

Debugging Hard JavaScript Bugs

In this section we are going to look at debugging hard bugs. (If there is a particular kind of bug that is old hat and is debugged easily, wonderful – but then you do not need this appendix's assistance.)

In this chapter we will cover:

- Some aspects of hard debugging in any context
- Using browser debugging tools, which will be considered as a group

Firefox (with Firebug), Chrome, IE 8, Opera, and Safari have something a bit like Firebug. Firefox, Chrome, IE8, and Opera all offer debugging facilities and all let you use `console.log()`.

"Just fiddling with Firebug" is considered harmful

With a tool like Firebug comes a temptation to offload the responsibility from our shoulders to the debugger's. Features like Firebug's `console.dir()`, which lets you explore an object interactively, can be very powerful. However, they can become a crutch.

Cargo cult debugging at your fingertips

Who is most effective in hand-to-hand combat: someone unskilled who is wielding a sword, someone unskilled wielding a mace, or a good blackbelt who is empty-handed?

Any, or depending on the martial art, all of these people may be more effective with a weapon than without. But weapons do not wield themselves: people, skilled or unskilled, wield them. And as impressive a tool as Firebug is, or Chrome's developer tools, a skilled developer using `alert()` well can and will debug better than a programmer who hopes that Firebug will provide a way out of the difficult work of understanding hard bugs.

Some people have raised concerns that expensive medical diagnostics make for worse physician diagnoses, not better. The reason for this is that a doctor relying on clinical observation is aware of the need for attentive vigilance in trying to make the best observations from what is available. With testing, there is a new temptation to try and let the test make the diagnosis, and a test that is asked to make a diagnosis is just not as good as a good physician's observation and reasoning. Nobody sets out for this to happen, but it does happen.

In the imagination of popular children's stories, detectives carry around magnifying glasses to look at things. Detectives today use more technology, not less, and can and do make magnified photographs. Perhaps the picturesque magnifying glass is a quaint image, and real police detectives may not use them as well-equipped soldiers do not have quaint flintlock muskets any more. But it is the martial artist, detective, doctor, or *developer*, who uses the tool. And that means that before we look at how good debugging can use Firebug or Google Chrome's developer tools, we should look at good debugging itself. Only after we have laid a foundation is it appropriate to ornament it with the virtues of good debugging weapons.

The scientific method of debugging

The core of the scientific method is not just something useful for explicitly labeled science. It can be used in several cases. In the scientific method, we do something like the following:

- Start with something you have observed but you don't understand
- Then you make hypotheses: educated guesses that might account for the phenomenon
- Think of an experiment, or experiments, where your hypotheses predict different results
- Perform the experiment(s) and try to narrow down which hypotheses account for all the data
- Repeat the cycle as desired

In science, this method is applied to understand how the natural world works. In scientific debugging, we use it to understand how our software does not work: we cannot solve the problem correctly until we understand what the problem is. And in debugging, band-aid solutions are just that: band-aids that leave the root cause unaddressed. The scientific method of debugging is not a tool for finding a band-aid that will mask a problem, but for finding the root cause so we can fix it there. We try first to duplicate a problem, then reproduce it consistently, then pin down when it does and does not happen. This is true whether or not we have debugging tools to use.

Exhausting yourself by barking up the wrong tree

Often when we are having trouble finding a bug, we are looking somewhere the bug cannot be found. You cannot find something by looking twice as hard in the wrong place.

Sometimes, *if we listen*, we will get subtle, easily overlooked clues that we are barking up the wrong tree. For instance, suppose we have the following code:

```
function init()
{
  if (detect_feature())
  {
    initialize_with_feature();
  }
  else
  {
    initialize_without_feature();
  }
}
```

And we can't tell which branch of the conditional is being executed. So we add `console.log()` calls (which work in either Firebug or Chrome):

```
function init()
{
  if (detect_feature())
  {
    console.log("init 1");
    initialize_with_feature();
  }
  else
```

```
{
  console.log("init 2");
  initialize_without_feature();
}
```



We might suggest a naming convention for `console.log()` calls of the function's name, and then a space, and then a number or unique identifier for the call, and then any variable or other information which we might want to inspect. This is both easy and makes large amounts of logging material, *if we need it*, more navigable.

So we run this, and to our consternation nothing shows up in the log: neither `init 1` nor `init 2` is logged. The temptation is to say that the experiment was a complete failure. But if we are open to a surprise and a bit of serendipity, the experiment was a success: it showed that neither branch was being executed. That could be because `init()` was not being called at all, or because `detect_feature()` was hanging and not returning, or throwing an exception that is silenced later on. So to test that, we could add another `log` line:

```
function init()
{
  console.log("init 3");
  if (detect_feature())
  {
    console.log("init 1");
    initialize_with_feature();
  }
  else
  {
    console.log("init 2");
    initialize_without_feature();
  }
}
```

And then we have evidence either that `init()` is not being called, or that it is being called but `detect_feature()` is not completing, quickly and successfully, and passing control to either branch of the conditional.

The humble debugger

Dijkstra's famous essay *The humble programmer* talked about how arrogance about one's programming abilities makes for a bad programmer. Humility can help us debug, and not only by letting us accept that our code has defects. Another side of programming humility, on a deeper level, lets us be open to surprises. Humility is an openness that can receive serendipity, and it is an openness that can receive what the code actually has instead of projecting its desires, reading into the code what it wants to see. More concretely, part of humble openness means that when you do something and the program doesn't do what you want, you are open, and choose to be open, to what the program is actually doing. What you avoid is doing the same thing a few more times in the hope that it will work correctly when you know you have seen a bug.

In concrete, this means unplugging a time sink that slowly nickels and dimes away time spent sticking one's head in the sand when we could cut to the chase and start real debugging the *first* time the bug shows itself, not reluctantly after it refuses to go away and yield to our wishful thinking. *If you are looking and looking for a bug and not finding it, the reason may well be that the bug you are looking for doesn't exist anywhere. What you need to do is recognize when you are searching and by accident stumble over a different problem that will let you address the issue.* Now part of debugging may involve legitimately doing things again to try to sound out the scope and what triggers a bug. But that does not make it helpful to consent to having our time nickeled and dimed away because we allow ourselves wishful thinking and the tacit hope that maybe the bug will go away without being fixed if only we try again a time or two.

Solving hard bugs is usually a matter of serendipity. It is like trying to find a needle in a haystack, and accidentally discovering that there are valuables that had been hidden in the haystack. It is tripping over the truth but not, *pace* Winston Churchill, picking yourself up and continuing on.

The value of taking a break

Taking a break can work. Asking an extra set of eyes can work. And pulling an all-nighter is working harder, not smarter: one hour of looking at code with fresh and well-rested eyes is worth a dozen hours of staring at the same code with blurry, sleepy eyes.

If humility matters more than you might think, persistence may be overrated. If you are locked into finding a bug one way, you may do well with a break. *A Whack on the Side of the Head* by Roger von Oech, *Business Plus*, available as an iPhone app, may be invaluable (see <http://creativethink.com/>). *Conceptual Blockbusting* by James Adams, *Perseus* is also invaluable. A bit of brainstorming may help you loosen up from being locked into searching where the problem may not be at all. And a bit of exercise can get blood flowing as well as loosening your thoughts.

Two major benefits to asking for help

There are two major benefits to asking for help. The first benefit is that the person you ask might be able to solve the problem. But there is another, less obvious benefit: when you ask another programmer for help, usually you do not just say "My program is not working"; you carefully explain what is going on so the other person has something to go on. And it happens more than you might expect that in the course of explaining what is going on, you realize what is causing the other bug, whether or not your dialogue partner spoke a word. "Confessional debugging" happens all the time.

Firebug and Chrome developer tools

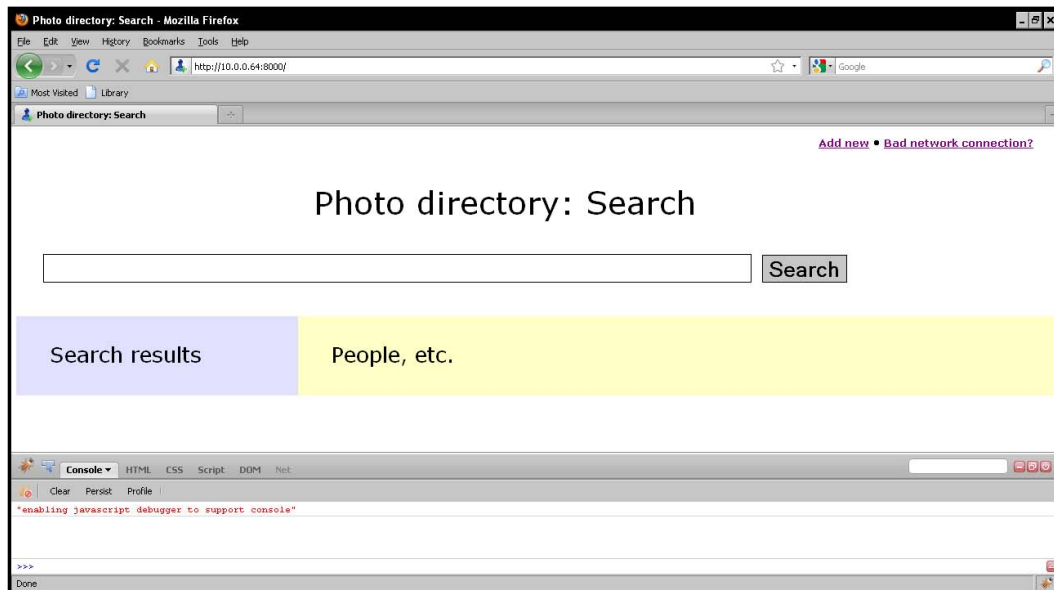
Firebug set the standard for good debugging support, and Chrome web development tools are not a port of Firebug to Chrome, but are for Chrome what Firebug is for Firefox. Both are powerful; for CSS work, sometimes there may be things visible in Firebug that do not show up in Chrome.

The basics across browsers

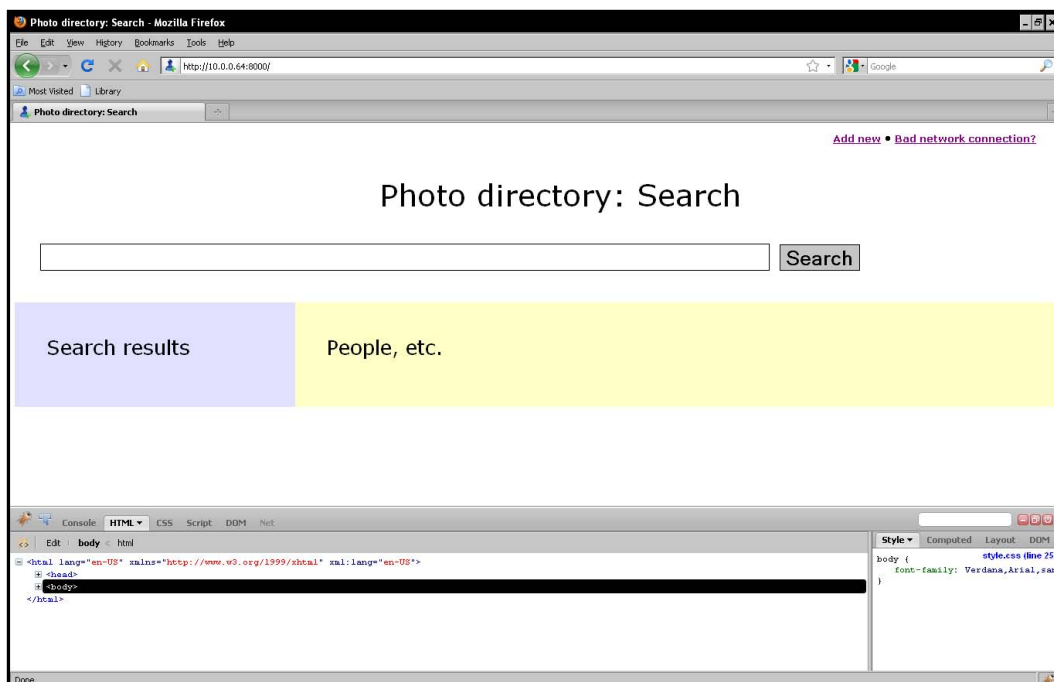
Firebug's console displays the output of `console.log()`, and it also displays requests (GET, POST, and the like), which you can drill down and see their contents.

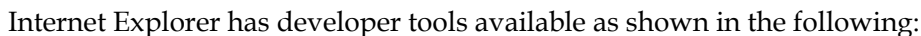
One helpful thing when something isn't working is to look at what the server is returning in response to requests. The basic question is whether the server is sending good data that the client is not handling properly, or whether the server is sending bad data: in terms of scientific debugging, this is a way to narrow down where the bug is hiding.

The console, with the output of `console.log()` statements, looks like this:

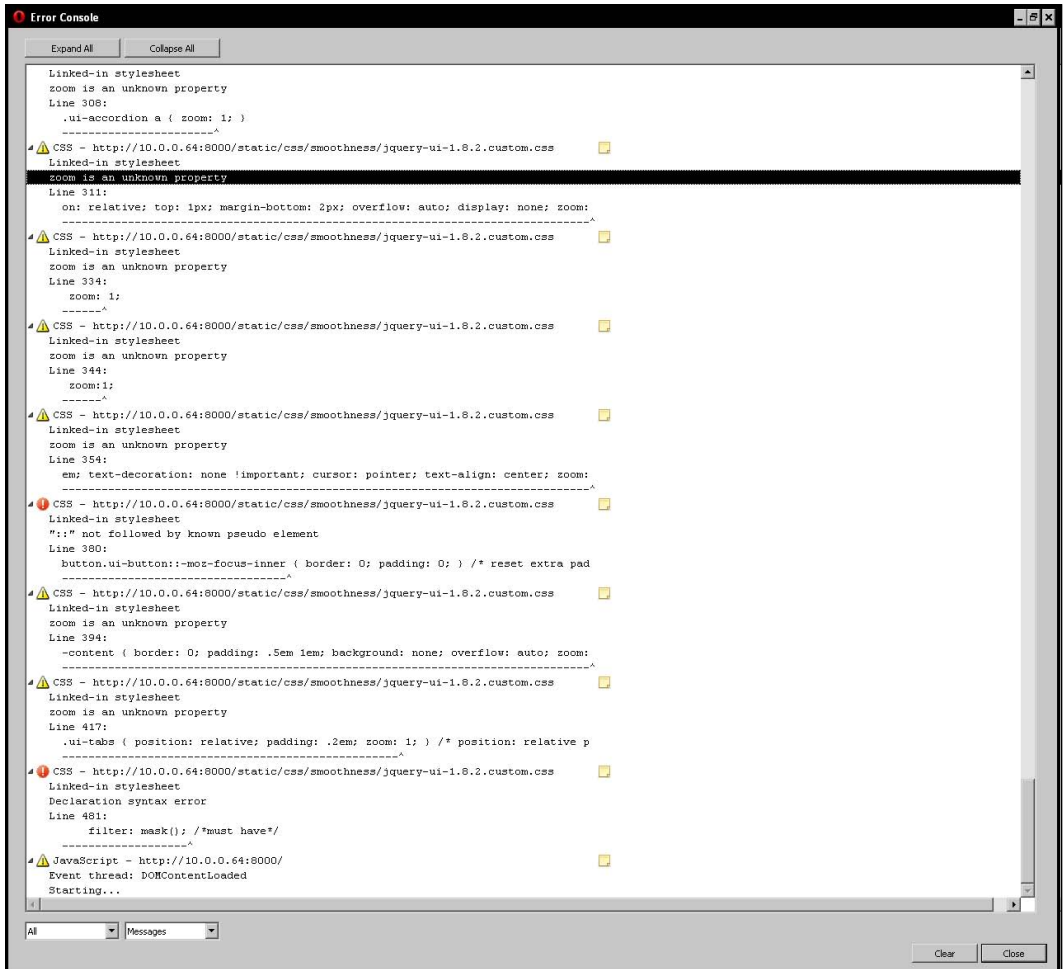


It is accessed by installing Firebug from <http://getfirebug.com/>, and then clicking the bug icon at the bottom-right of your window:





And in Opera, we have the following:

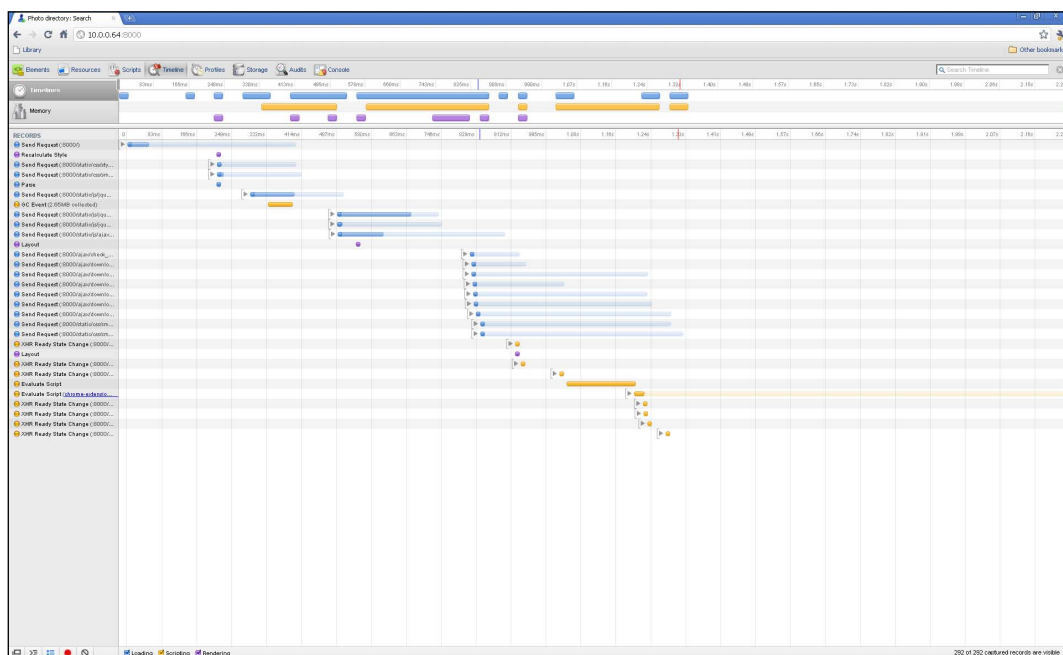


Zeroing in on Chrome

Chrome has the following tabs/buttons:

- **Elements**, which allows drilldown access to the live DOM.
- **Resources**, which is a dashboard for seeing what the delays were in a page being rendered.
- **Scripts**, which provides debugger services in the tradition of GDB for debugging JavaScript, including setting breakpoints, seeing the values of variables, and so on.

- **Timelines**, which can be used to track resource usage for optimization. The bottom-left hand of the screen has record/stop and clear buttons:



- **Profiles**, which allows CPU and memory usage profiling.
- **Storage**, which looks at cookies, sessions, HTML5 databases, and other local storage.
- **Audits**, which are meant to support third-party plugins to audit pretty much anything: mobile readiness, network optimization, and so on.
- **Console**, which provides a command-line console that includes but goes beyond Firebug's command line.

What you can see, in terms of the DOM, JavaScript, CSS, and so on, you can modify to allow surgical experiments rather than reloading a page and doing multiple things to keep track of state. In keeping with "Where there is output, let there be input," clicking, double-clicking, or right-clicking on the live DOM model, JavaScript code, properties in CSS as well as the DOM or JavaScript, and the like, can be used to modify existing values.

Firebug got the game going and showed what it was to utilize a JavaScript debugger correctly. Internet Explorer has something vaguely in the same vein, but perhaps not a full-blooded replacement. Google in making Chrome tried to see how far they could push the envelope; as already mentioned, the developer tools are for Chrome

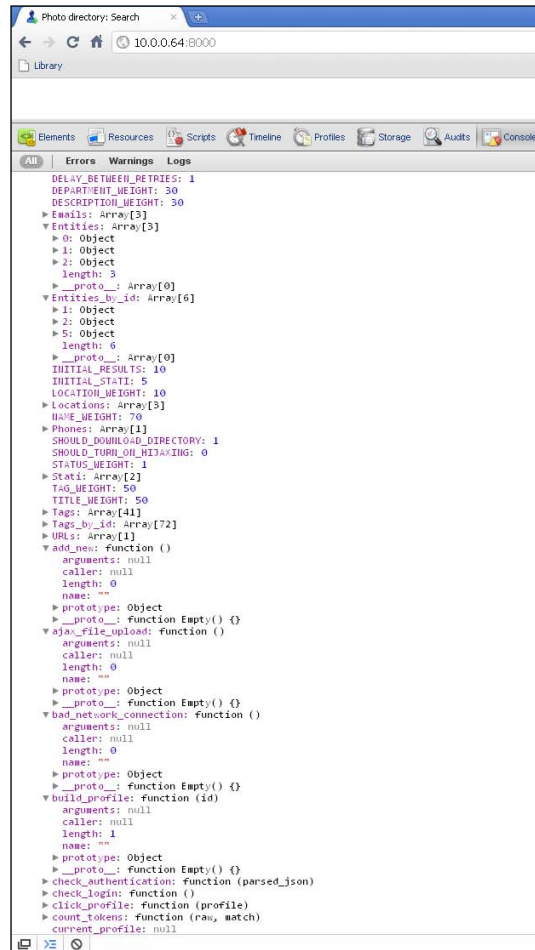
what Firebug is for Firefox, but they are not a Firebug port. They are extremely deep and worth a book in their own right; Google's resources on the web can be found at <http://www.chromium.org/devtools>, with an in-depth video at <http://tinyurl.com/GoogleChromeDevToolsInDepth>, to which we refer the reader for further study. A `dir()`, for example, of our `PHOTO_DIRECTORY` namespace yields both a broad overview and deep possibilities for drilling down:



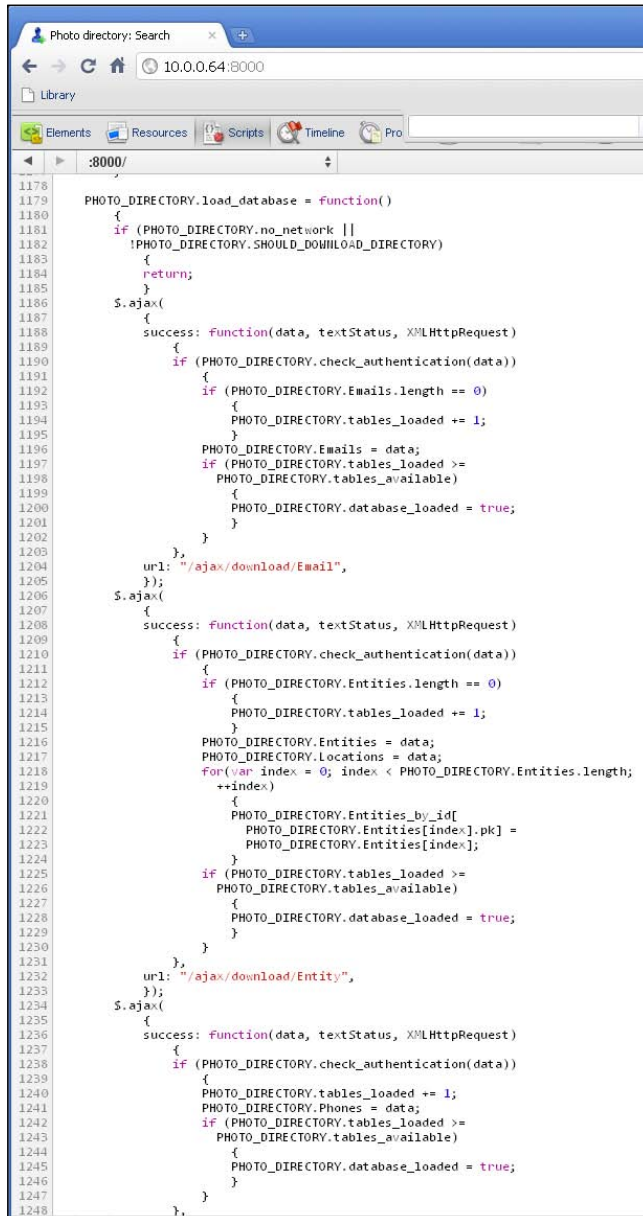
Let's stop a minute and pause for a moment of, "That's funny..."

We are not now actively trying to solve a bug, but we may have stumbled on something odd: this screenshot shows three **Entities** and six **Entities_by_id**. Our off-the-cuff expectation may be that that the two counts should be the same. Even though we aren't actively trying to track down a bug, nor address symptoms, this may be the root cause of a bug.

Before we drill down it would help to think. There are at least two obvious expectations, one or both of which may be wrong. It could be that **Entities_by_id** are getting things twice; we observe that six **Entities_by_id** is exactly a multiple of three **Entities**. Or it could just be a few more. If there are **Entities** that have been deleted, it could be that **Entities_by_id** reflects deleted entries and **Entities** does not. With that in mind, let's drill down and see what we see, how well the system treats our hypotheses:



This suggests another explanation, that despite Chrome leading us to believe we have a six-element array, we might have an array of length 6 from JavaScript's perspective, but the keys used to populate it max out at five. It has exactly three physical entries, just like the other. Let's go over to the **Scripts** page and select the **:8000/** entry (the one for JavaScript on the page, although we can see other source files too). That reads as follows:



```

1178 PHOTO_DIRECTORY.load_database = function()
1179 {
1180     if (PHOTO_DIRECTORY.no_network ||
1181         !PHOTO_DIRECTORY.SHOULD_DOWNLOAD_DIRECTORY)
1182     {
1183         return;
1184     }
1185     $.ajax(
1186     {
1187         success: function(data, textStatus, XMLHttpRequest)
1188         {
1189             if (PHOTO_DIRECTORY.check_authentication(data))
1190             {
1191                 if (PHOTO_DIRECTORY.Emails.length == 0)
1192                 {
1193                     PHOTO_DIRECTORY.tables_loaded += 1;
1194                 }
1195                 PHOTO_DIRECTORY.Emails = data;
1196                 if (PHOTO_DIRECTORY.tables_loaded >=
1197                     PHOTO_DIRECTORY.tables_available)
1198                 {
1199                     PHOTO_DIRECTORY.database_loaded = true;
1200                 }
1201             }
1202         },
1203         url: "/ajax/download/Email",
1204     });
1205     $.ajax(
1206     {
1207         success: function(data, textStatus, XMLHttpRequest)
1208         {
1209             if (PHOTO_DIRECTORY.check_authentication(data))
1210             {
1211                 if (PHOTO_DIRECTORY.Entities.length == 0)
1212                 {
1213                     PHOTO_DIRECTORY.tables_loaded += 1;
1214                 }
1215                 PHOTO_DIRECTORY.Entities = data;
1216                 PHOTO_DIRECTORY.Locations = data;
1217                 for(var index = 0; index < PHOTO_DIRECTORY.Entities.length;
1218                     ++index)
1219                 {
1220                     PHOTO_DIRECTORY.Entities_by_id[
1221                         PHOTO_DIRECTORY.Entities[index].pk] =
1222                         PHOTO_DIRECTORY.Entities[index];
1223                 }
1224                 if (PHOTO_DIRECTORY.tables_loaded >=
1225                     PHOTO_DIRECTORY.tables_available)
1226                 {
1227                     PHOTO_DIRECTORY.database_loaded = true;
1228                 }
1229             }
1230         },
1231         url: "/ajax/download/Entity",
1232     });
1233     $.ajax(
1234     {
1235         success: function(data, textStatus, XMLHttpRequest)
1236         {
1237             if (PHOTO_DIRECTORY.check_authentication(data))
1238             {
1239                 PHOTO_DIRECTORY.tables_loaded += 1;
1240                 PHOTO_DIRECTORY.Phones = data;
1241                 if (PHOTO_DIRECTORY.tables_loaded >=
1242                     PHOTO_DIRECTORY.tables_available)
1243                 {
1244                     PHOTO_DIRECTORY.database_loaded = true;
1245                 }
1246             }
1247         },
1248     },

```

`load_database()` loads into `Entities_by_id` exactly what the name would suggest: it is being used as a hash with what turn out to be primary keys assigned by Django, and this installation has some entries that are deleted, leaving holes. It so happens that, for our implementation, the JavaScript interpreter treats this as a (slightly) sparse integer array.

This particular bug hunt was a lottery ticket that, like most lottery tickets, didn't pan out. However, it is an example of how to debug live, and live debugging isn't just when you use the tool optimally and pin down an annoying bug with seven symptoms. That may be the kind of story we prefer to tell our fellow programmers, but it isn't the only story in a debugging phase of programming. It is a live example of the exploratory process that sometimes finds the bugs being traced, sometimes stumbles on unrelated bugs, and sometimes doesn't turn up anything interesting. (Or it may turn up something else interesting, like a juicy hook for an improved feature offering.)

The point of debugging remains scientific debugging, even with tools at our fingertips that will do all this and more.

Summary

In this appendix, we have looked at the basic mindset for debugging and use of tools, tools available for different browsers, and Chrome's developer tools.

In this book, we have had a whirlwind tour of Django Ajax with jQuery, and paid attention to usability. We have covered the basics of Ajax with Django, jQuery as the most common JavaScript framework, server-side validation, especially in light of usability, server-side database search in Django, jQuery UI in-place editing, how to implement autocomplete functionality, Django's `ModelForm` as a powerful and easy tool, client-side processing, including a client-side database, customization and further development, tinkering to make a working system better, usability for hackers, and debugging JavaScript.

Where next?

Probably best a mixture of web searches, library trips, practical development, and a brainstorm or two about places to go!