CHAPTER 8

Balancing Performance with Software Engineering Best Practices and Running in Production

Chapter 7 explored ways to improve runtime performance. You quantified these improvements using the example perfLogger library and charted the results with R. That has been a theme throughout this book—measure and prove a point with data. If I had to choose a single sentence to serve as a thesis statement for this book, it would have to be something that I said in the first chapter that deserves repeating: Any journeyman can create something to spec, but a master crafts with excellence and proves that excellence with empirical data.

We've strived to do that so far throughout this book, creating our own tools to instrument our code and monitor the web performance of our web sites. We crafted data visualizations to prepare our data for easier consumption.

But this chapter is a little different. We will still look at raw data and performance optimizations, but the focus will be on balancing the need to optimize with other needs, like adhering to coding standards and best practices, readability, and making our code modular for use across a larger team.

We'll also take a closer look at how to generate test data at scale, either making our own test lab using virtual machines, or putting our code on a production web site to crowd-source the data.

Balancing Performance with Readability, Modularity, and Good Design

At the time of this writing the size of the group that I lead is roughly 20 to 25 engineers, managers, and engineering leads. That's a lot of hands to have making changes in just two to three code repositories. I track our performance like a hawk would track a field mouse. I chart out our web performance from WebPagetest for tens of URLs. I meet with the team regularly to discuss the output of these reports, going over our first view and repeat view data to make sure we are making efficient use of cache. We look at all of the aspects of performance and try to eke out as many optimizations as we can.

But there are other things that I track as well; among them are things like: What is our defect density? What is our incident rate for each product in production? Those things can be harmful to a product brand, arguably more than performance, depending on the severity of the issue.

In looking at things like defects and production incidents, one of the leading root causes, in my experience, is communication. Are the engineers talking to the QA staff updating them on features? Are the engineers talking with the production operations staff about how to support the features? And are the engineers talking with each other? But communication issues don't stop there. With literally millions of lines of code, does everyone know what all of the code does? Are libraries written to modularize functionality? Does everyone know about these libraries? If I were to read through a piece of code, would I know what it does and how to use it? How readable is the code?

When code is breaking in production, it is more important to me that all twenty of my engineers know how to use all of the code and functionality available to them than to wring out an extra millisecond or two of performance.

That's why we strive for modularity, reusability, and readability.

We try to practice modular code design, in that we try to write code in small self-contained and interchangeable modules. Writing code in modules minimizes the potential harm that can come from changes—because the modules communicate with each other via their interfaces, we can easily unit-test the interface and create integration tests around how they interact.

By striving for reusability we reduce the chances of creating new bugs. Ideally, the code that we are reusing has been tested and proven already.

Making our code readable means we try to make it obvious what our code does. This includes

- Abstracting complex logic into clearly and meaningfully named functions or self-contained objects or modules (it's all circular).
- Using consistent formatting that we have all agreed upon as our standard.
- Using good and clear naming patterns for our variable names.

While all of those are good practices, they generally also work counter to having the leanest, most performant code humanly possible. Long, meaningful variable names take up characters that add to file size, which increases the payload of a page. The same principle applies to the extra lines of code needed to write functions and constructors, not to mention the extra overhead for the interpreter of creating these objects in the heap, managing their garbage collection, and traversing their scope chain.

But having our code in objects and functions abstracts our logic to meaningfully named, atomic pieces that can be updated and maintained without having too much of an impact on the rest of the system.

It's all about perspective and finding balance. That's part of what we will talk about this chapter.

Scorched-Earth Performance

In earlier chapters I've mentioned the term *scorched-earth performance*. That's a term I've coined that indicates that we have sacrificed all else in the ultimate pursuit of performance. In this section we look at some scorched-earth practices, and we quantify the benefit, but also discuss the cost involved to give the full picture.

Inlining Functions

Let's first look at the run-time performance benefit that we get from inlining functions. In past chapters we've looked at the overhead cost of having and traversing memory structures. Last chapter we talked about this in the context of differing memory scopes, but the same concept applies to object hierarchies we create.

Ostensibly there is a runtime performance boost that we can gain by coalescing all of our functionality into a single function, instead of abstracting out functionality into separate functions or even objects. Let's look at this.

In the next example you'll create a single page where you will benchmark the results of coalescing functionality into a single function, breaking the code into different functions, and creating objects to contain functionality. Let's get started!

Creating the Example

First create a new page with the basic skeletal HTML structure and include the `perfLogger.js` library:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Methodology Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
</body>
</html>
```

Next you'll create a script tag in the body of the page and create a function that will combine everything that we want to do. Call the function `unwoundfunction()`:

```
<script>
function unwoundfunction(){

}
</script>
```

Within `unwoundfunction()` you'll create a variable named `sum`, iterate through a `for` loop 300 times and sum up the incremental value of each step in the loop:

```
var sum = 0;
for(var x = 0; x < 300; x++){
     sum += x;
}
```

Then you will create a variable named `average`, iterate 300 times, sum up the incrementor, and divide the sum by 300. This gives you two operations to calculate—summing up series of numbers and finding an average.

```
var average = 0;
for(var x = 0; x < 300; x++){
     average += x;
}
average = average/300;
```

The completed function should look like the following. It is this function that you will benchmark to get the time for coalescing functionality:

```
function unwoundfunction(){
     var sum = 0;
     for(var x = 0; x < 300; x++){
          sum += x;
     }

     var avgerage = 0;
     for(var x = 0; x < 300; x++){
          avgerage += x;
     }
     avgerage = avgerage/300;
}
```

Next create two new functions, one to handle summing the numbers and the other to handle the averaging of the result:

```
function getAvg(p){
     var avg = 0;
     for(var x = 0; x < p; x++){
          avg += x;
     }
     return(avg/p);
}

function getSum(a){
     var sum = 0;
     for(var x = 0; x < a; x++){
          sum += x;
     }
     return(sum);

}
```

Next create a third function that will invoke `getSum()` and `getAvg()`. You'll benchmark this function as an example of using functions:

```
function usingfunctions(){
     var average = getAvg(300);
     var sum = getSum(300)
}
```

Now create an object constructor to handle this functionality. You can call this object `simpleMath` and give it two public methods, `sum()` and `avg()`:

```
function simpleMath(){
     this.sum = function(a){
          var sum = 0;
          for(var x = 0; x < a; x++){
               sum += x;
          }
          return(sum);
     }
```

```
     this.avg = function(p){
          var avg = 0;
          for(var x = 0; x < p; x++){
               avg += x;
          }
          return(avg/p);
     }
}
```

Then create a function called `usingobjects` that will instantiate a new `simpleMath` object and call the `sum` and `avg` methods. You will benchmark this function to get the metrics for using objects.

```
function usingobjects(){
     var m = new simpleMath();
     var average = m.avg(300);
     var sum = m.sum(300);
}
```

And finally you'll benchmark these functions, having `perfLogger` execute each function 100 times:

```
perfLogger.logBenchmark("UsingObjects", 100, usingobjects, true, true);
perfLogger.logBenchmark("UsingFunctions", 100, usingfunctions, true, true);
perfLogger.logBenchmark("unwoundfunction", 100, unwoundfunction, true, true);
```

The complete test page should look like this:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Methodology Comparison</title>
<script src="/lab/perfLogger.js"></script>
<script>
function getAvg(p){
     var avg = 0;
     for(var x = 0; x < p; x++){
          avg += x;
     }
     return(avg/p);
}

function getSum(a){
     var sum = 0;
     for(var x = 0; x < a; x++){
          sum += x;
     }
     return(sum);

}


function simpleMath(){
     this.sum = function(a){
```

```
            var sum = 0;
            for(var x = 0; x < a; x++){
                sum += x;
            }
            return(sum);
        }

        this.avg = function(p){
            var avg = 0;
            for(var x = 0; x < p; x++){
                avg += x;
            }
            return(avg/p);
        }
    }
    </script>
    </head>
    <body>
    <script>

    function usingfunctions(){
        var average = getAvg(300);
        var sum = getSum(300)
    }

    function usingobjects(){
        var m = new simpleMath();
        var average = m.avg(300);
        var sum = m.sum(300);
    }

    function unwoundfunction(){
        var sum = 0;
        for(var x = 0; x < 300; x++){
            sum += x;
        }

        var average = 0;
        for(var x = 0; x < 300; x++){
            average += x;
        }
        average = average/300;
    }


    perfLogger.logBenchmark("UsingObjects", 100, usingobjects, true, true);
    perfLogger.logBenchmark("UsingFunctions", 100, usingfunctions, true, true);
    perfLogger.logBenchmark("unwoundfunction", 100, unwoundfunction, true, true);

    </script>
    </body>
```

```
</html>
```

Viewing this page in a browser you should see something like the following results:

```
benchmarking function usingobjects() { var m = new simpleMath; var average = m.avg(300); var sum
= m.sum(300); }
average run time: 0.025260580000000914ms
path: http://tom-barker.com/lab/useFunctions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:16.0) Gecko/16.0 Firefox/16.0

benchmarking function usingfunctions() { var average = getAvg(300); var sum = getSum(300); }
average run time: 0.020855050000000687ms
path: http://tom-barker.com/lab/useFunctions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:16.0) Gecko/16.0 Firefox/16.0

benchmarking function unwoundfunction() { var sum = 0; for (var x = 0; x < 300; x++) { sum += x;
} var avgerage = 0; for (var x = 0; x < 300; x++) { avgerage += x; } avgerage = avgerage / 300;
}
average run time: 0.01648929999999666ms
path: http://tom-barker.com/lab/useFunctions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:16.0) Gecko/16.0 Firefox/16.0
```

Once again you'll put this either in production or in a test lab to get traffic pointed at the code to give a nice breadth of results in the log file.

Let's grab these results and chart them in R!

Analyzing Results

For charting you can reuse the `PlotResultsofTestsByBrowser()` function from the last chapter, and pass in the ID of each test. This will create the chart shown in Figure 8-1.

```
PlotResultsofTestsByBrowser(c("unwoundfunction", "UsingFunctions", "UsingObjects"),
c("Firefox"), "Comparison of average benchmark time \nfor coding methodology \nin milliseconds")
```

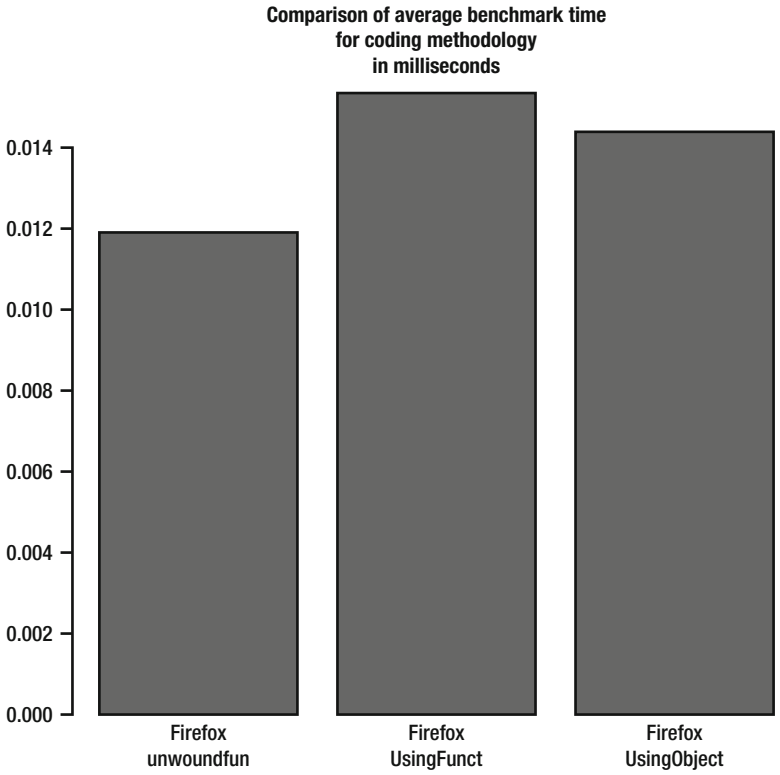

Figure 8-1. *Comparison of runtime performance for coalescing functionality, using functions, and using objects*

So you can see that there is a performance increase by stripping out all overhead and writing the code as line-by-line imperative statements. In this simple example the differences are less than a millisecond in scope, but the percentages are significant. From the smallest to the largest there is a 23% improvement in performance for coalescing functionality compared to using functions, and an 18% improvement in performance for coalescing functionality over using objects. In situations where performance is everything, as in financial transactions, this is a significant difference.

But splitting our functionality into functions makes our code much more readable. It's fairly obvious what code like `average = getAvg` or `sum = getSum` does.

Creating objects takes that improvement even further. You can reuse objects between projects, pass the objects between applications, extend the objects into new ones, or decorate the prototype chain, to reuse functionality.

In most cases the extra overhead is worth the reusability and readability gains.

Closure Compiler

Another case of scorched-earth performance is what Google's Closure Compiler does to JavaScript when using Advanced mode. I touched a little on Closure Compiler back in Chapter 2, but let's now look at a fleshed-out example.

Closure Compiler can be run in either of two modes:

- In Simple mode it mostly performs like most other minifiers, removing whitespace, line breaks, and comments
- In Advanced mode it rewrites the JavaScript by renaming variables and functions from longer descriptive names to single letters to save file size, and it inlines functions, coalescing them into single functions wherever it determines that it can.

It is Advanced mode that I would consider scorched-earth. Let's take a look at an example.

Creating an Example

First create a baseline file called `benchmarkobjects.html`. On this page you will create two objects, a `user` object and a `video` object. The user will be able to add video to their favorites list. You'll exercise this ability in a function and benchmark that function.

Start with the familiar basic skeletal HTML structure and include the `perfLogger` library:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Loop Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
</body>
</html>
```

In the body, create a `script` tag and start making object constructors. First create the constructor for the `video` object; it will accept a parameter that becomes the video title, and it has a public method called `printInfo()` that simply returns the video title.

```
<script>
function video(title){
          this.title = title;
          this.printInfo = function(){
               return this.title;
          }

     }
</script>
```

Next create the constructor for the `user` object. The `user` object accepts a parameter that is set as the user name, and it has two public methods: `addToFavorite()`, which pushes the passed-in object into the user's `favoriteList`, and `showFavorites()`, which loops through the `favoriteList`. The `user` object then console-logs the return value from calling `printInfo` on each video in the `favoriteList`:

```
function user(uname){
     this.username = uname;
     this.favoriteList = [];
     this.addToFavorite = function(a){
          this.favoriteList[this.favoriteList.length] = a;
     }
```

```
        this.showFavorites = function(){
            for(var f = 0; f < this.favoriteList.length; f++){
                var t = this.favoriteList[f].printInfo();
                console.log(t);
            }
        }
}
```

Finally, create a function that will exercise the functionality you just created and benchmark that function. It will create a `new user` object and iterate 20 times, creating a `new video` object each step and adding that `new video` to the user's `favoriteList`:

```
function testUserObject(){
    var u1 = new user("tom");
    for(var i = 0; i < 20; i++){
        u1.addToFavorite(new video("video "+ i));
    }
    u1.showFavorites();
}

perfLogger.logBenchmark("benchmarkObject", 10, testUserObject, true, true);
```

Your completed page should look like this:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Loop Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
    <script>
    function user(uname){
        this.username = uname;
        this.favoriteList = [];
        this.addToFavorite = function(a){
            this.favoriteList[this.favoriteList.length] = a;
        }

        this.showFavorites = function(){
            for(var f = 0; f < this.favoriteList.length; f++){
                var t = this.favoriteList[f].printInfo();
                console.log(t);
            }
        }
    }

    function video(title){
        this.title = title;
        this.printInfo = function(){
            return this.title;
```

```
        }

    }

    function testUserObject(){
        var u1 = new user("tom");
        for(var i = 0; i < 20; i++){
            u1.addToFavorite(new video("video "+ i));
        }
        u1.showFavorites();
    }

    perfLogger.logBenchmark("benchmarkObject", 10, testUserObject, true, true);
    </script>
</body>
</html>
```

And when you look at the page in a browser, you should see something like the following:

```
benchmarking function testUserObject(){ var u1 = new user("tom"); for(var i = 0; i < 20; i++){
u1.addToFavorite(new video("video "+ i)); } u1.showFavorites();      }
average run time: 0.6119000026956201ms
path: http://tom-barker.com/lab/benchmarkobjects.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

Run Through Closure Compiler

Now you are ready to run the code through Closure Compiler. The easiest way to do that is to use the Closure Compiler UI, accessible here: `http://closure-compiler.appspot.com/home`.

Closure Compiler UI is a web application (see Figure 8-2). On the left you enter the JavaScript and choose from a number of options, and on the right is the output of Closure Compiler.

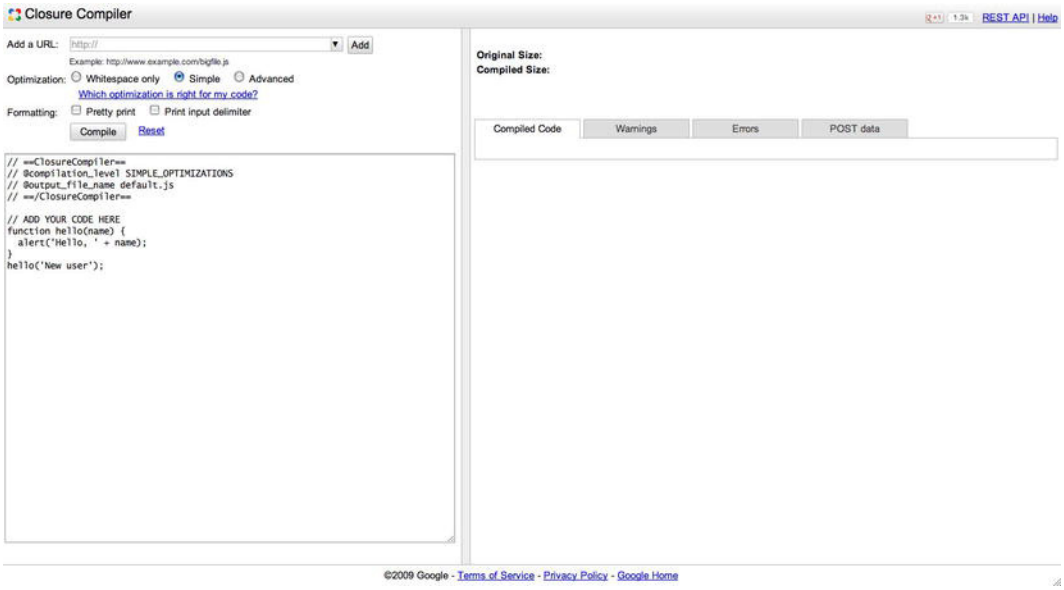

Figure 8-2. *The Closure Compiler UI*

The options on the top left are:

- A text box where you can enter the URL of a remote JavaScript file to be included in the compilation. To use this, simply type in the URL and click the Add button. You'll see the URL reflected in the large text area in the bottom, like so:

```
// @code_url http://tom-barker.com/lib/perfLogger.js
```

- A series of radio buttons that indicate the mode that Closure Compiler should run in, either Whitespace Only, Simple, or Advanced. Whitespace Only does just what it sounds like; it removes comments, line breaks, and unneeded whitespace. Simple compilation removes whitespace, line breaks, and comments but it also renames local variables to use smaller names. As you've already seen, Advanced compilation completely rewrites the JavaScript.
- Your choice of formatting. Pretty Print includes line breaks and indents for easier reading, and Print Input Delimiter allows you to pass in a string that will function as boundaries between blocks of passed-in code—if we pass in multiple remote files, the input delimiter will print (in comments) between the code from each file so that we can tell which code block came from which file.
- A Compile button, and finally a large text area where you can enter your options and any additional code you want to compile.

On the right side is a large text area where the compiled code is output. There are also tabs to see any warnings or errors that were generated during compilation.

For this test if you include `perfLogger.js` as a file include and compile it, you get a JavaScript error when trying to use the results because Closure Compiler has renamed the shim for `performance.now();` see Figure 8-3.

Figure 8-3. *JavaScript error thrown when running perfLogger through Closure Compiler Advanced Compilation*

So to make the test work you can just copy and paste the contents of `perfLogger` into the text area on the right side, and change the `performance.now` references to `Date.now()`.Then copy the contents of the `script` tag from `benchmarkobjects.html` into the text area below the contents of `perfLogger`. See Figure 8-4.

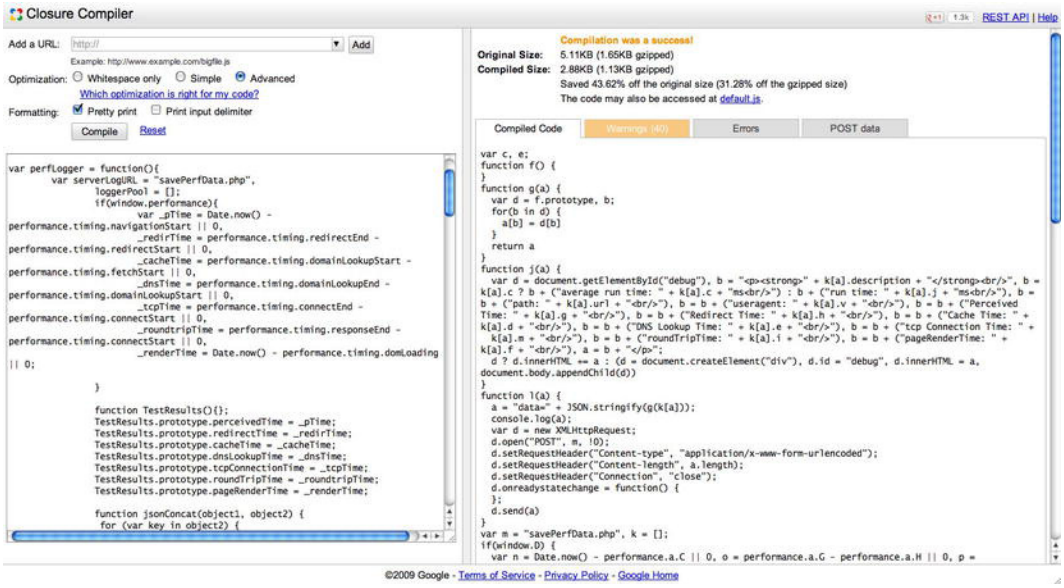

Figure 8-4. *Running perfLogger through Closure Compiler UI*

Then create a new page with just the basic HTML skeletal structure, put a `script` tag in the body, and copy and paste the compiled JavaScript into the `script` tag. The test file should look like the following:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Closure Compiler Benchmark</title>
</head>
<body>
<script>
var c, e;
function f() {
}
function g(a) {
```

```
 var d = f.prototype, b;
 for(b in d) {
   a[b] = d[b]
 }
 return a
}
function j(a) {
 var d = document.getElementById("debug"), b = "<p><strong>" + k[a].description + "</
strong><br/>", b = k[a].c ? b + ("average run time: " + k[a].c + "ms<br/>") : b + ("run time: "
+ k[a].j + "ms<br/>"), b = b + ("path: " + k[a].url + "<br/>"), b = b + ("useragent: " + k[a].v
+ "<br/>"), b = b + ("Perceived Time: " + k[a].g + "<br/>"), b = b + ("Redirect Time: " + k[a].h
+ "<br/>"), b = b + ("Cache Time: " + k[a].d + "<br/>"), b = b + ("DNS Lookup Time: " + k[a].e +
"<br/>"), b = b + ("tcp Connection Time: " +
 k[a].m + "<br/>"), b = b + ("roundTripTime: " + k[a].i + "<br/>"), b = b + ("pageRenderTime: "
+ k[a].f + "<br/>"), a = b + "</p>";
 d ? d.innerHTML += a : (d = document.createElement("div"), d.id = "debug", d.innerHTML = a,
document.body.appendChild(d))
}
function l(a) {
 a = "data=" + JSON.stringify(g(k[a]));
 console.log(a);
 var d = new XMLHttpRequest;
 d.open("POST", m, !0);
 d.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
 d.setRequestHeader("Content-length", a.length);
 d.setRequestHeader("Connection", "close");
 d.onreadystatechange = function() {
 };
 d.send(a)
}
var m = "savePerfData.php", k = [];
if(window.D) {
 var n = Date.now() - performance.a.C || 0, o = performance.a.G - performance.a.H || 0, p =
performance.a.p - performance.a.B || 0, q = performance.a.A - performance.a.p || 0, r =
performance.a.w - performance.a.o || 0, t = performance.a.I - performance.a.o || 0, u = Date.
now() - performance.a.z || 0
}
c = f.prototype;
c.g = n;
c.h = o;
c.d = p;
c.e = q;
c.m = r;
c.i = t;
c.f = u;
e = {k:function(a, d, b, h) {
 k[a] = new f;
 k[a].id = a;
 k[a].startTime = Date.now();
 k[a].description = d;
 k[a].q = b;
```

```
 k[a].s = h
}, l:function(a) {
 k[a].u = Date.now();
 k[a].j = k[a].u - k[a].startTime;
 k[a].url = window.location.href;
 k[a].v = navigator.userAgent;
 k[a].q && j(a);
 k[a].s && l(a)
}, r:function(a, d, b, h, v) {
 for(var i = 0, s = 0;s < d;s++) {
   e.k(a, "benchmarking " + b, !1, !1), b(), e.l(a), i += k[a].j
 }
 k[a].c = i / d;
 h && j(a);
 v && l(a)
}, g:function() {
 return n
}, h:function() {
 o
}, d:function() {
 return p
}, e:function() {
 return q
}, m:function() {
 return r
}, i:function() {
 return t
}, f:function() {
 return u
}, J:function() {
 this.k("no_id", "draw perf data to page", !0, !0);
 this.l("no_id")
}};
function w() {
 this.K = "tom";
 this.b = [];
 this.n = function(a) {
   this.b[this.b.length] = a
 };
 this.t = function() {
   for(var a = 0;a < this.b.length;a++) {
     console.log(this.b[a].title)
   }
 }
}
function x(a) {
 this.title = a;
 this.F = function() {
   return this.title
 }
}
```

```
e.r("benchmarkClosureCompiler", 10, function() {
 for(var a = new w, d = 0;20 > d;d++) {
   a.n(new x("video " + d))
 }
 a.t()
}, !0, !0);
</script>
</body>
</html>
```

If you view this in a browser you should see the following.

```
benchmarking function () { for(var a = new w, d = 0;20 > d;d++) { a.n(new x("video " + d)) }
a.t() }
average run time: 0.9ms
path: http://tom-barker.com/lab/benchmarkclosurecompiler.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

When checking the log file, you can see that the Closure Compiler–rewritten code doesn't quite save all of the fields to the log file. This is because Closure Compiler renamed most of the variables, including runtime. If you `console.log` the serialized data, you can see that the data being posted looks like this:

```
data={"id":"benchmarkClosureCompiler","startTime":1344114475936,"description":"benchmarking
function () {\n    for (var a = new w, d = 0; 20 > d; d++) {\n        a.n(new x(\"video \" +
d));\n    }\n    a.t();\n}","q":false,"s":false,"u":1344114475943,"j":7,"url":"http://tom-
barker.com/lab/benchmarkclosurecompiler.html","v":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5;
rv:16.0) Gecko/16.0 Firefox/16.0","c":18.6}
```

The runtime variable is that little `"c"` value at the end. Good luck trying to parse that out of the stew that the compiled source code is now. To be fair, there are ways to preserve property names, like using quoted string property names—for example by using `testResult["runtime"]` instead of `obj.runTime`. For more information about this, see Google's documentation here: `https://developers.google.com/closure/compiler/docs/api-tutorial3`.

When you pass the data back to `savePerfData.php`, that code is expecting a variable `runTime` or `avgRunTime`, not `c`, so runtime data is never retrieved.

But that's OK; the benefit we are interested in here is in web performance, so we'll compare the two pages in WebPagetest.

Compare and Analyze

Let's go to `webpagetest.com` and run tests for both of our URLs. The following table has the URLs tested and the test result URLs for the tests that I ran.

URL to Test	Test Result URL
`tom-barker.com/lab/benchmarkobjects.html`	`http://www.webpagetest.org/result/120803_WS_ad105844519fccc308dd9f678bc0caae/`
`tom-barker.com/lab/benchmarkclosurecompiler.html`	`http://www.webpagetest.org/result/120803_B0_20df854313c5101e6339e24bb0d958ec/`

The summary results are shown in Figures 8-5 and 8-6.

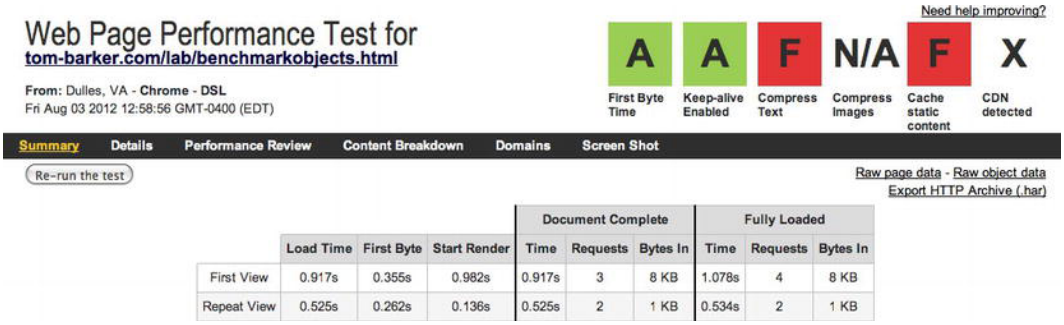

Figure 8-5. *Summary Web Performance Results for benchmarkobject.html*

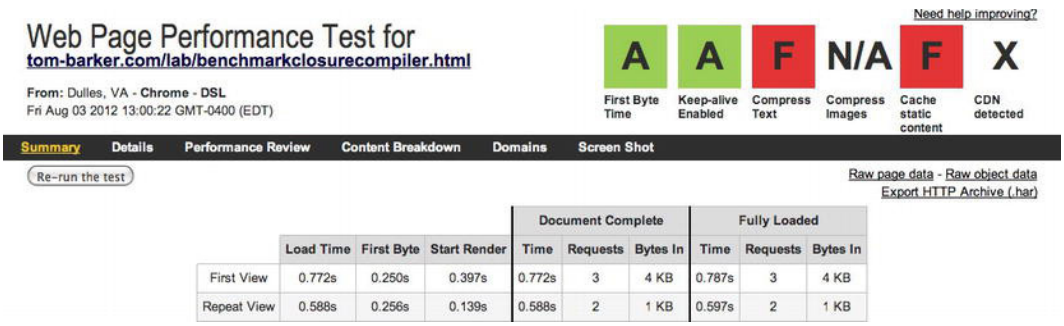

Figure 8-6. *Summary Web Performance Results for benchmarkclosurecompiler.html*

From the result screens just shown, you can see that the Closure Compiler–generated test has a load time that is 200 milliseconds faster, a first byte time that is 100 milliseconds faster, a start render time that is almost 600 milliseconds faster, a document complete time that is 145 milliseconds faster, and a fully loaded time that is 291 milliseconds faster. Clearly there are significant gains to be had by using Closure Compiler's Advanced mode.

But at this point you should also be able to see the downside. It was necessary to alter the original code just for the compiled code to work in the browser without generating errors. Once it was working in a browser, you saw that some of the hooks into the back-end stopped working.

And all of this is just a small test example, very self-contained. Imagine if we had third-party ad code embedded in the page. Imagine if we interacted with plug-ins with our JavaScript.

Now imagine adding new features to our original code and doing this all over again. Every week. With 20 people having their hands in the code base.

We've had to alter our workflow and introduce at least one extra debugging step, testing that everything actually works after compilation. We've added a level of complexity, a new breakpoint for our code to stop working. And debugging at this level, after everything has been obfuscated, is degrees of magnitude harder than debugging our own native code that we have already written.

Are the gains in performance that we see worth the extra effort and additional level of complexity that would be entailed in maintaining and updating compiled code?

Next Steps: From Practice to Practical Application

Throughout the book so far we have been creating tests and talking about and looking at data that is being generated from these tests at scale. We now look at the tactics of doing this on the job.

Monitoring Web Performance

This is fairly straightforward. You'll just need to choose a number of URLs that you want to monitor, plug them into WPTRunner, and begin tracking those URLs over time.

As you gather data you should review that data with your team. Identify areas for improvement—are your images not optimized, is your content not gzipped, how can you minimize HTTP requests? Set performance goals and begin working toward those goals with your team.

Instrumenting Your Site

The next thing you want to do, if you aren't doing so already, is to instrument your site—to put benchmarking code live in production to gather real live performance data for your pages that are already out in the wild.

To do this, you just need to choose what pages you want to monitor, choose a set of metrics that you want to gather—maybe the perceived load time of the page, maybe the runtime of the more complex pieces of functionality—and integrate perfLogger into those pages to capture that data. Note that you probably don't want to use the benchmark feature of perfLogger, since that will impact the performance of the pages, but rather use the `startTimeLogging` and `stopTimeLogging` functions to capture timing information.

I do this on my own site. In Figure 8-7 you can see a screenshot of my home page with perfLogger debug information on the far right side of the page.

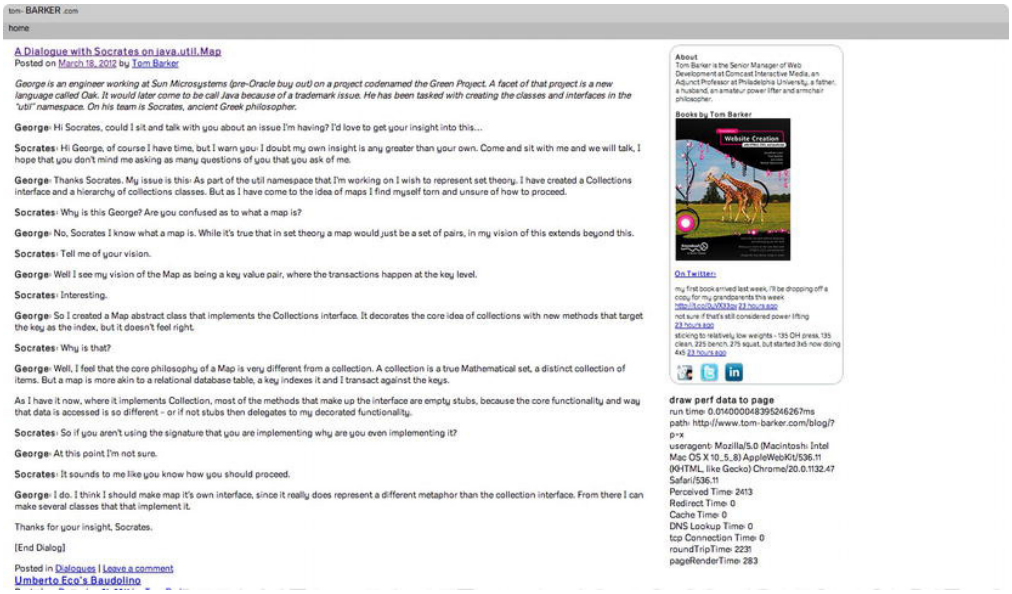

Figure 8-7. *The tom-barker.com site with performance data drawn to the screen*

The benefit of putting instrumentation on our live sites is that we get real live data from our actual users. We track this data over time and look for issues as conditions change—new browsers and browser versions get introduced, new features get promoted to production, and so on. Issues can appear as spikes in performance numbers to indicate that something is suddenly running much slower, or even unexpected drops in performance numbers, which could indicate that your site might be unavailable to your users.

Instrumenting your site is something that is done as regular maintenance of your site.

Benchmark in Your Test Lab

So we instrument our code in production and we monitor our web performance regularly, but how do we make sure our code is performant before we release it? We benchmark in a test lab.

A test lab can be a physical lab full of workstations, or it can be one or two machines with virtual machines running on them.

If you have the scale, budget, and staff for a physical test lab, then that's awesome! That's how it's done for real world-class web applications. But for 7 out of the last 12 companies that I've worked at, that was a pipe dream. My developers and I would have to test and certify our own code on our own machines. And in some cases that may be all that you need.

In either case your first order of business is to establish a browser support matrix—how else can you know what browsers and clients to assemble if you don't have a clear list of what you will support. At the bare minimum a browser support matrix is a list of browsers and browser versions and what level of support you will provide for those browsers. In my own experience there are generally three levels of support—will we provide support to an end user using this browser (Support in Production), should our QA team test with this browser in their regular testing (Test in QA), and should our engineers be conducting developer testing with these browsers, at least making sure that features function in these browsers (Developer Testing)? See Figure 8-8 for an example of this sort of browser matrix.

Browser	Support in Production	Test in QA	Developer Testing
IE 10	N	Y	Y
IE 9	Y	Y	Y
IE 8	Y	Y	N
IE 7	Y	Y	N
Chrome 21b	N	Y	Y
Chrome 20	Y	Y	Y
Chrome 19	Y	Y	N
Firefox Aurora	N	N	Y
Firefox Beta	N	Y	Y
Firefox 14	Y	Y	Y
Firefox 13	Y	Y	N
Safari 5.5	Y	Y	Y
Safari 5.0x	N	Y	N

Figure 8-8. *A bare-minimum browser support matrix*

Ideally and eventually, though, your browser support matrix should include things like plug-ins, as well as a breakdown of features, because not every feature may work in every browser. See Figure 8-9 for a more robust browser support matrix.

Browser	Support in Production	Test in QA	Developer Testing	Expandable Left Nav	Content Reflow	Ad Break Out	On Load Fade
iOS 5.0	Y	Y	Y	Y	Y	Y	Y
iOS 4.3	Y	Y	N	Y	Y	Y	Y
Android 4.0	Y	Y	Y	Y	Y	Y	Y
Android 3.1	Y	Y	N	Y	Y	Y	Y
IE 10	N	Y	Y	Y	Y	Y	Y
IE 9	Y	Y	Y	Y	Y	Y	Y
IE 8	Y	Y	N	Y	N	Y	N
IE 7	Y	Y	N	N	N	Y	N
Chrome 21b	N	N	Y	Y	Y	Y	Y
Chrome 20	Y	Y	Y	Y	Y	Y	Y
Chrome 19	Y	Y	N	Y	Y	Y	Y
Firefox Aurora	N	N	Y	Y	Y	Y	Y
Firefox Beta	N	N	Y	Y	Y	Y	Y
Firefox 14	Y	Y	Y	Y	Y	Y	Y
Firefox 13	Y	Y	N	Y	Y	Y	Y
Safari 5.5	Y	Y	Y	Y	Y	Y	Y
Flash 11	Y	Y	Y	N/A	N/A	N/A	N/A
Flash 10	Y	Y	N	N/A	N/A	N/A	N/A
Silverlight 5	Y	Y	Y	N/A	N/A	N/A	N/A
Silverlight 4	Y	Y	N	N/A	N/A	N/A	N/A

Figure 8-9. *A more detailed browser support matrix*

The way you start to choose your browser matrix is by looking at your logs—what browsers are most used by your clients? Make sure you take into account at least the most active browsers, which won't always be the most attractive browsers. (How many years did you need to support IE 6 just because your user base was locked into it because of corporate upgrade policy?) But don't assume that because a certain set of browsers are being used to visit your site, you only need to focus on those browsers. It may just be that your site only works best on those browsers. Also make sure you include beta and earlier browsers in your matrix as well, so that you can code for the future.

Once you have your browser support matrix, you can begin gathering workstations or virtual machines to test on those. Even if you have a QA staff that handles testing, as web developers the onus is on us to make sure that what we create is functional and in a good state before handing off to QA. That includes benchmarking our code against our browser matrix.

If you are going to use virtual machines (VMs), my favorite option is to use Virtual Box from Oracle, available at `https://www.virtualbox.org/`. It's a completely free, open source, lightweight, professional-level solution for running VMs. See Figure 8-10 for the Virtual Box homepage.

Figure 8-10. *The homepage for Virtual Box*

You simply go to the download section and choose the correct binary for your native operating system. See Figure 8-11 for the Virtual Box download page.

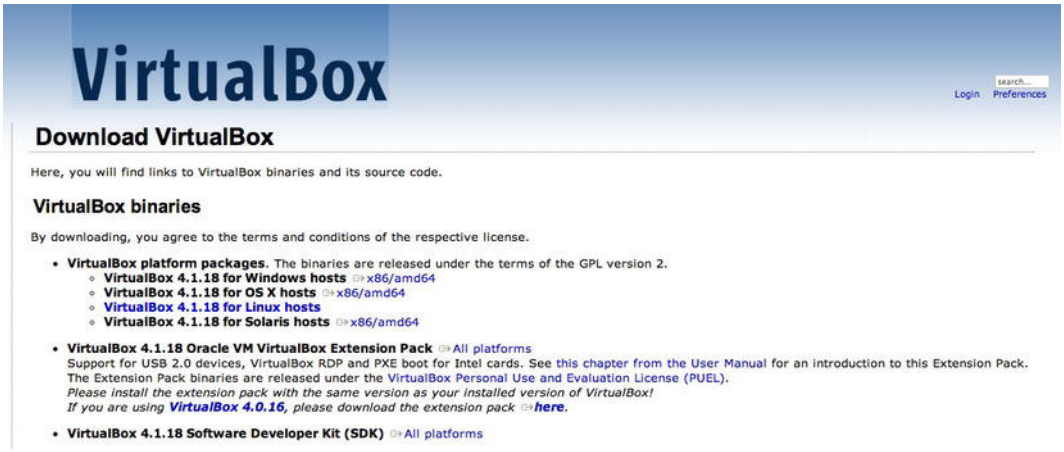

Figure 8-11. *The Virtual Box download page*

Once you've downloaded and installed Virtual Box, you can simply add new virtual machines by following the instructions in the application. Note that you'll need the install disk or the disk images for each operating system that you want to run. Once you have all of your VMs set up, your Virtual Box installation should look something like Figure 8-12.

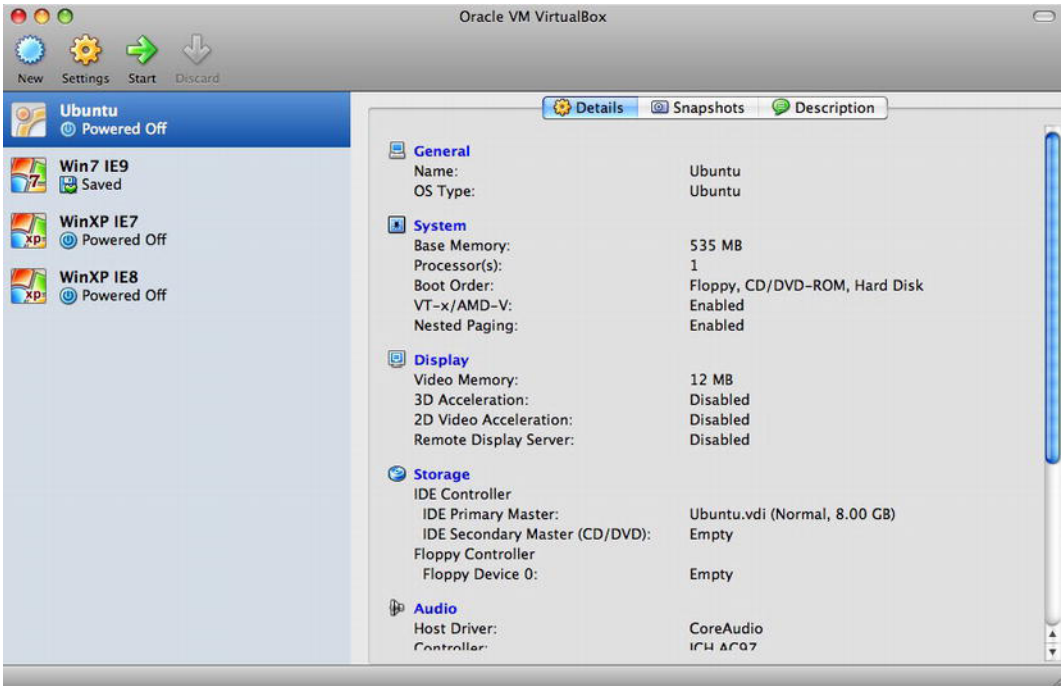

Figure 8-12. *Virtual Box with multiple VMs*

In this post-PC era don't forget to include mobile browsers in your matrix. Your best bet is, of course, to have devices on hand to test on, but barring that you can run an emulator/simulator on your laptop or use a third party like Keynote Device Anywhere that can make available a complete test center full of devices, available for manual or scripted testing remotely. More information about Keynote Device Anywhere can be found at their website, `http://www.keynotedeviceanywhere.com/`.

The iOS simulator comes bundled in with XCode, but getting and installing the Android emulator it is a bit more involved. You must first download the Android SDK from `http://developer.android.com/sdk/index.html`. Figure 8-13 shows the download page.

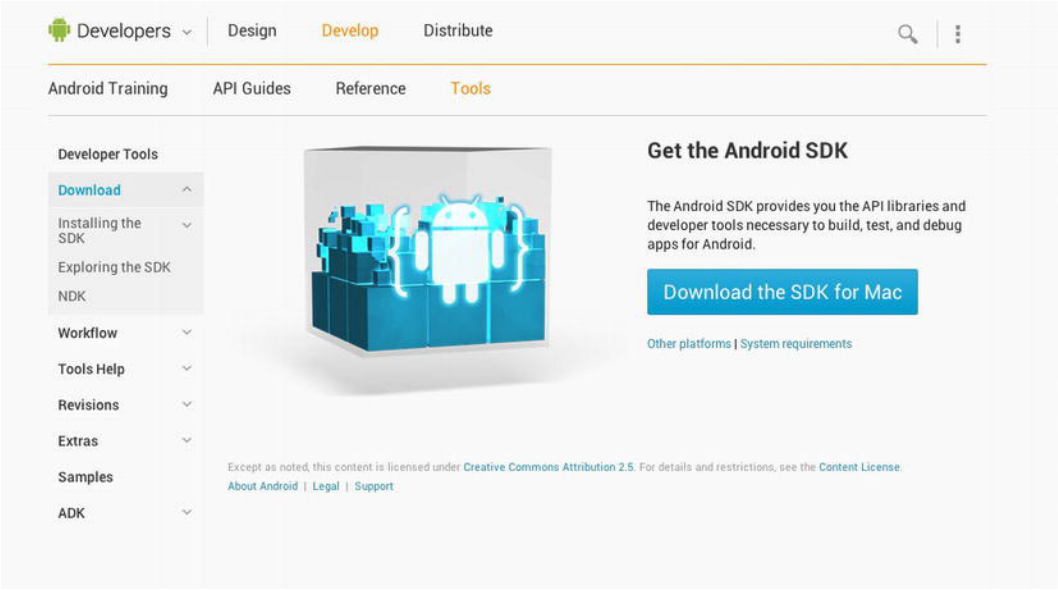

Figure 8-13. *Android SDK download page*

Once it's downloaded you'll need to expand the compressed file and navigate to the `tools` directory to get to the SDK Manager. On a Windows machine you can simply double-click the SDK Manager executable in the `tools` directory. On a Mac or Linux box you need to go into Terminal, `cd` to the directory, and launch `android sdk`:

```
>cd /android-sdk-macosx/tools
> ./android sdk
```

This opens the SDK Manager, shown in Figure 8-14. From the SDK Manager you can download and install an Android Platform and a set of platform tools. The platform you download will be the device image that you load up.

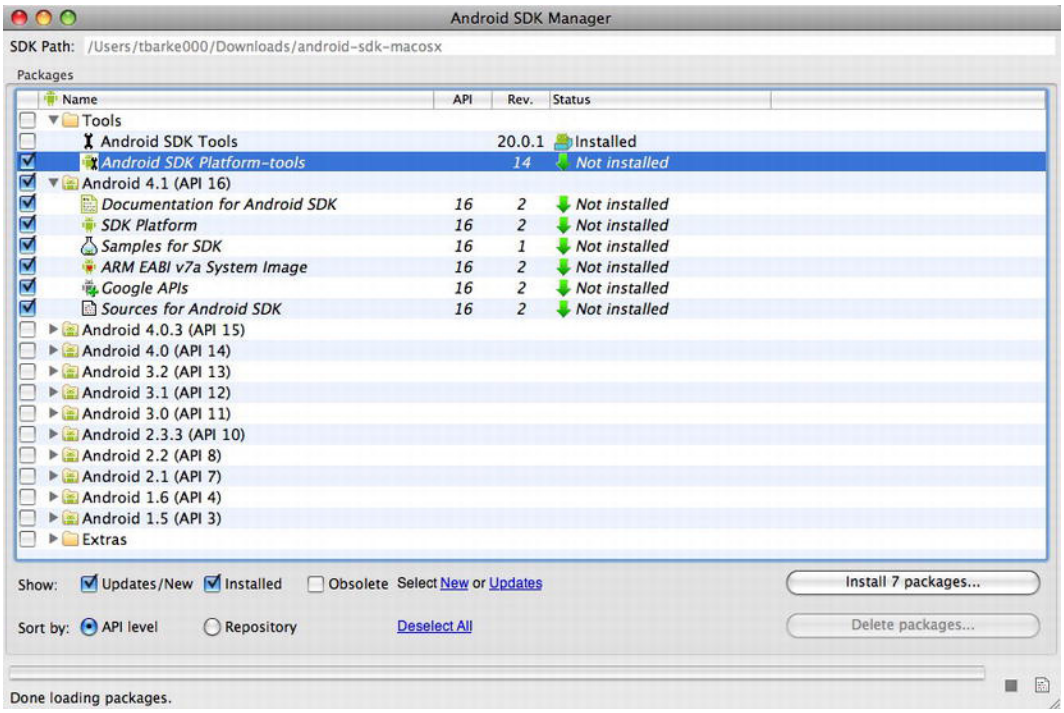

Figure 8-14. *Android SDK Manager*

Once you have downloaded a platform and the platform tools, you next need to launch the Android Virtual Device Manager, by running `android avd` in the `tools` directory:

```
./android avd
```

The Android Device Manager, much like Virtual Box, will allow you to create and run virtual machines. See Figure 8-15 for the Android Virtual Device Manager.

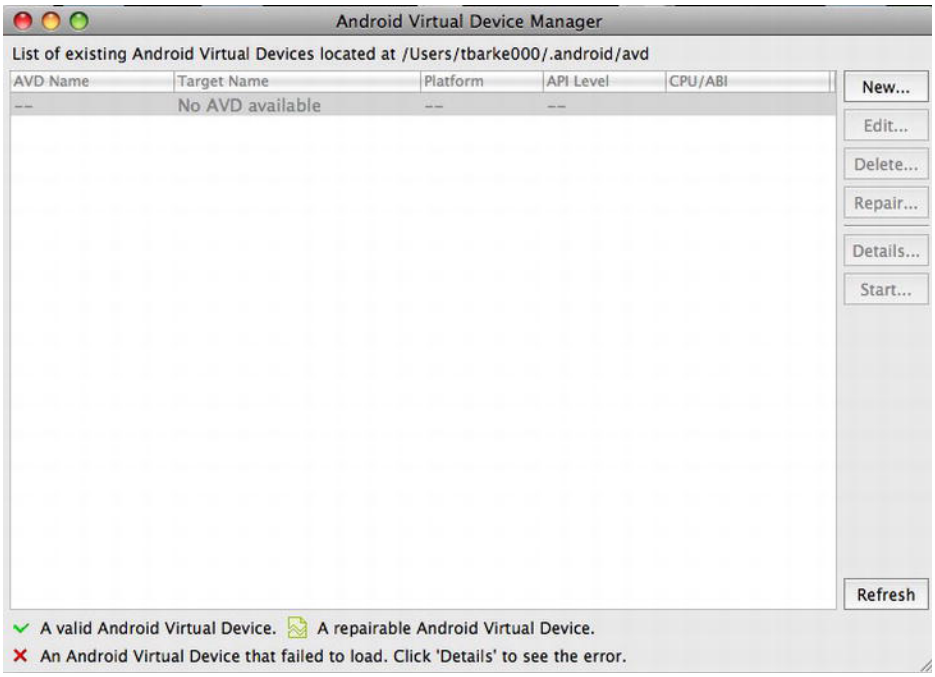

Figure 8-15. *The Android Virtual Device Manager, running on a Mac*

In the Android Virtual Device Manager you can create a new virtual device from the platform that you just downloaded. To do so, just click the New button to bring up the screen shown in Figure 8-16. Here you configure your new virtual device.

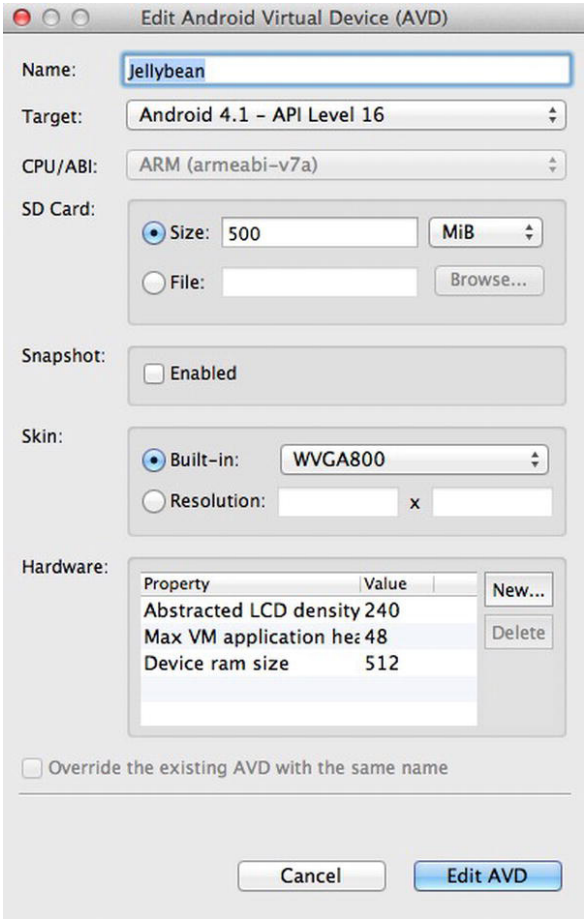

Figure 8-16. *Adding an Android Virtual Device*

When you are done you can launch the device and load up the browser. See Figure 8-17 for the emulator in action.

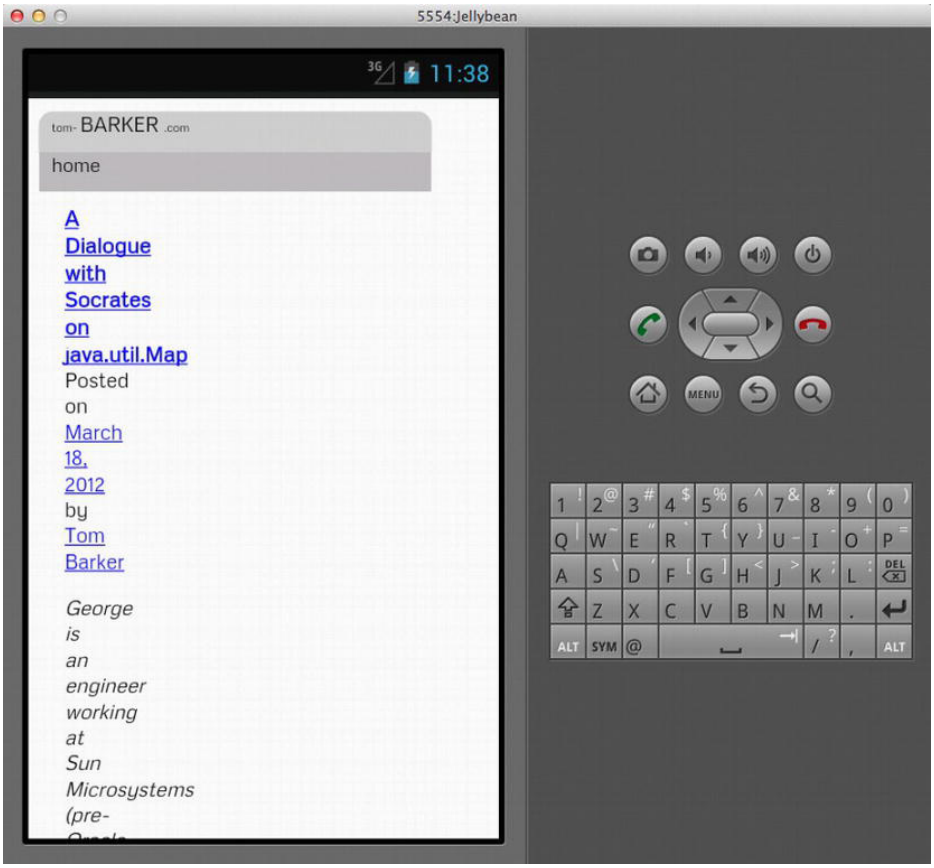

Figure 8-17. *tom-barker.com running on the Android emulator*

OK. You're instrumenting your production code, monitoring the web performance of your pages in production, and benchmarking the code in a test lab against your browser matrix. What now?

Share Your Findings

Benchmarking, instrumenting, and monitoring are great, but if you don't tell anyone, what is the point? Share your findings with your team. But first analyze your data—I mean really understand your data so that you can speak to each data point and have an idea about the cause or implication of each finding.

Use the data visualization skills that you have been refining throughout the course of the book to generate charts, assemble the charts into a report, and share your analysis. Play with different types of charts to see which ones better communicate your point.

Have an open mind, and consider the context. Are you missing something that can explain a larger picture? Get second opinions and double-check your tests. Maybe the test you are benchmarking is flawed in some way, like an improperly scoped variable throwing off your results.

Once you have your findings double-checked your analysis complete, and your charts created, you should assemble your results into a report, maybe an email, maybe a PDF, maybe a wiki entry; it just needs

to be something in which you can include not just your graphs but your analysis and context as well, and that can be distributed.

Review your report with your team, go over root causes, and come up with a plan of attack to address the areas for improvement. Above all else always strive to improve.

Summary

In this chapter we explored some closing thoughts about performance.

We talked about balancing performance with keeping the best practices of readability, reuse, and modularity. We looked at scorched-earth performance practices. We looked at the practice of inlining functions, coalescing them into a single function to reduce overhead that the JavaScript interpreter must go through to construct and execute function and object hierarchy. We created a test to compare the runtime performance of inlining functions versus using functions versus using objects.

While we saw performance gains with this scorched-earth performance practice, we also lost the gains of modularity, readability, and reusability that good software design gives us.

We also looked at running our code through Google's Closure Compiler. We saw significant web performance benefits. But we also saw that compiling our JavaScript down to the barest minimum also made our code much harder to debug, and would add a much more difficult layer of abstraction to maintain and update.

The point of these two examples was not just the raw numbers, it was that in all things we do we must strive to find balance. Performance is immensely important, but there are other aspects of quality just as important, if not more so.

We also talked about how to implement the things that we have learned. We discussed monitoring the web performance of our production sites using WPTRunner. We talked about using perfLogger to instrument our code live in the wild. We talked about assembling a browser support matrix and creating a test lab to benchmark our code in our test lab.

And finally we talked about the importance of sharing our data; using our findings as a feedback loop to identify areas of improvement in our continual quest to be excellent.