at http://blogs.sun.com/eschrock/resource/ajax/index.html.