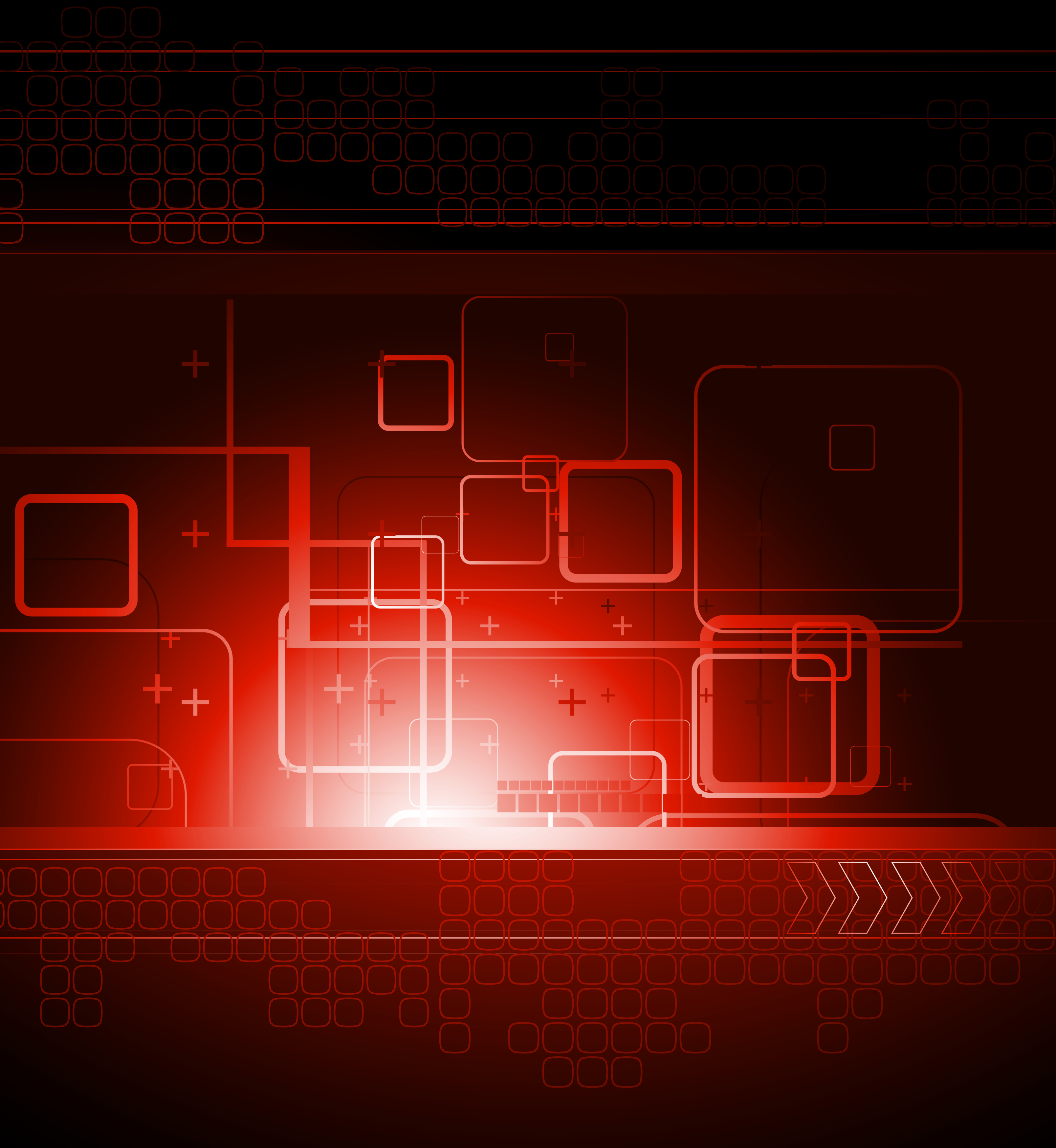
;login: *logout*

EXCLUSIVE ELECTRONIC EDITION

MARCH 2014

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION



;login: logout

EXCLUSIVE ELECTRONIC EDITION MARCH 2014

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

COPY EDITOR
Steve Gilmartin
proofshop@usenix.org

PRODUCTION MANAGER
Michele Nelson

PRODUCTION
Arnold Gatilao
Casey Henderson

USENIX ASSOCIATION
2560 Ninth Street, Suite 215,
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738
www.usenix.org

To Wash It All Away

JAMES MICKENS

This is my last column! Thanks for reading, and thanks for all of the support :-). Never forget that when you are alone, I am with you, and when you are with someone else, I am also with you, because I think that I am better than that other person and I have lengthy opinions about why this is true. mickens@microsoft.com

When I was in graduate school in Ann Arbor, I had a friend who was deeply involved with the environmentalist movement. He purchased his food from local farmers' markets, and he commuted by bike instead of by car to reduce his carbon footprint, and he maintained a horrid compost bin that will probably be the origin of the next flu pandemic. One day, he told me that he was going to visit a farm for a week. I asked him why, and he said that he wanted to "get closer to the land," a phrase that you can only say with a straight face if you're narrating a documentary about ancient South American tribes. I told my friend that the land didn't want to get closer to him, and if he really looked at the land, he'd see that it was not composed of delicious organic trail mix, but famine and vultures and backbreaking labor involving wheelbarrows and generally unacceptable quantities of insects. He responded with an extended lecture about eco-responsibility, a lecture that I immediately forgot because I realized that my naïve friend was going to die on that farm. So, I told my friend that he shouldn't be afraid to end his trip early if he wasn't having a good time. He smiled at me, the way that people in slasher movies smile before they get chopped up, and he left for the farm. Precisely 37 hours later, he called me on the phone. I asked him how everything was going, and he made a haunting, elegiac noise, like a foghorn calling out for its mate. I asked him to describe his first day, and he said that his entire existence revolved around bleating things: bleating goats that wanted to be fed, and bleating crows that wanted to steal the food that he gave the bleating goats, and bleating farm machines that were composed of spinning metal blades and had no discernable purpose besides enrolling you in the "Hook Hand of the Month" club. I asked my friend when he was coming back home, and he said that he was calling me from the Ann Arbor train station; he had already returned. And then he let out that foghorn noise, that awful, lingering sound, and I thought, MAYBE THAT'S THE FIRST SYMPTOM OF COMPOST BIN FLU.

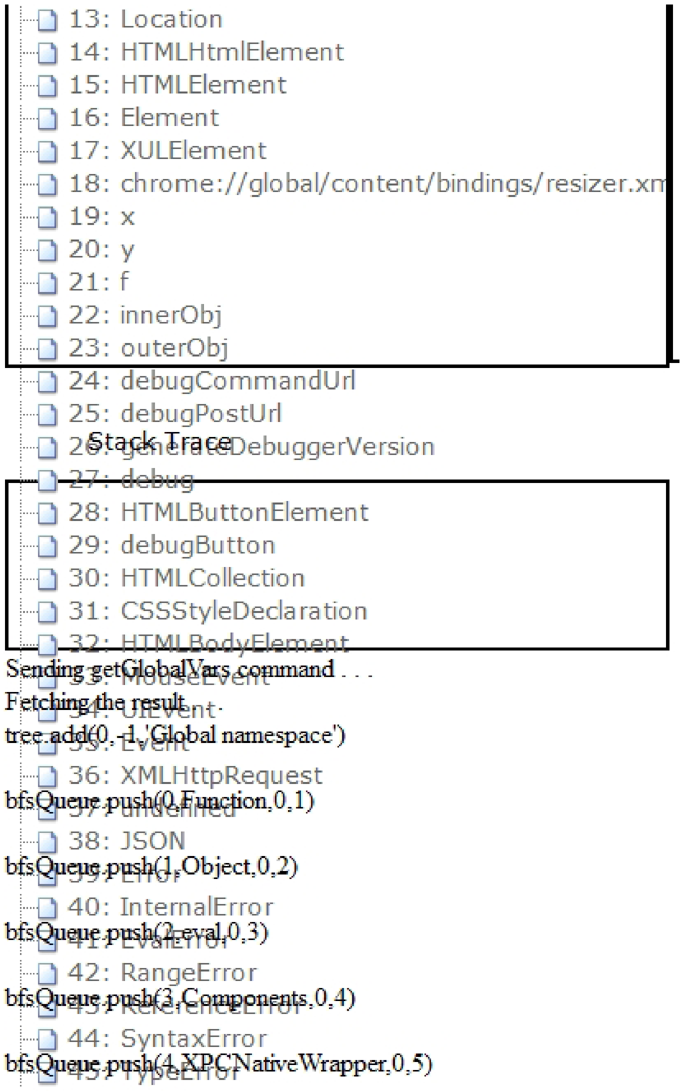
Computer scientists often look at Web pages in the same way that my friend looked at farms. People think that Web browsers are elegant computation platforms, and Web pages are light, fluffy things that you can edit in Notepad as you trade ironic comments with your friends in the coffee shop. *Nothing could be further from the truth*. A modern Web page is a catastrophe. It's like a scene from one of those apocalyptic medieval paintings that depicts what would happen if Galactus arrived: people are tumbling into fiery crevasses and lamenting various lamentable things and hanging from playground equipment that would not pass OSHA safety checks. This kind of stuff is exactly what you'll see if you look at the HTML, CSS, and JavaScript in a modern Web page. Of course, no human can truly "look" at this content, because a Web page is now like V'Ger from the first "Star Trek" movie, a piece of technology that we once understood but can no longer fathom, a thrashing leviathan of code and markup written by people so untrustworthy that they're not even third parties, they're fifth parties who weren't even INVITED to the party, but who showed up anyways because the hippies got it right and free love or whatever. I'm pretty sure that the Web browser is one of the "dens of iniquity" that I keep hearing about on Fox News; I would verify this using a Web search, but a Web search would require me to use a browser, AND THIS IS EXACTLY WHAT BICOASTAL LIBERAL ELITES WANT ME TO DO.

Describing why the Web is horrible is like describing why it's horrible to drown in an ocean composed of pufferfish that are pregnant with tiny Freddy Kruegers—each detail is horrendous in isolation, but the aggregate sum is delightfully arranged into a hate flower that blooms all year. For example, the World Wide Web Consortium (W3C) provides "official" specifications for many client-side Web technologies. Unfortunately, these specifications are binding upon browser vendors in the same way that you can ask a Gila monster to meet you at the airport, but that gila monster may, in fact, have better things to do [1]. Each W3C document is filled with alienating sentences that largely consist of hyperlinks to different hyperlinks. For instance, if you're a browser vendor, and you want to add support for HTML selectors, you should remember that, during the third step of parsing the selector string, "If ***result*** is invalid ([SELECT], section 12), raise a SYNTAX_ERR exception ([DOM-LEVEL-3-CORE], section 1.4) and abort this algorithm." Such bodice-ripping legalese is definitely exciting for people who yearn for the dullness of the Cheerios ingredient list combined with the multi-layered bureaucracy of the Soviet Union. Indeed, you could imagine a world in which browser vendors hire legions of Talmudic scholars to understand why, precisely, SYNTAX_ERR is orange and not mauve, and how, exactly, this orangeness relates to the parenthetic purpleness of ([DOM-LEVEL-3-CORE]). You could also imagine a world in which browser vendors do not do this, and instead implement 53% of each spec and then hope that no Web page tries to use HTML selectors and then the geolocation interface and then a <canvas> tag, because that sequence of events will unleash the Antichrist and/or a rendered Web page that looks like one of those Picasso paintings that you pretend to understand, but which everyone wants to throw into an ocean because nobody wants to look at a painting of a blue man who is composed of isosceles triangles and has a guitar emerging from his forehead for no reason at all.

Given the unbearable proliferation of Web standards, and the comically ill-expressed semantics of those standards, browser vendors should just give up and tell society to stop asking for such ridiculous things. However, this opinion is unpopular, because nobody will watch your TED talk if your sense of optimism is grounded in reality. I frequently try to explain to my friends why they should abandon Web pages and exchange information using sunlight reflected from mirrors, or the enthusiastic waving of colored flags. My friends inevitably respond with a spiritually vacant affirmation like, "People invented flying machines, so we can certainly make a good browser!" Unfortunately, defining success for a flying machine is easy ("I'M ME BUT I'M A BIRD"), whereas defining success for a Web browser involves Cascading Style Sheets, a technology which intrinsically dooms any project to epic failure. For the uninitiated, Cascading Style Sheets are a cryptic language developed by the Freemasons to obscure the visual nature of reality and encourage people to depict things using ASCII art. Ostensibly, CSS files allow you to separate the definition of your content from the definition of how that content looks—using CSS, you can specify the layout for your HTML tags, as well as the fonts and the color schemes used by those tags. Sadly, the relationship between CSS and HTML is the same relationship that links the instructions for building your IKEA bed, and the unassembled, spiteful wooden planks that purportedly contain latent bed structures. CSS is not so much a description of what your final page will look like, but rather a loose, high-level overview of what *could* happen to your page, depending on the weather, the stock market, and how long it's been since you last spoke to your mother. Like a naïve Dungeon Master untouched by the sorrow of adulthood, you create imaginative CSS classes for your <div> tags and your <span> tags, assigning them strengths and weaknesses, and defining the roles that they will play in the larger, uplifting narrative of your HTML. Everything is assembled in its proper place; you load your page in a browser and prepare yourself for a glorious victory. However, you quickly discover that your elf tag is overweight. THE ELF CAN NEVER BE OVERWEIGHT. Even worse, your barbarian tag does not have an oversized hammer or axe. Without an oversized hammer or axe, YOUR BARBARIAN IS JUST AN ILLITERATE STEROID USER. And then you look at your wizard tag, and you see that he's not an old white man with a flowing beard, but a young black man from Brooklyn. FOR COMPLEX REASONS THAT ARE ROOTED IN EUROPEAN COLONIAL NARRATIVES, YOUR WIZARD MUST BE AN OLD WHITE MAN WITH A FLOWING BEARD, NOT A BLACK MAN WITH HIPSTER SHOES AND A FANTASTIC VINYL COLLECTION. Such are the disasters

[1] "Gila Monsters Meet You at the Airport" is the name of a real children's book that had an enormous impact on my emotional growth. The book's unflinching realism inspired my own series of ill-received children's books, such as "Spiders Ate Your Sister That We Never Talk About," "Capitalist Advertising Makes You Hate Your Body and Buy Things Made from Slave Labor," and "I Could Lie and Say that Your Comic Book Collection Is Interesting, Or I Could Tell The Truth and Explain Why You Don't Get Second Dates."

To Wash It All Away



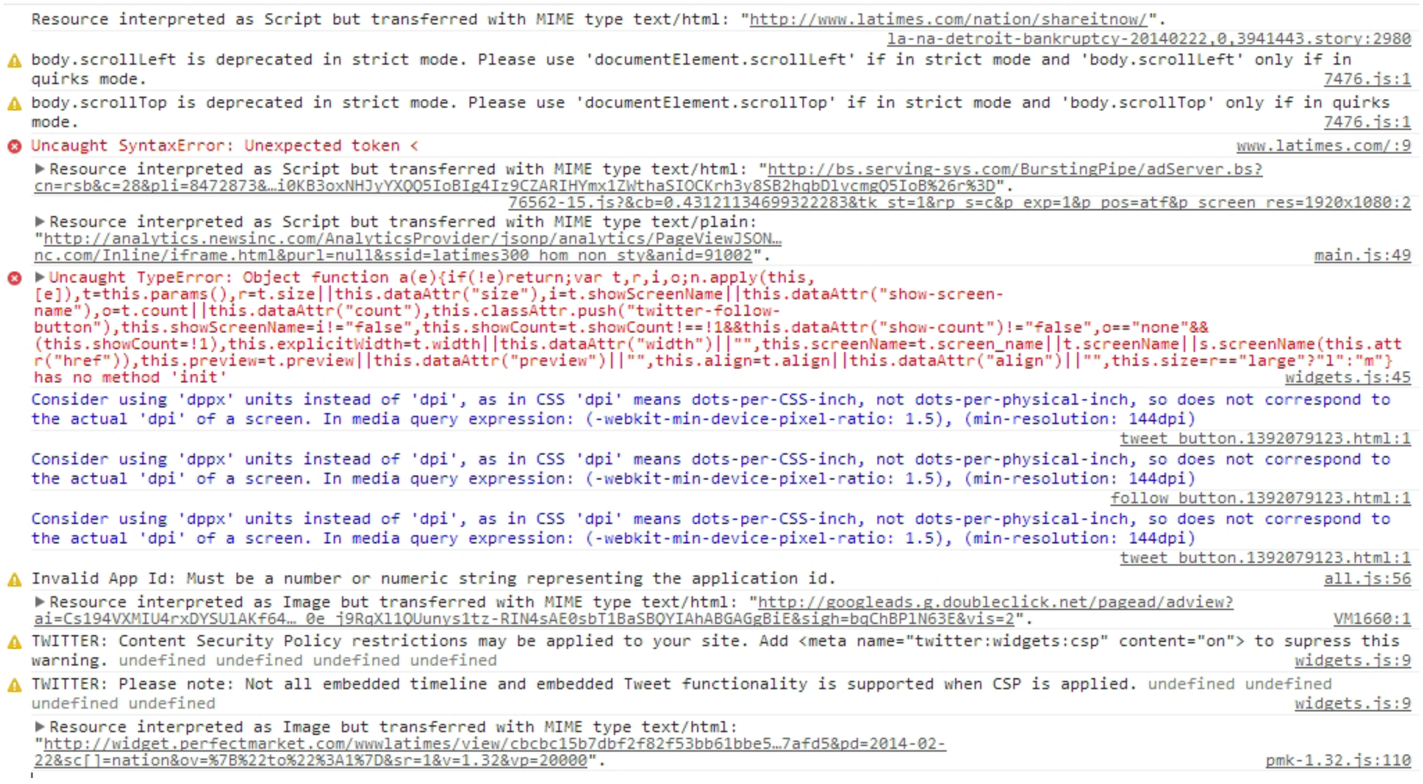
```
13: Location
14: HTMLHtmlElement
15: HTMLElement
16: Element
17: XULElement
18: chrome://global/content/bindings/resizer.xm
19: x
20: y
21: f
22: innerObj
23: outerObj
24: debugCommandUrl
25: debugPostUrl
Stack Trace
26: generateDebuggerVersion
27: debug
28: HTMLButtonElement
29: debugButton
30: HTMLCollection
31: CSSStyleDeclaration
32: HTMLBodyElement
Sending getGlobalVars command . . .
33: MouseEvent
Fetching the result . . .
34: UIEvent
tree.add(0,-1,'Global namespace')
35: Event
36: XMLHttpRequest
bfsQueue.push(0,Function,0,1)
37: undefined
38: JSON
bfsQueue.push(1,Object,0,2)
39: Error
40: InternalError
bfsQueue.push(2,eval,0,3)
41: EvalError
42: RangeError
bfsQueue.push(3,Components,0,4)
43: ReferenceError
44: SyntaxError
bfsQueue.push(4,XPCNativeWrapper,0,5)
45: TypeError
```

Figure 1: One time, I tried to build a browser-agnostic debugging infrastructure. I had a client-side JavaScript library that could traverse the JavaScript heap and display fun things about the page's state. My art history friends told me that console output is for Neanderthals, so I made an HTML GUI to display the diagnostic information. The first version of the GUI used the browser's default layout policies. Much like Icarus, I dreamt of more, so I decided to make A Fancy Layout™. I wrote CSS that specified whether my tags should have static positioning, or floating positioning, or relative zodiac-based positioning. Here's what I learned: *Never specify whether your tags should be static or floating or zodiac-based.* As soon as a single tag is released from the automatic layout process, the browser will immediately go insane and stack random HTML tags along the z-axis, an axis which apparently is an option even if your monitor can only display two dimensions. I eventually found a working CSS file inside a bottle that washed up on the beach, and I tweaked the file until it worked for my GUI. Then I went home and cried big man tears that were filled with ninja stars and that turned into lions when they hit the ground.

that CSS will wroughth upon thee. Or wrought *at* thee. To be honest, I don't know how to conjugate or spell "wrought," but my point is undoubtedly understood. See Figure 1 for a concrete example of CSS's wroughtiness. Or CSS' (no trailing "s") wroughtiness. MY NON-CASCADING STYLE MANUALS FIGHT FOR MY SOUL.

When you're a Web developer, CSS is just one of your worries. The aggregate stack of Web technologies is so fragile that developers just accept a world in which various parts of a Web page will fail at random times. Apparently this is okay because e-commerce isn't a serious thing, and if you really wanted a secure banking experience, you'd visit the bank in person like someone from the 1800s instead of accessing a banking Web site that is constantly (*but silently*) vomiting execution errors to the console log (a console log which the browser does not show by default, because if you knew about it, and you read its tales of woe, you'd abandon computer science and become a maker of fine wooden shoes). In Figure 2, I provide an unaltered example of such a console log; the log was generated by a real Web page from a popular site.

- The first log entry says that the browser executed a downloaded file as JavaScript, even though the MIME type of the file was text/html. Here's a life tip: when you're confused about what something is, DON'T EXECUTE IT TO DISCOVER MORE CLUES. This is like observing that your next-door neighbor is a creepy, bedraggled man with weird eyes, and then you start falling asleep on his doorstep using a chloroform rag as a pillow, just to make sure that he's not going to tie you to a radiator and force you to paint tiny figurines. Here's how your life story ends: YOU ARE A PAINTER OF TINY FIGURINES.
- The second and third errors say that the page's JavaScript used a variable name that is deprecated in strict mode, but acceptable in quirks mode. How can I begin to explain this delicious confection of awfulness? Listen: when a man and a woman fall in love, they want to demonstrate their love to each other. So, they force browsers to support different types of runtime environments. "Standards mode" refers to the unreliable browser APIs that are described by recent HTML and CSS specifications. "Quirks mode" refers to the unreliable browser APIs that were defined by browsers from the Eisenhower administration. Quirks mode was originally invented because many Web pages were made during the Eisenhower administration, and the computing industry wanted to preserve Web-based narratives about why "the rock and roll" is corrupting our youth. Quirks mode then persisted because Web developers learned about quirks mode and used it as an excuse to not learn new skills. But then some Web developers *wanted* to learn new skills, so standards mode was invented to allow these developers to make old mistakes in new ways. There is also a third browser mode called "almost standards mode"; this mode is similar to



```
Resource interpreted as Script but transferred with MIME type text/html: "http://www.latimes.com/nation/shareitnow/".
                                                                    la-na-detroit-bankruptcy-20140222,0,3941443.story:2980
body.scrollLeft is deprecated in strict mode. Please use 'documentElement.scrollLeft' if in strict mode and 'body.scrollLeft' only if in
quirks mode.                                                                                                      7476.js:1
body.scrollTop is deprecated in strict mode. Please use 'documentElement.scrollTop' if in strict mode and 'body.scrollTop' only if in quirks
mode.                                                                                                             7476.js:1
Uncaught SyntaxError: Unexpected token <                                                                  www.latimes.com/:9
 ▶Resource interpreted as Script but transferred with MIME type text/html: "http://bs.serving-sys.com/BurstingPipe/adServer.bs?
 cn=rsb&c=28&pli=8472873&…i0KB3oxNHJyYXQQ5IoBIg4Iz9CZARIHYmx1ZWthaSIOCKrh3y8SB2hqbDlvcmgQ5IoB%26r%3D".
                                    76562-15.js?&cb=0.43121134699322283&tk_st=1&rp_s=c&p_exp=1&p_pos=atf&p_screen_res=1920x1080:2
 ▶Resource interpreted as Script but transferred with MIME type text/plain:
 "http://analytics.newsinc.com/AnalyticsProvider/jsonp/analytics/PageViewJSON…
 nc.com/Inline/iframe.html&purl=null&ssid=latimes300_hom_non_sty&anid=91002".                                     main.js:49
 ▶Uncaught TypeError: Object function a(e){if(!e)return;var t,r,i,o;n.apply(this,
 [e]),t=this.params(),r=t.size||this.dataAttr("size"),i=t.showScreenName||this.dataAttr("show-screen-
 name"),o=t.count||this.dataAttr("count"),this.classAttr.push("twitter-follow-
 button"),this.showScreenName=i!="false",this.showCount=t.showCount!==!1&&this.dataAttr("show-count")!="false",o=="none"&&
 (this.showCount=!1),this.explicitWidth=t.width||this.dataAttr("width")||"",this.screenName=t.screen_name||t.screenName||s.screenName(this.att
 r("href")),this.preview=t.preview||this.dataAttr("preview")||"",this.align=t.align||this.dataAttr("align")||"",this.size=r=="large"?"l":"m"}
 has no method 'init'                                                                                          widgets.js:45
Consider using 'dppx' units instead of 'dpi', as in CSS 'dpi' means dots-per-CSS-inch, not dots-per-physical-inch, so does not correspond to
the actual 'dpi' of a screen. In media query expression: (-webkit-min-device-pixel-ratio: 1.5), (min-resolution: 144dpi)
                                                                                               tweet_button.1392079123.html:1
Consider using 'dppx' units instead of 'dpi', as in CSS 'dpi' means dots-per-CSS-inch, not dots-per-physical-inch, so does not correspond to
the actual 'dpi' of a screen. In media query expression: (-webkit-min-device-pixel-ratio: 1.5), (min-resolution: 144dpi)
                                                                                              follow_button.1392079123.html:1
Consider using 'dppx' units instead of 'dpi', as in CSS 'dpi' means dots-per-CSS-inch, not dots-per-physical-inch, so does not correspond to
the actual 'dpi' of a screen. In media query expression: (-webkit-min-device-pixel-ratio: 1.5), (min-resolution: 144dpi)
                                                                                               tweet_button.1392079123.html:1
Invalid App Id: Must be a number or numeric string representing the application id.                               all.js:56
 ▶Resource interpreted as Image but transferred with MIME type text/html: "http://googleads.g.doubleclick.net/pagead/adview?
 ai=Cs194VXMIU4rxDYSUlAKf64…_0e_j9RqX11QUunys1tz-RIN4sAE0sbT1BaSBQYIAhABGAGgBiE&sigh=bqChBP1N63E&vis=2".            VM1660:1
TWITTER: Content Security Policy restrictions may be applied to your site. Add <meta name="twitter:widgets:csp" content="on"> to supress this
warning. undefined undefined undefined undefined                                                                widgets.js:9
TWITTER: Please note: Not all embedded timeline and embedded Tweet functionality is supported when CSP is applied. undefined undefined
undefined undefined                                                                                             widgets.js:9
 ▶Resource interpreted as Image but transferred with MIME type text/html:
 "http://widget.perfectmarket.com/wwwlatimes/view/cbcbc15b7dbf2f82f53bb61bbe5…7afd5&pd=2014-02-
 22&sc[]=nation&ov=%7B%22to%22%3A1%7D&sr=1&v=1.32&vp=20000".                                                 pmk-1.32.js:110
```

Figure 2: They said that I could become anything, so I became the error log of a Web browser. Now I own fifteen cats and I wonder where the parties are.

standards mode, except that it renders images inside table cells using the quirks mode algorithm. For reasons that have been eaten by a wildebeest, "almost standards mode" is also called "strict" mode, even though it is less strict than standards mode. For reasons so horrendous that the wildebeest would not eat them, there is no completely reliable way to make all browsers load your page using the same compatibility mode. Thus, even if your page recites the recommended incantations, the browser may still do what *it* wants to do, how *it* wants to do it. And that's where babies come from.

- The fourth and seventh errors represent uncaught JavaScript exceptions. In a rational universe, a single uncaught exception would terminate a program, and if a program continued to execute after throwing such an exception, we would know that Ragnarok is here and Odin is not happy. In the browser world, ignoring uncaught exceptions is called "Wednesday, and all days not called 'Wednesday.'" The JavaScript event loop is quite impervious to conventional notions of software reliability, so if an event handler throws an exception, the event loop will literally pretend like nothing happened and keep running. This ludicrous momentum continues even if, in the case of the seventh error, the Web page tries to call `init()` on an object that has no `init()` method. You should feel uncomfortable that a Web page can disagree with itself about the existence of initialization routines, *but the page is still allowed to do things with things*. Such a dramatic mismatch of expectations would be unacceptable in any other context. You would be sad if you went to the hospital to have your appendix removed, and the surgeon opened you up, and she said, "I DIDN'T EXPECT YOUR LIVER TO HAVE GILLS," and then she proceeded with her original surgical plan, *despite the fact that you're apparently a mer-person*. Being a mer-person should have non-ignorable ramifications in the material universe. Similarly, if a Web page thinks than an object should be initialized, but the object has no initialization method, the browser shouldn't laugh about it and then proceed under the assumption that the rest of the page is agnostic about whether its objects are composed of folly.

An interpretation of the remaining errors is left as an exercise to the reader. Note that understanding the eighth error requires a Ouija board, the eye of a newt, and the whispering of a secret to a long-lost friend.

At this point, it should be intuitively obvious that different browsers may or may not produce the same error log for the same page. In general, if a Web page has more than three bits of entropy, different browsers will generate extravagantly unique mappings between the Web developer's intentions and the schizophrenic beast palette that browsers use to paint the world. Thus, picking the "best browser" is like playing one of those horrid trust-building exercises where you decide which three of your five senses you would prefer to lose, and then your coworkers berate you for making different tradeoffs than they made, even though there is no partial ordering that relates scuba diving accidents in which you lose your ears and eyes, and industrial accidents in which you lose your nose and tongue. All options are bad options; it's a world of lateral moves. Indeed, trying to pick the best browser is like trying to decide which of your worthless children should inherit the family business. Little Oliver refuses to accept society's notion of what an event handling loop should do, so whenever the user presses a key on the keyboard, Oliver does not fire one keyPress event, but instead three keyDown events, a keyUp event, and the deleted saxophone solo from Mozart's eighth symphony. Dearest Fiona, an unrepentant workaholic, designed her browser so that when you "close" it, the GUI goes away, but the underlying process lingers in the background, silent and angry, slowly consuming entries in kernel tables and making it impossible to restart the browser without receiving the error message "Somewhere in this world, another copy of the browser is running; find Carmen Sandiego and she will reveal the truth." Beloved Christopher, in an attempt to make his browser fast and lightweight, decided to replace his Flash plugin with code that prints "Shockwave has crashed" and then immediately dereferences a NULL pointer; this ensures that most attempts to watch a video will end with you wishing for the simpler audiovisual pleasures of a woodcut or cave painting. And poor IE6, voted "Least Likely to Succeed Because IE6 Is Not a Proper Christian Name," manages to stumble through the world while surviving more assassination attempts than Fidel Castro.

Each browser is reckless and fanciful in its own way, but all browsers share a love of epic paging to disk. Not an infrequent showering of petite I/Os that are aligned on the allocation boundaries of the file system—I mean adversarial thunder-snows of reads and writes, a primordial deluge that makes you gather your kinfolk and think about which things you need two of, and what the consequences would be if you didn't bring fire ants, because fire ants ruin summers. Browsers don't require a specific reason to thrash the disk; instead, paging is a way of life for browsers, a leisure activity that is fulfilling in and of itself. If you're not a computer scientist or a tinkerer, you just accept the fact that going to CNN.com will cause the green blinky light with the cylinder icon to stay green and not blinky. However, if you know how computers work, *the incessant paging drives you mad*. It turns you into Torquemada, a wretched figure consumed by the fear that your ideological system is an elaborate lie designed to hide the excessive disk seeks of shadowy overlords. You launch your task manager, and you discover that your browser has launched 67 different processes, all of which are named "browser.exe," and all of which are launching desperate volleys of I/Os to cryptic parts of the file system like "\roaming\pots\pans\cache\4$$Dtub.partial", where "\4$$" is an exotic escape sequence that resolves to the Latvian double umlaut. You do an Internet search for potential solutions, and you're confronted with a series of contradictory, ill-founded opinions: your browser has a virus; your virus has a virus; you should be using Emacs; you should be using vi, and this is why your marriage is loveless.

Of course, the most popular advice for solving *any* browser problem is to clear your browser cache. It is definitely true that emptying the cache will sometimes help, in the same sense that if you're poor, kicking a tree will sometimes lead to a hilarious series of events that conclude with you finding a big bag of money on the ground with a note that says, "Spend it all! XOXO, Life." Unfortunately, kicking a tree does not typically lead to riches, so your faith-based act of tree assault really just makes you a savage, tree-kicking monster who will be vilified by children and emotionally sensitive adults. Similarly, your arbitrary clearing of the browser cache, however well-intentioned, is just a topical anesthetic to briefly dull the pain of existence. Clearing the cache to fix a Web browser is like when your dad was driving you to kindergarten, and the car started to smoke, and he tried to fix the car by banging on the hood three times and then asking you if you could still smell the carbon monoxide, and you said, "Yeah, it's better," because you didn't want to expose your dad as a fraud, and then both of you rode to school in silence as you struggled to remain conscious.

So, yes, it would be great if fixing your browser involved actions that were not semantically equivalent to voodoo. But, on the bright side, things could always be worse. For example, it would definitely be horrible if your browser's scripting language combined the prototype-based inheritance of Self, a quasi-functional aspect borrowed from LISP, a structured syntax adapted from C, and an aggressively asynchronous I/O model that requires elaborate callback chains that span multiple generations of hard-working Americans. OH NO I'VE JUST DESCRIBED JAVASCRIPT. What an unpleasant turn of events! People were begging for a combination of Self, LISP, and C in the same way that the denizens of Middle Earth were begging Saruman to breed Orcs and men to make Uruk-hai. Orcs and men were doing a fine job of struggling in their separate communities—creating a new race with the drawbacks of both is not a good way to win popularity contests. But despite its faults, JavaScript has become widespread. Discovering why this happened is similar to understanding the causes for World War I—everyone agrees on the top five reasons, but everyone ranks those causes differently. The basic story is that, in the '90s, when JavaScript and Java were competing for client-side supremacy, Java applets were horrendously slow and lacked a story for interacting with HTML; in contrast, JavaScript was only semi-horrendously slow, and it had a bad (but extant) story for interacting with HTML. So, Java lost, despite facts like this:

- JavaScript is dynamically typed, and its aggressive type coercion rules were apparently designed by Monty Python. For example, `12 == "12"` because the string is coerced into a number. This is a bit silly, but it kind of makes sense. Now consider the fact that `null == undefined`. That is completely janky; a reference that points to `null` is not undefined—IT IS DEFINED AS POINTING TO THE NULL VALUE. And now that you're warmed up, look at this: `"\r\n\t" == false`. Here's why: the browser detects that the two operands have different types, so it converts `false` to `0` and retries the comparison. The operands still have different types (string and number), so the browser coerces `"\r\n\t"` into the number `0`, because somehow, a non-zero number of characters is equal to `0`. Voila—`0` equals `0`! AWESOME. That explanation was like the plot to *Inception*, but the implanted idea was "the correctness of your program has been coerced to `false`."
- Hello, kind stranger—let me keep you warm during this cold winter night! Did you know that JavaScript defines a special `NaN` ("not a number") value? This value is what you get when you do foolish things like `parseInt("BatmanIsNotAnInteger")`. In other words, `NaN` is a value that is not indicative of a number. However, `typeof(NaN)` returns… "number." A more obvious return value would be "HAIL BEELZEBUB, LORD OF DARKNESS," but I digress.
- By the way, `NaN != NaN`, so Aristotle was wrong about that whole "Law of Identity" thing.
- Also, JavaScript defines two identity operators (`===` and `!==` operators) which don't perform the type coercion that the standard equality operators do; however, `NaN !== NaN`. So, basically, don't use numbers in JavaScript, and if you absolutely have to use numbers, implement a software-level ALU. It's slow, but it's the only way to be sure.
- Actually, you still can't be sure. Unlike C++, which uses statically declared class interfaces, JavaScript uses prototype-based inheritance. A prototype is a dynamically defined object which acts as an exemplar for "instances" of that object. For example, if I wanted to declare a Circle class in JavaScript, I could do something like this:

```
//This is the constructor, which defines a
//"radius" property for new instances.
function Circle(radius){
      this.radius = radius;
}

//The constructor function has an object property
//called "prototype" which defines additional
//attributes for class instances.
Circle.prototype.getDiameter = function(){
    return 2*this.radius;
};
var circle = new Circle(2);
alert(circle.getDiameter()); //Displays "4".
```

The exemplar object for the `Circle` class is `Circle.prototype`, and that prototype object is a regular JavaScript object. Thus, by dynamically changing the properties of that object, I can dynamically change the properties *of all instances of that class*. YEAH I KNOW. For example, at some random point in my program's execution, I can do this…

```
Circle.prototype.getDiameter = function(){
    return -5;
};
```

…and all of my circles will think that they have a diameter of less than nothing. That's a shame, but what's worse is that the predefined (or "native") JavaScript objects can also have their prototypes reset. So, if I do something like this…

```
Number.prototype.valueOf = function(){return 42;};
```

…then *any* number primitive that is boxed into a Number object will think that it's the answer to the ultimate question of life, the universe, and everything:

```
alert((0).valueOf());  //0 should be 0 for all values of 0,
                       //but it is 42.
alert((1).valueOf());           //Zeus help me, 1 is 42 as well.
alert((NaN).valueOf());        //NaN is 42. DECAPITATE ME AND
                                   //BURN MY WRITHING BODY WITH FIRE.
```

I obviously get what I deserve if my JavaScript library redefines native prototypes in a way that breaks my own code. However, a single frame in a Web page contains *multiple* JavaScript libraries from *multiple* origins, so who knows what kinds of horrendous prototype manipulations those heathen libraries did before my library even got to run. This is just one of the reasons why the phrase "JavaScript security" causes Bibles to burst into flames.

- Much like C, JavaScript uses semicolons to terminate many kinds of statements. However, in JavaScript, if you forget a semicolon, the JavaScript parser can automatically insert semicolons where it thinks that semicolons might ought to possibly maybe go. This sounds really helpful until you realize that *semicolons have semantic meaning*. You can't just scatter them around like you're the Johnny Appleseed of punctuation. Automatically inserting semicolons into source code is like mishearing someone over a poor cell-phone connection, and then assuming that each of the dropped words should be replaced with the phrase "your mom." This is a great way to create excitement in your interpersonal relationships, but it is not a good way to parse code. Some JavaScript libraries intentionally begin with an initial semicolon, to ensure that if the library is appended to another one (e.g., to save HTTP roundtrips

To Wash It All Away

during download), the JavaScript parser will not try to merge the last statement of the first library and the first statement of the second library into some kind of semicolon-riven statement party. Such an initial semicolon is called a "defensive semicolon." That is the saddest programming concept that I've ever heard, and I am fluent in C++.

I could go on and on about the reasons why JavaScript is a cancer upon the world. I know that there are people who like JavaScript, and I hope that these people find the mental health services that they so desperately need. I don't know all of the answers in life, but I do know all of the things which aren't the answers, and JavaScript falls into the same category as Scientology, homeopathic medicine, and making dogs wear tiny sweaters due to a misplaced belief that this is what dogs would do if they had access to looms and opposable thumbs.

In summary, Web browsers are like quantum physics: they offer probabilistic guarantees at best, and anyone who claims to fully understand them is a liar. At this stage in human development, there are big problems to solve: climate change, heart disease, the poor financial situation of Nigerian princes who want to contact you directly. With all of these problems unsolved, Web browsing is a *terrible* way to spend our time; the *last* thing that we should do is run unstable hobbyist operating systems that download strange JavaScript files from people we don't know. Instead, we should exchange information using fixed-length ASCII messages written in a statically verifiable subset of Latin, with images represented as mathematical combinations of line segments, arcs, and other timeless shapes described by dead philosophers who believed that minotaurs were real but incapable of escaping mazes. That is the kind of clear thinking that will help us defeat the space Egyptians that emerge from the StarGates. Or whatever. I'm an American and I don't really understand history, but I strongly believe that Greeks spoke Latin to defeat intergalactic Egyptians. #TeachTheControversy! Anyways, my point is that browsers are too complex to be trusted. Unfortunately, youth is always wasted on the young, and the current generation of software developers is convinced that browsers need more features, not fewer. So, we are encouraged to celebrate the fact that browsers turn our computers into little Star Wars cantinas where everyone is welcome and you can drink a blue drink if you want to drink a blue drink and if something bad happens, then maybe a Jedi will save you, and if not, HEY IT'S A STAR WARS CANTINA YESSSSS. Space cantinas are fun, but they're just a fantasy; they're just a series of outlandish details stitched together to amuse and entertain. You have to open your eyes and see that in the real, non-hyperbolic world that you actually inhabit, your browser will frequently stop playing a video and then display flashing epilepsy pixels while making the sound that TVs make in Japanese horror movies before a pasty salamander child steps out of the screen and voids your warranty. That's a thing which could actually happen, and we should wash it all away.

Fold Time to Navigate Faster

EMMA JANE WESTBY

Emma Jane Westby is an internationally renowned open source software advocate, technical author, and teacher with Drupalize.Me. In January 2010 she was recognized by the Google Diversity Program for her efforts in increasing female participation in software development. Emma has been teaching Internet technology since 2002 and is the author of *Front End Drupal* and *Drupal User's Guide*. She lives in rural Canada where she raises bees, and enjoys sipping on a nice single malt scotch. You can follow her adventures on Twitter @emmajanehw.

Recently a friend of mine was drooling over a new feature in Omni-Outliner: section folding. I was a bit confused as I've been using folding in Vim for nearly a decade. I didn't think it was all that novel. I just thought it was a sane way to quickly review, and rearrange structured documents. The first memory I have of working with folds in Vim was with DocBook. It must have been turned on by default in a `.vimrc` file that I copied and pasted as I'd always thought of it as a DocBook-specific way of working with text.

Of course, the concept of folding isn't tied to DocBook. In this article, I'll show you how to: use folding in any type of file, enable folding by default in some file types, and add a few shortcuts to making working with folds a lot easier.

What Is Folding

Folding allows you to collapse lines of a structured text document into a single line, based on an arbitrary set of rules. Once folded, the range of lines within the fold is hidden, but not removed from the document.

When collapsed, a folded line of text looks like this:

```
### Fold Methods [13 lines]------------------------------------------
```

If an entire document is folded, you will see only these "summary" lines in your document.

You can test it out inside any document with the combination of two commands: fold creation (`zf`) and text object, such as a paragraph (`ap`). The complete command is: `zfap`.

The user help manual in Vim provides an excellent description, with an ASCII art diagram! To read the manual page: `:help user-manual`, then look for the section on folding (for me it's Chapter 28). You can navigate to this chapter by positioning your cursor inside the `|vertical bars|` and pressing `CTRL-]`.

Enabling Folding

Chances are very good that folding is already available in your copy of Vim. The packages for Vim on Ubuntu and via brew on OS X both come with folding compiled into the build. It is far more likely that the document type you're editing has no fold methods available.

Fold Methods

There are a number of different ways that a folding point can be defined: for example, a text object, such as a paragraph, or the number of indents at the beginning of a line. The plugin we'll be exploring in this article uses an expression to define folds.

There are lots of plugins similar to the one I found for other file types. You might want to fold files based on function, for example, or class declarations in PHP, or perhaps patch files. A little time with your favorite search engine will probably yield at least a few relevant plugins to try out. If you're not feeling very adventurous, but you do want to know more about how Vim calculates different folding points, take a look at the manual page at `:help foldmethod`.

Markdown Folding Plugin

For this article, I was most interested in replicating OmniOutliner's folding functionality, but in Markdown files. To do this I used a plugin with a fold expression specific to Markdown syntax. You can download this plugin from GitHub at https://github.com/nelstrom/vim-markdown-folding.

Seeing as I'm using pathogen, all I needed to do to install the Markdown folding plugin was to download the plugin and place it into the directory `~/.vim/plugins`.

This plugin merely provides the definitions on what can be folded. In the case of a Markdown file, you can fold a document based on the headings. Your Markdown files can use either of the accepted syntaxes for a heading; however, if you use # to identify headings, the outline is much easier to read since there is a clue in the text as to the level of the heading.

Configuring Folding

The author of the plugin for Markdown folding recommends making a few changes to your `.vimrc` file. In addition to his configuration, I add a few of my own:

```
set nocompatible
if has("autocmd")
    filetype plugin indent on
endif
" To start, close all folds
" you will likely want this set to 1 for non-markdown files.
set foldlevelstart=0
" Use the space bar to conveniently open and close folds.
nnoremap <Space> za
vnoremap <Space> za
```

Basic Fold Commands

Assuming you're following along with the article and you've installed the Markdown folding plugin, and set the fold level to zero, you're going to have a big surprise waiting the next time you open a Markdown file in Vim. Instead of seeing your text, you'll be greeted with a list of all of the headings in your document.

The letter "z" represents the folding of a sheet of paper, just as you are folding a block of text. (I personally find my pinkie has a hard time reaching the z key, but I'm refraining from remapping everything.) You can think of the "z" as a preface to all fold-related commands.

Open and Fold

To navigate folded sections, you can use your standard navigation keys: `j` to move down a line; `k` to move up a line. When you locate a fold you want to open, use the key combination `zo` or press the space bar. To open all of the folds at once, use the key combination: `zR`. Once all of the folds have been opened, you can jump between headings using the key combinations `zj` and `zk`. To close a fold, use the key combination `zc` or press the space bar again. To close all folds, use the key combination `zM`. To reverse what's open and what's closed, use `zi`. I find this one particularly handy when switching frequently between everything open and everything closed.

Although I find "open" and "close" to be easy to remember, you may prefer using the key combination `za`. This command will do the opposite (or "alternate") of whatever the current state is, but only for the current fold. For convenience' sake, `za` is mapped to the space bar in my `.vimrc` file (and yours too if you've been following along). It will only toggle open/closed the current fold though. If you want to recursively close or open folds, you'll need to use `zc` and `zo`, respectively.

Navigating Open Folds

Assuming you're using the standard navigation keys, `j` and `k`, navigating open folds, is also a breeze. To move to the next heading, use the key combination `zj`; when traveling in the opposite direction, `zk` will take you to the end of the previous fold. Within your current fold, you can navigate to the end of the fold using `]z`, or to the beginning using `[z`.

Customizing the Folds

When I was first learning the folding commands, I found it much easier to have everything closed by default, so that I didn't forget to take advantage of folding. As I worked with files more, however, I found my preferences varied from day-to-day depending on the task I was working on.

For this article, it was best if level one headings were visible by default, but level three headings were invisible. To achieve this I tried customizing the fold start level in my `.vimrc` file (`set foldlevelstart=1`). If you're not sure what fold level is right for you, you can play around by setting `:set foldlevel=X` (where X is a number you'd like to try).

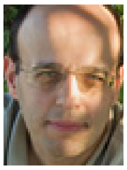
It wasn't quite what I wanted though. What I was looking for was the function `FoldToggle`. You can run the function from within Vim (`:FoldToggle`) or add the following to your `.vimrc` configuration: `let g:markdown_fold_style = 'nested'`.

In Closing

This is just starting to scratch the surface of what you can do with folding in Vim. If you didn't know about folding before, hopefully you'll be motivated to see how else (and for what other types of files) people are using this feature.

Using the R Software for Log File Analysis

MIHALIS TSOUKALOS



Mihalis Tsoukalos is a UNIX system administrator who also knows programming, databases, and mathematics. You can reach him at http://www.mtsoukalos.eu/ and @mactsouk (Twitter). mactsouk@gmail.com

In this article, I am hoping I can inspire you to use R for analyzing log files. Although I selected Apache log files to analyze, you can get your data from anywhere you can imagine including Squid log files and uptime, sar, or vmstat output.

In case you are not familiar with R [1], it is a GNU project based on S, which is a statistics-specific language and environment developed at the famous AT&T Bell Labs. I am using R primarily because it can handle huge amounts of data. I am not so sure that Microsoft Excel, for example, could process 1 million lines of data.

You may find other software that can automatically analyze Apache log files and create reports, but it is easier to generate your own customized reports by using R. And, as you are going to see later, R can deal with log files pretty easily.

The first time you try a new R command, it is better to use the R interactive shell, but keep in mind that all R commands can be put into an R script and be executed from the UNIX shell or as a cron job.

I use RStudio [2] for interacting with the R shell but you can also run R on its own. Nevertheless, RStudio is free, and my advice is to use it as it simplifies your interaction with R.

Advantages

Although R is not easy to learn, it has many advantages, including the fact that it can be used in UNIX scripts, its large number of existing packages (CRAN) [3], its outstanding graphical capabilities, its ability to process lots of data, its advanced statistical capabilities, its ability to connect to a database, and that it is a powerful programming language.

You do not have to use everything at once, but having software with so many capabilities is very convenient.

The Problem

Processing log files that contain Web server data can be a very demanding job, so I wanted a solution that is powerful, customizable, efficient, and expandable. As I am already familiar with R and comfortable with statistics, I decided to use R to do the job.

The format of my Apache log file is called "myformat" and is defined inside Apache's main configuration file (/etc/apache2/apache2.conf):

```
$ grep myformat /etc/apache2/apache2.conf
LogFormat "%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\" %D"        myformat
```

A sample entry looks as follows:

```
66.249.74.239 - - [09/Feb/2014:20:21:34 +0200] "GET  /ten-things-love-about-programming HTTP/1.1" 200
7588 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)" 1575345
```

Using the R Software for Log File Analysis

The "myformat" definition is a non-standard Apache Log-Format: it is the "combined" standard LogFormat with the addition of the %D field at the end. The %D field makes Apache record "the time taken to serve the request" in microseconds. If you are using a standard LogFormat, some of the presented examples will not work for you but you can still use another column like the eighth column that records the size of the request in bytes.

Getting the Data

Before being able to use the log file data, you must first import it into R. The good thing is that R can parse log files without requiring any additional work from you. Reading a log file named log.log is as simple as executing the following R command:

```
> LOGS = read.table("log.log", sep=" ", header=F)
```

In case your log file is not regular, you may get output that is similar to the following warning messages:

```
1: In scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings,  :
   EOF within quoted string
2:In scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings,  :
  number of items read is not a multiple of the number of columns
```

Apache log files are regular and always have the correct number of columns. If your log file is not regular, the `read.table()` function will not work; a Perl or Python script could solve such problems but you should have to decide what to do with the records with the missing fields.

I used a small log file (about 1500 lines) for the purposes of this article, but R is capable of processing huge log files without problems provided that you have a relatively modern computer.

After executing the `read.table()` command, the LOGS variable holds all data from the log.log file. The `head(LOGS)` command will show you the first few lines of the LOGS variable to get a better idea of how your data is stored by R. As there is no header line with the name of each column inside log.log, R automatically named columns as V1, V2, etc. If you want to process the data of the V10 column, you can use the LOGS$V10 variable.

I can see all the available column names by running the following command:

```
> names(LOGS)
 [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11"
```

Note that if your log file entries have a different format than mine, you may need to change some of the forthcoming R commands to match your format.

The `attach()` command creates new variables after the names of the columns of the LOGS variable, and it is used for convenience.

```
> attach(LOGS)
```

From now on, I can use V1 instead of LOGS$V1 to refer to the first column of the LOGS variable, V2 for the second column, etc.

The most important thing to consider when trying to analyze data is deciding which data you are going to analyze. I will use the following sets:

1. Column V11, which is the last field of each line in the log file.
2. Column V7, which is the Apache status code.
3. Column V4, which contains both the date and the time of each request. It will be used for counting the total number of requests per week day and per hour.
4. Columns V7, V8 (the size in bytes), and V11 will be used for having a general overview of the Web site traffic.

Analyzing the Data

Getting the data into R is not that difficult, but analyzing the data is a totally different story! This section will show some basic yet powerful R commands.

The single most useful command you can run on a data set with numeric values is the `summary()` command. The following statistical definitions will help you better understand the output of the `summary()` command:

- Min.: This is the minimum value of the whole data set.

- Median: It is an element that divides the data set into two subsets (left subset and right subset) with the same number of elements. If the data set has an odd number of elements, then the Median is part of the data set. On the other side, if the data set has an even number of elements, then the Median is the mean value of the two center elements of the data set.

- 1st Qu.: The 1st Quartile (q1) is a value that does not necessarily belong to the data set, with the property that at most 25% of the data set values are smaller than q1 and at most 75% of the data set values are bigger than q1. You can also consider it as the Median value of the left half subset of the sorted data set.

- Mean: This is the mean value of the data set (the sum of all values divided by the number of the items in the data set).

- 3rd Qu.: The 3rd Quartile (q3) is a value, not necessarily belonging to the data set, with the property that at most 75% of the data set values are smaller than q3 and at most 25% of the data set values are bigger than q3. Similarly to the 1st Quartile, you can consider the 3rd Quartile as the Median of the right half subset of the sorted data set.

- Max.: This is the maximum value found in the data set.

Using the R Software for Log File Analysis

Note that there are many ways to determine Quartiles, so if you try another statistical package, you may get slightly different results.

I definitely wanted statistics about response time values, so I ran the `summary()` command on the V11 column:

```
> summary(V11)
    Min.    1st Qu.    Median     Mean     3rd Qu.     Max.
    474      787      12760    507800    905200    9101000
```

The output shows that the 25% of requests are served extremely fast (the value of the 1st Quartile is very small). The other good thing is that the value of the Median is also small. It does make sense for some requests to take longer since the Web site serves images and also has an Administrator Interface which asks the MySQL database to run complex queries that take longer to execute than regular user visits.

I also wanted to know the total number of status codes, so I ran the following command using the V7 column:

```
> table(LOGS[,7])

    200    302    304    403    404    500
    943     2     162     71     12      8
```

The output shows that most requests are served without problems so, generally speaking, the Web site works well. Having a large number of 404 and 403 status codes is not a good thing.

Visualizing the Data

I ran the following command which creates a very simple bar plot for the status code data (Figure 1):

```
> barplot(table(LOGS[,7]))
```

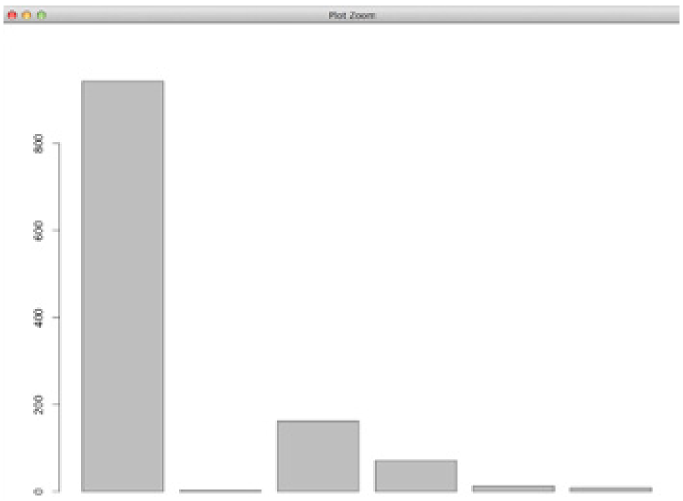


Figure 1: Creating a bar plot for the status code data

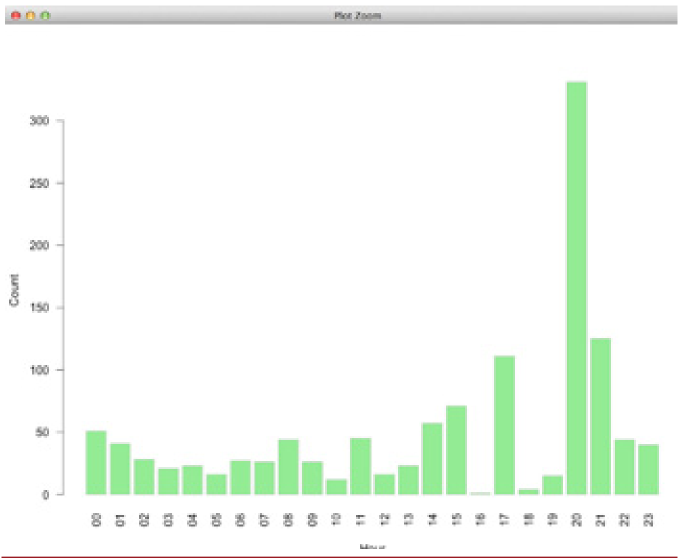


Figure 2: Traffic per hour

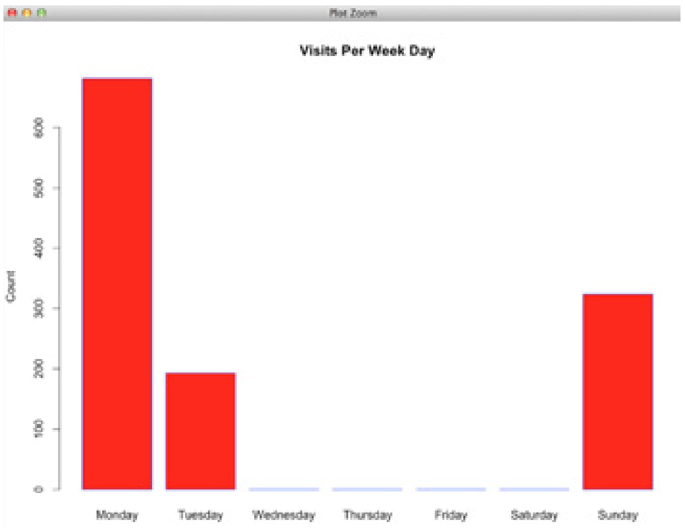


Figure 3: Traffic per weekday

If you want R to save the bar plot to an image which is 1024 x 1024 pixels, you should run the following R commands instead:

```
> png('test.png', width=1024, height=1024)
> barplot(table(LOGS[,7]))
> dev.off()
```

Next, I wanted to visualize the number of requests per week day and per hour of the day. Getting the date and time from the V4 column is a little tricky but it can be done using the following command:

```
> newV4 <- strptime(V4 , format('[%d/%b/%Y:%H:%M:%S'))
```

Now, I could use the new variable (newV4) to create plots that showed the traffic per hour (Figure 2) and per weekday (Figure 3) as follows:

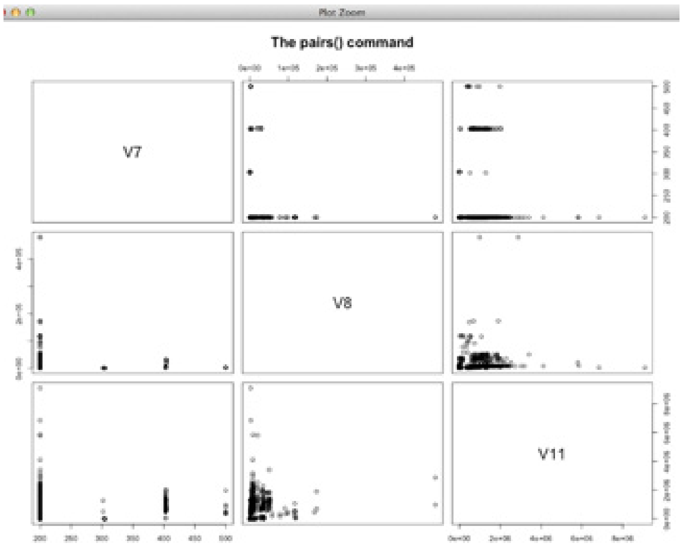


Figure 4: The pairs() command

```
> day = format(newV4, "%A")
> head(day)
[1] "Sunday" "Sunday" "Sunday" "Sunday" "Sunday" "Sunday"
> hours = format(newV4, "%H")
> head(hours)
[1] "19" "19" "19" "19" "19" "19"
barplot( table(factor(day, levels=c("Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "Sunday"))), xlab="Day", ylab="Count",
col="red", border="blue", main='Visits Per Weekday')
> barplot(table(hours), xlab="Hour", ylab="Count", col="lightgreen",
las=2, border="gray")
```

The `pairs()` command is especially useful since it gives a general overview of the data. I will only use three columns, but you can use as many columns as you want. Figure 4 is generated using the following commands.

```
> tempLOGS <- LOGS
> tempLOGS[1:6] <- list(NULL)
> names(tempLOGS)
[1] "V7"  "V8"  "V9"  "V10" "V11"
> tempLOGS$V9 <- NULL
> tempLOGS$V10 <- NULL
> names(tempLOGS)
[1] "V7"  "V8"  "V11"
> pairs(tempLOGS, main="The pairs() command")
```

The `tempLOGS <- LOGS` command creates a copy of the LOGS variable into the tempLOGS variable. The `tempLOGS$V9 <- NULL` command empties the V9 column of the tempLOGS table, which practically erases the V9 column of the tempLOGS variable. The `tempLOGS[1:6] <- list(NULL)` command empties the first six columns of the table, which practically deletes them from the tempLOGS variable.

Outliers in the graphical output is a sign of unusual or hostile behavior and should be further examined.

Using R for Detecting Security Threats

I can also check for security threats using the R language. As my log file is from a Drupal site, I want to monitor the "GET /?q=node/add HTTP/1.1", "GET /?q=user/register HTTP/1.1", "GET /?q=node/add HTTP/1.0", and "GET /?q=user/register HTTP/1.0" requests that indicate hack attempts. To do so, I ran the following R commands that use the V4 and V6 columns:

```
> HACK = subset(LOGS, V6 %in% c("GET /?q=node/add HTTP/1.1", "GET
/?q=user/register HTTP/1.1", "GET /?q=node/add HTTP/1.0", "GET
/?q=user/register HTTP/1.0" ))
> names(HACK)
[1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11"
> HACK[1:3] <- list(NULL)
> names(HACK)
[1] "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11"
> HACK$V5 <- NULL
> names(HACK)
[1] "V4"  "V6"  "V7"  "V8"  "V9"  "V10" "V11"
> HACK[3:5] <- list(NULL)
> names(HACK)
[1] "V4"  "V6"  "V10" "V11"
> HACK[3:4] <- list(NULL)
> names(HACK)
[1] "V4" "V6"
> head(HACK)
                       V4                          V6
253 [09/Feb/2014:20:47:47      GET /?q=node/add HTTP/1.0
254 [09/Feb/2014:20:47:48 GET /?q=user/register HTTP/1.0
257 [09/Feb/2014:20:50:44      GET /?q=node/add HTTP/1.0
258 [09/Feb/2014:20:50:50 GET /?q=user/register HTTP/1.0
271 [09/Feb/2014:21:41:00      GET /?q=node/add HTTP/1.0
272 [09/Feb/2014:21:41:01 GET /?q=user/register HTTP/1.0

> summary(HACK)
                    V4                             V6
 [10/Feb/2014:17:59:58: 2   GET /?q=node/add HTTP/1.0     :41
 [11/Feb/2014:04:16:12: 2   GET /?q=user/register HTTP/1.0:41
 [09/Feb/2014:20:47:47: 1   GET /?q=user/register HTTP/1.1:10
 [09/Feb/2014:20:47:48: 1   GET /?q=node/add HTTP/1.1     : 7
 [09/Feb/2014:20:50:44: 1   GET / HTTP/1.0                : 0
 [09/Feb/2014:20:50:50: 1   GET / HTTP/1.1                : 0
 (Other)              :91   (Other)                       : 0

> newV4 <- strptime(HACK$V4 , format('[%d/%b/%Y:%H:%M:%S'))
> day = format(newV4, "%A")
> barplot( table(factor(day, levels=c("Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "Sunday"))), xlab="Day", ylab="Count",
col="red", border="blue", main="Hack Attempts!")
```

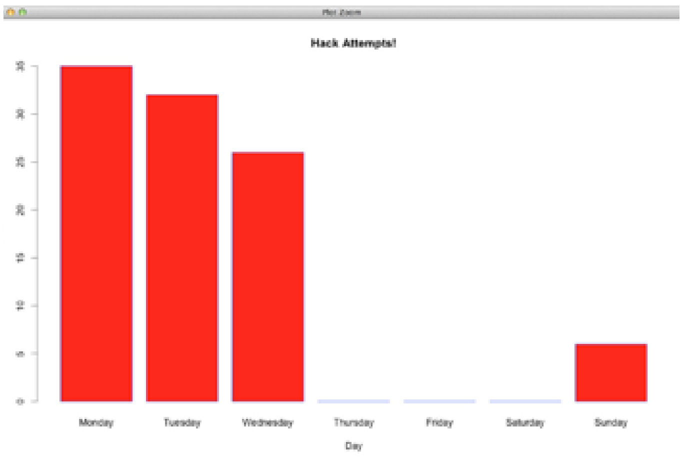


Figure 5: Counting hacking attempts

The produced bar plot can be seen in Figure 5.

Occasionally, if there are many hacking attempts, I may search a little more and find the IP—using column V1— that had the highest number of hacking attempts in order to add it to my firewall's deny list.

R offers many more packages that can help you analyze and visualize your data, but you have to decide what is best for your data and your needs and use it. Experimentation is a key part of the process that I also happen to find very interesting!

Conclusion

I think that you should definitely learn an advanced statistics package such as R because it will make you see your data in a totally different way. It will also help you create graphical output that is really unique and different from all those Excel charts.

I hope this article will be the beginning of your journey to R!

References

[1] R Project home page: http://www.r-project.org/.

[2] RStudio IDE: http://www.rstudio.com/.

[3] CRAN: http://cran.r-project.org/.