hacker bits

November 2016

new bits

What if working less actually increases productivity?

Itamar Turner-Trauring makes the case for working less in this month's most highly-upvoted article.

Are you an old geek? Are twenty-something co-workers making you feel over the hill?

If you've ever been told that your age makes you a "poor cultural fit" for a company, then don't miss IT veteran Tim Bray's insightful article on Silicon Valley's rampant ageism.

Entrepreneur John Wheeler also has an excellent follow-up article and solution for all old geeks out there. Check it out: oldgeekjobs.com.

We are also thrilled to have the creator of SQLite, Dr. Richard Hipp, break down the pros and cons of SQLite for us.

Lastly, a big welcome to our new subscribers! Thanks for reading *Hacker Bits* and remember, we are always here to help you learn more, read less and stay current!

— Maureen and Ray

us@hackerbits.com

content bits

November 2016

contributor bits

Richard Hipp
Dr. Richard Hipp began the SQLite project on 2000-05-29 and continues to serve as the project architect. Richard was born, lives, and works in Charlotte, NC. He holds degrees from Georgia Tech (MSEE 1984) and Duke University (PhD 1992), and is the founder of the consulting firm Hwaci.

Tim Bray
Tim is a software developer and entrepreneur. He is one of the co-authors of the original XML specification. He has worked for Amazon Web Services since Dec-2014 and previously at Google, Sun Microsystems and DEC.

Itamar Turner-Trauring
Itamar is working on a book called *The Programmer's Guide to a Sane Workweek*, and sharing his many mistakes as a software engineer at Software Clown.

Ozan Onay
Ozan is an instructor at Bradfield, a San Francisco based school focused on foundational computer science. Previously he was CTO of Vida Health and Topguest (acquired by Switchfly).

Liz Bennett
By day, Liz is a software engineer writing scalable backend services in Java. By night, she is a knitter/gamer/web dev tinkerer who enjoys pondering engineering philosophy and developer productivity.

Vlad Zelinschi
Vlad is a pragmatic frontend engineer and an avid caffeine consumer. He loves surrounding himself with ever smiling people and he's constantly pushing his limits in order to become a better professional.

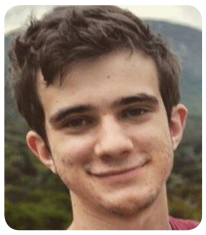
John Wheeler
John has been developing for two decades in Java, JavaScript, Python, C#, and Ruby. He's founded several SaaS businesses for eBay sellers and written technical articles for IBM and O'Reilly. He lives in San Diego with his wife, daughter, and twin boys.

Justin Vincent
Justin is founder of Nugget.one, a startup incubator & community that sends a new startup idea to your inbox every single day.

Tim Bray image by Tim Bray (commons: uploader user: ims) [GFDL or CC-BY-SA-3.0], via Wikimedia Commons

Alex Martins

Alex currently works as a software engineer for The New York Times; previously he was a lead engineer at Globo.com, helping shape the technology vision and architecture of the live video streaming platform. Alex also worked as a Consultant at ThoughtWorks helping businesses in Australia and Asia become ready for the connected world.



Frederico Bittencourt

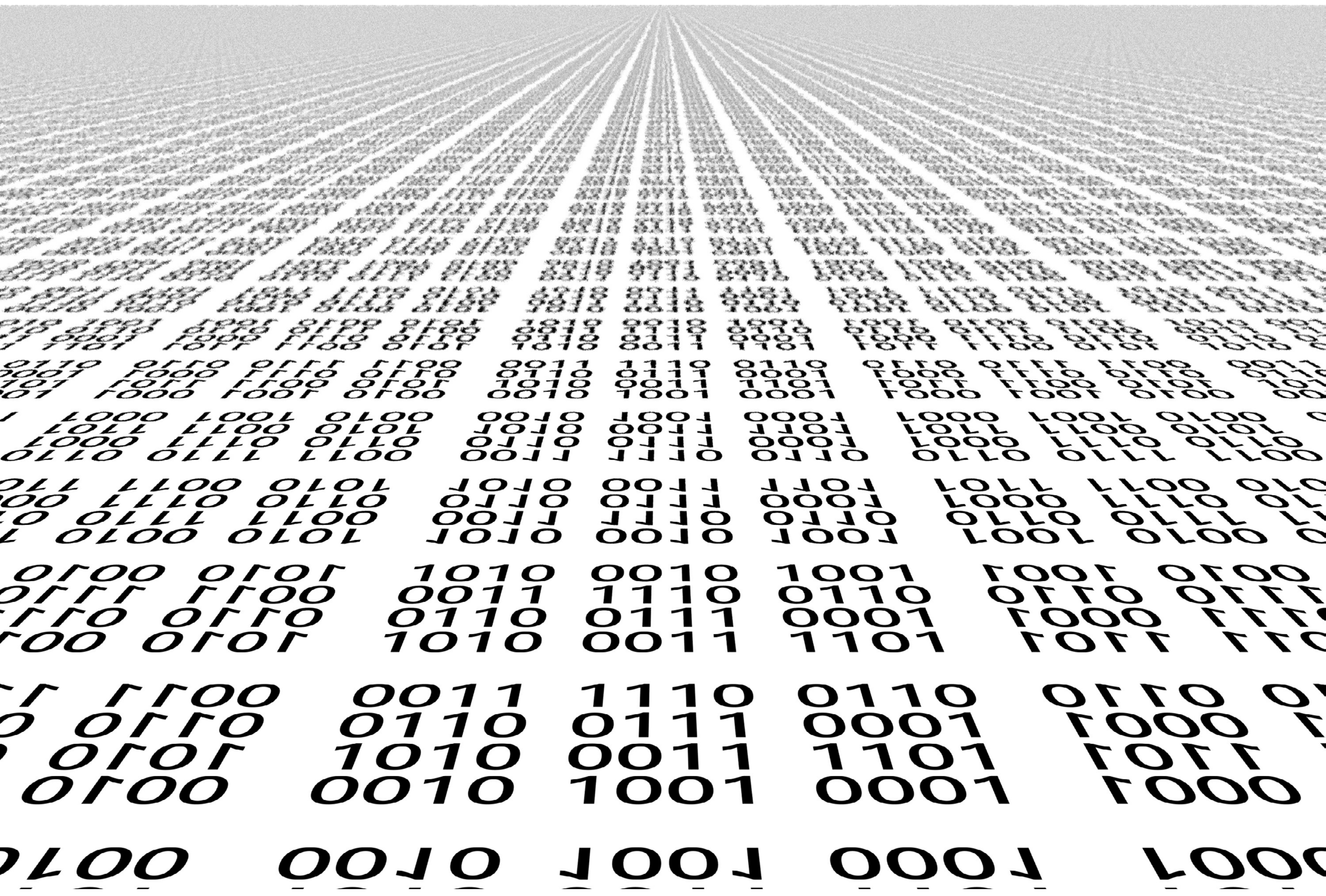
Frederico is a computer science student from Brazil. He is 21 years old and is currently working as a software developer at SAP. Besides his interest in technology and programming, he is also an open source enthusiast. Github: https://github.com/fredrb/

Ray Li
Curator

Ray is a software engineer and data enthusiast who has been blogging at rayli.net for over a decade. He loves to learn, teach and grow. You'll usually find him wrangling data, programming and lifehacking.

Maureen Ker
Editor

Maureen is an editor, writer, enthusiastic cook and prolific collector of useless kitchen gadgets. She is the author of 3 books and 100+ articles. Her work has appeared in the *New York Daily News*, and various adult and children's publications.



Appropriate uses for SQLite

By RICHARD HIPP

SQLite is not directly comparable to client/server SQL database engines such as MySQL, Oracle, PostgreSQL, or SQL Server since SQLite is trying to solve a different problem.

Client/server SQL database engines strive to implement a shared repository of enterprise data. They emphasis scalability, concurrency, centralization, and control. SQLite strives to provide local data storage for individual applications and devices. SQLite emphasizes economy, efficiency, reliability, independence, and simplicity.

SQLite does not compete with client/server databases. SQLite competes with fopen().

Situations Where SQLite Works Well

Embedded devices and the Internet of things

Because an SQLite database requires no administration, it works well in devices that must operate without expert human support. SQLite is a good fit for use in cellphones, set-top boxes, televisions, game consoles, cameras, watches, kitchen appliances, thermostats, automobiles, machine tools, airplanes, remote sensors, drones, medical devices, and robots: the "internet of things".

Client/server database engines are designed to live inside a lovingly-attended datacenter at the core of the network. SQLite works there too, but SQLite also thrives at the edge of the network, fending for itself while providing fast and reliable data services to applications that would otherwise have dodgy connectivity.

Application file format

SQLite is often used as the on-disk file format for desktop applications such as version control systems, financial analysis tools, media cataloging and editing suites, CAD packages, record keeping programs, and so forth. The traditional File/Open operation calls sqlite3_open() to attach to the database file. Updates happen automatically as application content is revised so the File/Save menu option becomes superfluous. The File/Save_As menu option can be implemented using the backup API.

There are many benefits to this approach, including improved application performance, reduced cost and complexity, and improved reliability. See technical notes here and here for details.

Websites

SQLite works great as the database engine for most low to medium traffic websites (which is to say, most websites). The amount of web traffic that SQLite can handle depends on how heavily the website uses its database. Generally speaking, any site that gets fewer than 100K hits/day should work fine with SQLite. The 100K hits/day figure is a conservative estimate, not a hard upper bound. SQLite has been demonstrated to work with 10 times that amount of traffic.

The SQLite website (https://www.sqlite.org/) uses SQLite itself, of course, and as of this writing (2015) it handles about 400K to 500K HTTP requests per day, about 15-20% of which are dynamic pages touching the database. Each dynamic page does roughly 200 SQL statements. This setup runs on a single VM that shares a physical server with 23 others and yet still keeps the load average below 0.1 most of the time.

Data analysis

People who understand SQL can employ the sqlite3 command-line shell (or various third-party SQLite access programs) to analyze large datasets. Raw data can be imported from CSV files, then that data can be sliced and diced to generate a myriad of summary reports. More complex analysis can be done using simple scripts written in Tcl or Python (both of which come with SQLite built-in) or in R or other languages using readily available adaptors. Possible uses include website log analysis, sports statistics analysis, compilation of programming metrics, and analysis of experimental results. Many bioinformatics researchers use SQLite in this way.

The same thing can be done with an enterprise client/server database, of course. The advantage of SQLite is that it is easier to install and use and the resulting database is a single file that can be written to a USB memory stick or emailed to a colleague.

Cache for enterprise data

Many applications use SQLite as a cache of relevant content from an enterprise RDBMS. This reduces latency, since most queries now occur against the local cache and avoid a network round-trip. It also reduces the load on the network and on the central database server. And in many cases, it means that the client-side application can continue operating during network outages.

Server-side database

Systems designers report success using SQLite as a data store on server applications running in the datacenter, or in other words, using SQLite as the underlying storage engine for an application-specific database server.

With this pattern, the overall system is still client/server: clients send requests to the server and get back replies over the network. But instead of sending generic SQL and getting back raw table content, the client requests and server responses are high-level and application-specific. The server translates requests into multiple SQL queries, gathers the results, does post-processing, filtering, and analysis, then constructs a high-level reply containing only the essential information.

Developers report that SQLite is often faster than a client/server SQL database engine in this scenario. Database requests are serialized by the server, so concurrency is not an issue. Concurrency is also improved by “database sharding”: using separate database files for different subdomains. For example, the server might have a separate SQLite database for each user, so that the server can handle hundreds or thousands of simultaneous connections, but each SQLite database is only used by one connection.

File archives

The SQLite Archiver project shows how SQLite can be used as a substitute for ZIP archives or Tarballs. An archive of files stored in SQLite is only very slightly larger, and in some cases actually smaller, than the equivalent ZIP archive. And an SQLite archive features incremental and atomic updating and the ability to store much richer metadata.

SQLite archives are useful as the distribution format for software or content updates that are broadcast to many clients. Variations on this idea are used, for example, to transmit TV programming guides to set-top boxes and to send over-the-air updates to vehicle navigation systems.

Replacement for ad hoc disk files

Many programs use fopen(), fread(), and fwrite() to create and manage files of data in home-grown formats. SQLite works particularly well as a replacement for these *ad hoc* data files.

Internal or temporary databases

For programs that have a lot of data that must be sifted and sorted in diverse ways, it is often easier and quicker to load the data into an in-memory SQLite database and use queries with joins and ORDER BY clauses to extract the data in the form and order needed rather than to try to code the same operations manually. Using an SQL database internally in this way also gives the program greater flexibility since new columns and indices can be added without having to recode every query.

Stand-in for an enterprise database during demos or testing

Client applications typically use a generic database interface that allows connections to various SQL database engines. It makes good sense to include SQLite in the mix of supported databases and to statically link the SQLite engine in with the client. That way the client program can be used standalone with an SQLite data file for testing or for demonstrations.

Education and Training

Because it is simple to setup and use (installation is trivial: just copy the **sqlite3** or **sqlite3.exe** executable to the target machine and run it) SQLite makes a good database engine for use in teaching SQL. Students can easily create as many databases as they like and can email databases to the instructor for comments or grading. For more advanced students who are interested in studying how an RDBMS is implemented, the modular and well-commented and documented SQLite code can serve as a good basis.

Experimental SQL language extensions

The simple, modular design of SQLite makes it a good platform for prototyping new, experimental database language features or ideas.

Situations Where A Client/Server RDBMS May Work Better

Client/Server Applications

If there are many client programs sending SQL to the same database over a network, then use a client/server database engine instead of SQLite. SQLite will work over a network filesystem, but because of the latency associated with most network filesystems, performance will not be great. Also, file locking logic is buggy in many network filesystem implementations (on both Unix and Windows). If file locking does not work correctly, two or more clients might try to modify the same part of the same database at the same time, resulting in corruption. Because this problem results from bugs in the underlying filesystem implementation, there is nothing SQLite can do to prevent it.

A good rule of thumb is to avoid using SQLite in situations where the same database will be accessed directly (without an intervening application server) and simultaneously from many computers over a network.

High-volume Websites

SQLite will normally work fine as the database backend to a website. But if the website is write-intensive or is so busy that it requires multiple servers, then consider using an enterprise-class client/server database engine instead of SQLite.

Very large datasets

An SQLite database is limited in size to 140 terabytes (2^{47} bytes, 128 tibibytes). And even if it could handle larger databases, SQLite stores the entire database in a single disk file and many filesystems limit the maximum size of files to something less than this. So if you are contemplating databases of this magnitude, you would do well to consider using a client/server database engine that spreads its content across multiple disk files, and perhaps across multiple volumes.

High Concurrency

SQLite supports an unlimited number of simultaneous readers, but it will only allow one writer at any instant in time. For many situations, this is not a problem. Writer queue up. Each application does its database work quickly and moves on, and no lock lasts for more than a few dozen milliseconds. But there are some applications that require more concurrency, and those applications may need to seek a different solution.

Checklist For Choosing The Right Database Engine

1. Is the data separated from the application by a network? → choose client/server

Relational database engines act as bandwidth-reducing data filters. So it is best to keep the database engine and the data on the same physical device so that the high-bandwidth engine-to-disk link does not have to traverse the network, only the lower-bandwidth application-to-engine link.

But SQLite is built into the application. So if the data is on a separate device from the application, it is required that the higher bandwidth engine-to-disk link be across the network. This works, but it is suboptimal. Hence, it is usually better to select a client/server database engine when the data is on a separate device from the application.

Nota Bene: In this rule, "application" means the code that issues SQL statements. If the "application" is an application server and if the content resides on the same physical machine as the application server, then SQLite might still be appropriate even though the end user is another network hop away.

2. Many concurrent writers? → choose client/server

If many threads and/or processes need to write the database at the same instant (and they cannot queue up and take turns) then it is best to select a database engine that supports that capability, which always means a client/server database engine.

SQLite only supports one writer at a time per database file. But in most cases, a write transaction only takes milliseconds and so multiple writers can simply take turns. SQLite will handle more write concurrency that many people suspect. Nevertheless, client/server database systems, because they have a long-running server process at hand to coordinate access, can usually handle far more write concurrency than SQLite ever will.

3. Big data? → choose client/server

If your data will grow to a size that you are uncomfortable or unable to fit into a single disk file, then you should select a solution other than SQLite. SQLite supports databases up to 140 terabytes in size, assuming you can find a disk drive and filesystem that will support 140-terabyte files. Even so, when the size of the content looks like it might creep into the terabyte range, it would be good to consider a centralized client/server database.

4. Otherwise → choose SQLite!

For device-local storage with low writer concurrency and less than a terabyte of content, SQLite is almost always a better solution. SQLite is fast and reliable and it requires no configuration or maintenance. It keeps thing simple. SQLite "just works". ■

Old geek

By TIM BRAY

I'm one. We're not exactly common on the ground; my profession, apparently not content with having excluded a whole gender, is mostly doing without the services of a couple of generations.

This was provoked by a post from James Gosling, which I'll reproduce because it was on Facebook and I don't care to learn how to link into there:

> *Almost every old friend of mine is screwed. In the time between Sun and LRI I'd get lines like "We normally don't hire people your age, but in your case we might make an exception". In my brief stint at Google I had several guys in their 30s talk about cosmetic surgery. It's all tragically crazy.*

He'd linked to It's Tough Being Over 40 in Silicon Valley, by Carol Hymowitz and Robert Burson on Bloomberg. It's saddening, especially the part about trying to look younger.

I've seen it at home, too; my wife Lauren is among the most awesome project managers I've known, is proficient with several software technologies, and is polished and professional. While she has an OK consulting biz, she occasionally sees a full-time job that looks interesting. But these days usually doesn't bother reaching out; 40-plus women are basically not employable in the technology sector.

On the other hand

To be fair, not everyone wants to go on programming into their life's second half. To start with, managers and marketers make more money. Also, lots of places make developers sit in rows in poorly-lit poorly-ventilated spaces, with not an atom of peace or privacy. And then, who, male or female, wants to work where there are hardly any women?

And even if you do want to stay technical, and even if you're a superb coder, chances are that after two or three decades of seniority you're going to make a bigger contribution helping other people out, reviewing designs, running task forces, advising executives, and so on.

Finally, there's a bad thing that can happen: If you help build something important and impactful, call it X, it's easy to slip into year after year of being the world's greatest expert on X, and when X isn't important and impactful any more, you're in a bad place.

But having said all that, Bay Area tech culture totally has a blind spot, just another part of their great diversity suckage. It's hurting them as much as all the demographics they exclude, but apparently not enough to motivate serious action.

Can old folks code?

I don't know about the rest of the world, but they can at Amazon and Google. There are all these little communities at Google: Gayglers, Jewglers, and my favorite, the Greyglers; that's the only T-shirt I took with me and still wear. The Greyglers are led by Vint Cerf, who holds wine-and-cheese events (*good* wine, *good* cheese) when he visits Mountain View from his regular DC digs. I'm not claiming it's a big population, but includes people who are doing serious shit with core technology that you use every day.

There's no equivalent at Amazon, but there is the community of Principal Engineers (I'm one), a tiny tribe in Amazon's huge engineering army. There are a few fresh-faced youthful PEs, but on average we tend to grizzle and sag more than just a bit. And if you're a group trying to do something serious, it's expected you'll have a PE advising or mentoring or even better, coding.

Like I do. Right now there's code I wrote matching and routing millions and millions of Events every day, which makes me happy.

Not that *that* much of my time goes into it — in fact, I helped Events more with planning and politicking than coding. But a few weeks ago I got an idea for another project I'd been helping out with, a relatively cheap, fast way to do something that isn't in the "Minimum Viable Product" that always ships, but would be super-useful. I decided it would be easier to build it than convince someone else, so… well, it turned out that I had to invent an intermediate language, and a parser for it, and I haven't been blogging and, erm, seem a little short on sleep.

Advice

Are you getting middle-aged-or-later and find you still like code? I think what's most helped me hang on is my attention span, comparable to a gnat's. I get bored really, really fast and so I'm always wandering away from what I was just doing and poking curiously at the new shiny.

On top of which I've been extra lucky. The evidence says my taste is super-mainstream, whatever it is I find shiny is likely to appeal to millions of others too.

Anyhow, I don't usually wear a T-shirt, but when I do, it's this one. ■

"time is money" by ewvasquez2001 is licensed under CC BY 2.0

Less stress, more productivity

By ITAMAR TURNER-TRAURING

There's always too much work to be done on software projects, too many features to implement, too many bugs to fix. Some days you're just not going through the backlog fast enough, you're not producing enough code, and it's taking too long to fix a seemingly-impossible bug. And to make things worse you're wasting time in pointless meetings instead of getting work done.

Once it gets bad enough you can find yourself always scrambling, working overtime just to keep up. Pretty soon it's just expected, and you need to be available to answer emails at all hours even when there are no emergencies. You're tired and burnt out and there's still just as much work as before.

The real solution is not working even harder or even longer, but rather the complete opposite: working fewer hours.

Some caveats first:

- The more experienced you are the better this will work. If this is your first year working after school you may need to just deal with it until you can find a better job, which you should do ASAP.
- Working fewer hours is effectively a new deal you are negotiating with your employer. If you're living from paycheck to paycheck you have no negotiating leverage, so the first thing you need to do is make sure you have some savings in the bank.

Fewer hours, more productivity

Why does working longer hours not improve the situation? Because working longer makes you less productive at the same time that it encourages bad practices by your boss. Working fewer hours does the opposite.

1. A shorter work-week improves your ability to focus

As I've discussed before, working while tired is counter-productive. It takes longer and longer to solve problems, and you very quickly hit the point of diminishing returns. And working *consistently* for long hours is even worse for your mental focus, since you will quickly burn out.

Long hours: "It's 5 o'clock and I should be done with work, but I just need to finish this problem, just one more try," you tell yourself. But being tired it actually takes you another three hours to solve. The next day you go to work tired and unfocused.

Shorter hours: "It's 5 o'clock and I wish I had this fixed, but I guess I'll try tomorrow morning." The next morning, refreshed, you solve the problem in 10 minutes.

2. A shorter work-week promotes smarter solutions

Working longer hours encourages bad programming habits: you start thinking that the way to solve problems is just forcing yourself to get through the work. But programming is all about automation, about building abstractions to *reduce* work. Often you can get huge reductions in effort by figuring out a better way to implement an API, or that a particular piece of functionality is not actually necessary.

Let's imagine your boss hands you a task that *must ship* to your customer in 2 weeks. And you estimate that optimistically it will take you 3 weeks to implement.

Long hours: "This needs to ship in two weeks, but I think it's 120 hours to complete... so I guess I'm working evenings and weekends again." You end up even more burnt out, and probably the feature will still ship late.

Shorter hours: "I've got two weeks, but this is way too much work. What can I do to reduce the scope? Guess I'll spend a couple hours thinking about it."
And soon: "Oh, if I do this restructuring I can get 80% of the feature done in one week, and that'll probably keep the customer happy until I finish the rest. And even if I underestimated I've still got the second week to get that part done."

3. A shorter work-week discourages bad management practices

If your response to any issue is to work longer hours you are encouraging bad management practices. You are effectively telling your manager that your time is not valuable, and that they need not prioritize accordingly.

Long hours: If your manager isn't sure whether you should go to a meeting, they might tell themselves that "it might waste an hour of time, but they'll just work an extra hour in the evening to make it up." If your manager can't decide between two features, they'll just hand you both instead of making a hard decision.

Shorter hours: With shorter hours your time becomes more scarce and valuable. If your manager is at all reasonable less important meetings will get skipped and more important features will be prioritized.

Getting to fewer hours

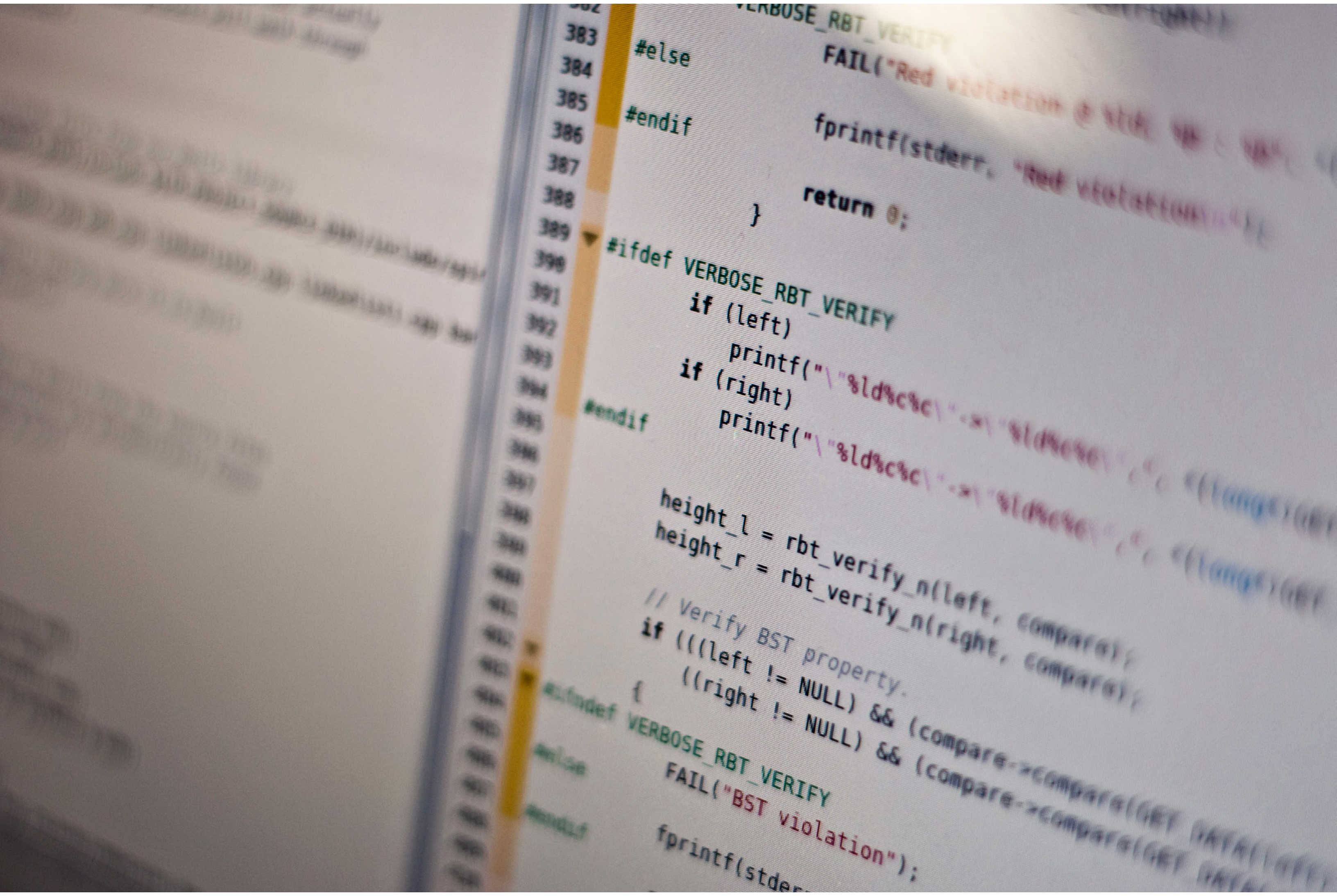
A short work-week mean different things to different people. One programmer I know made clear when she started a job at a startup that she worked 40-45 hours a week and that's it. Everyone else worked much longer hours, but that was her personal limit. Personally I have negotiated a 35-hour work week.

Whatever the number that makes sense to you, the key is to clearly explain your limits and then stick to them. Tell you manager "I am going to be working a 40-hour work week, unless it's a real emergency." Once you've explained your limits you need to stick to them: no answering emails after hours, no agreeing to do just one little thing on the weekend.

And then you need to prove yourself by still being productive, and making sure that when you are working you are *working*. Spending a couple hours a day at work watching cat videos probably won't go well with shorter hours.

There are companies where this won't fly, of course, where management is so bad or norms are so out of whack that even a 40-hour work week by a productive team member won't be acceptable. In those cases you need to look for a new job, and as part of the interview figure out the work culture and project management practices of prospective employers. Do people work short hours or long hours? Is everything always on fire or do projects get delivered on time?

Whether you're negotiating your hours at your existing job or at a new job, you'll do better the more experienced and skilled of a programmer you are. If you want to learn how to get there check out The Programmer's Guide to a Sane Workweek. ■



```
#else          FAIL("Red violation
#endif         fprintf(stderr, "Red violation
               return 0;
           }
#ifdef VERBOSE_RBT_VERIFY
      if (left)
          printf("\"%ld%c%c\"->\"%ld%c%c
      if (right)
          printf("\"%ld%c%c\"->\"%ld%c%c
#endif

      height_l = rbt_verify_n(left, compare);
      height_r = rbt_verify_n(right, compare);
      // Verify BST property.
      if (((left != NULL) && (compare->compare(GET_DATA(left
          ((right != NULL) && (compare->compare(GET_DATA
      {
#ifndef VERBOSE_RBT_VERIFY
          FAIL("BST violation");
#else
          fprintf(stder
#endif
```

"Code" by Michael Himbeault is licensed under CC BY 2.0

The cost of forsaking C

By OZAN ONAY

The C programming language is not trendy. The *most recent* edition of the canonical C text (the excitingly named *The C Programming Language*) was published in 1988; C is so unfashionable that the authors have neglected to update it in light of 30 years of progress in software engineering. Everyone "has been meaning to" learn Rust or Go or Clojure over a weekend, not C. There isn't even a cute C animal in C's non-logo on a C decal not stuck to your laptop.

But Myles and I are not trendy people, so we insist that all of our students become fluent in C. A fresh class of C converts has just finished working through the K&R bible, making this a good time for me to reflect on why we deify this ancient tongue.

We give students four reasons for learning C:

1. It is still one of the most commonly used languages outside of the Bay Area web/mobile startup echo chamber;
2. C's influence can be seen in many modern languages;
3. C helps you think like a computer; and,
4. Most tools for writing software are written in C (or C++)

The first is easy to dismiss if one *likes* the Bay Area web/mobile startup echo chamber, the second if one *hates* C's influence on many more modern languages. Most engineers should take heed of reason three, although our students also learn computer architecture and at least one assembly language, so have a firm mental model of how computers actually compute. But reason four is hard to ignore.

Forsaking C means forsaking anything below the level of abstraction at which one happens to currently work. Those who work for instance as web developers forsake thoroughly understanding the browsers, operating systems and languages on top of which their own work stands.

Concretely:

Most of our students use **interpreted languages with popular implementations written in C**. One exercise we do is to write a Python bytecode interpreter to better understand stack machines and interpreted languages; doing so involves reading the CPython implementation closely. The ruby reference implementation is also written in C, and most JavaScript implementations are written in C++.

When learning about **common data structures** such as hashmaps and dynamic arrays, we need to either implement these ourselves in a language where we can think concretely about memory layout, or to read good implementations that do so. For a student to understand how a Python list or ruby array works, we have them either write a version from scratch or read the source of the standard library implementations—gaining the same level of understanding without diving into C is almost impossible.

Learning about **operating systems** would be much harder without C. The operating systems that we use are mostly written in C, the C standard library is tightly coupled to the syscall interface, and most resources dealing with operating systems concepts assume knowledge of C.

While it is certainly possible to learn about **computer networking** without being fluent in C, practitioners who need to understand **their operating system's TCP/IP stack** will find themselves grasping for C.

Lastly most of the work we do with databases, key value stores, message queues and other distributed systems technologies mandates C, for performance reasons.

Many practicing software engineers do fine work without fully understanding the above. Routine work in a narrow domain may not demand foundational knowledge. But our students strive to do novel, high-impact work, and quickly find that a solid understanding of C is a prerequisite. If you have similar goals, I would encourage you to put aside that trendy new language for a few more weekends and brush up on good old C.

Want to peel back a layer on your understanding of computer science? Enrollment is open for the next cohort at Bradfield. ■

Why learning Angular 2 was excruciating

By LIZ BENNETT

A few months ago I decided to try my hand at web development. Angular 2 was new and shiny at the moment, so I decided to download it and start building a website. Now, as someone who is primarily a Java developer, this was an altogether new and very interesting experience. I followed the alleged "5 Minute Quick Start Guide" and after an hour and a half of wrangling with Angular 2 and its plethora of dependencies, I was able to get something up and running.

Next, I started to build a real app. I decided to build a personal blog platform from scratch, mostly for the learning, and partially because I've been meaning to start a blog for ages. The desire to have a blog would be the carrot keeping me motivated to learn new technologies and keep building my project.

Over the months, I've been keeping up with the Angular 2 releases and, each weekend, I've been chugging along on the blog. Oh, did I say chugging? I meant banging my head vigorously against the wall over and over and over again trying to understand and deal with the freaking Javascript ecosystem.

Maybe I'm just used to the slow paced, sturdy, battle tested libraries of the Java world. Maybe learning web development starting with Angular 2 is like trying to learn a new video game and starting on hard core mode. I don't know.

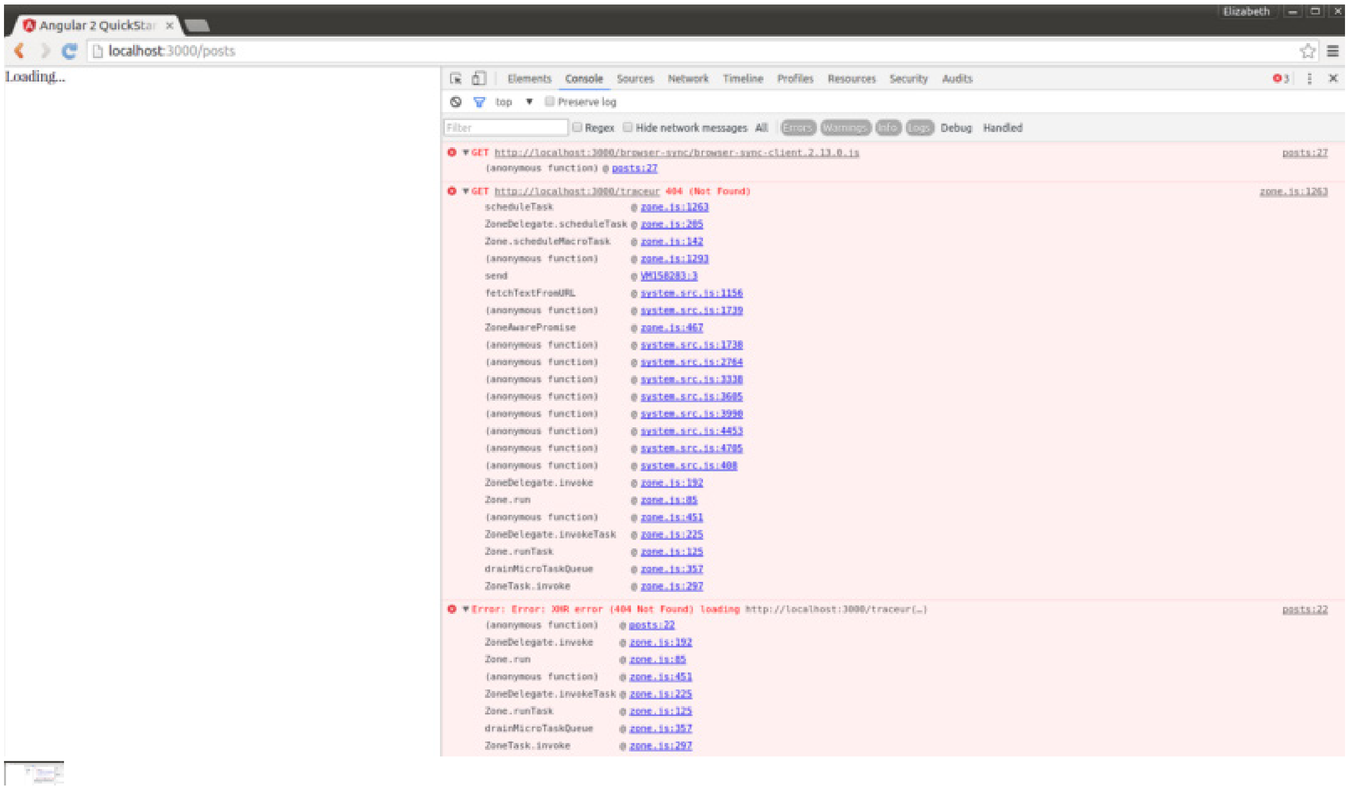
All I know is that the Angular 2 release was, from my perspective as a Java engineer, a freaking disaster. Every single minor release made lots of breaking changes. Every single time I checked back on the accursed "5 Minute Quick Start", huge swaths of it had completely changed. Don't even get me started on the whole Angular 2 router business. That was truly a fiasco, and agonizing to deal with as a user, particular one who is less familiar with the Javascript world.

I find myself writing this post in a moment of passion, fuming over the latest Angular release which was announced a few days ago. Angular 2, for rulz this time, has been released and you can upgrade to it and not have to deal with breaking changes for a whopping 6 months! Well then, I naively thought to myself, I should upgrade my blog from @angular 2.0.0-rc.4 to @angular 2.0.0.

This was the journey that I just underwent:

1. Upgrade to @angular 2.0.0
2. Remove all 'directives' fields from my Components. Apparently Angular2 has decided that Modules are the way to go.
3. Get rid of all imports anywhere that end with _DIRECTIVES.
4. Upgrade @angular/forms from 0.3.0 to 2.0.0. For some reason, @angular/forms was super behind the rest of the versioning for angular. Until this release.
5. Upgrade angular-material to 2.0.0-alpha.8–2 (can we just pause for a second and appreciate the ridiculousness of a version called 2.0.0-alpha.8–2??).
6. Upgrade to typescript 2.o, which, as I was unpleasantly surprised by, is currently in beta. Having finally reached a version of relative stability in Angular, it was dismaying to realize that angular-material, a key tool in my stack, has unstable dependencies that are ahead of Angular 2's dependencies.

By this point, `npm start` was working. This is where the hard part began, because now I had to deal with tremendously cryptic error messages that have been plaguing me ever since I started learning Angular 2. Such as this one:

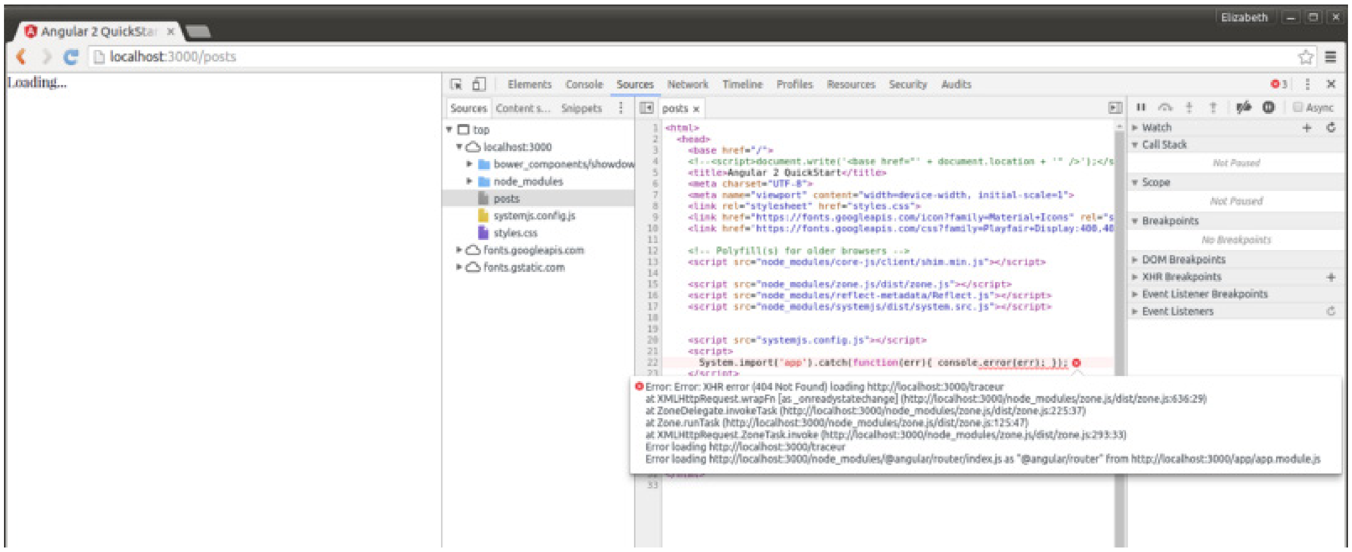


```
Loading...

GET http://localhost:3000/browser-sync/browser-sync-client.2.13.0.js          posts:27
    (anonymous function) @ posts:27

GET http://localhost:3000/traceur 404 (Not Found)                             zone.js:1263
    scheduleTask                 @ zone.js:1263
    ZoneDelegate.scheduleTask    @ zone.js:205
    Zone.scheduleMacroTask       @ zone.js:142
    (anonymous function)         @ zone.js:1293
    send                         @ VM158283:3
    fetchTextFromURL             @ system.src.js:1156
    (anonymous function)         @ system.src.js:1739
    ZoneAwarePromise             @ zone.js:467
    (anonymous function)         @ system.src.js:1738
    (anonymous function)         @ system.src.js:2764
    (anonymous function)         @ system.src.js:3338
    (anonymous function)         @ system.src.js:3605
    (anonymous function)         @ system.src.js:3990
    (anonymous function)         @ system.src.js:4453
    (anonymous function)         @ system.src.js:4705
    (anonymous function)         @ system.src.js:408
    ZoneDelegate.invoke          @ zone.js:192
    Zone.run                     @ zone.js:85
    (anonymous function)         @ zone.js:451
    ZoneDelegate.invokeTask      @ zone.js:225
    Zone.runTask                 @ zone.js:125
    drainMicroTaskQueue          @ zone.js:357
    ZoneTask.invoke              @ zone.js:297

Error: Error: XHR error (404 Not Found) loading http://localhost:3000/traceur(…)     posts:22
    (anonymous function)         @ posts:22
    ZoneDelegate.invoke          @ zone.js:192
    Zone.run                     @ zone.js:85
    (anonymous function)         @ zone.js:451
    ZoneDelegate.invokeTask      @ zone.js:225
    Zone.runTask                 @ zone.js:125
    drainMicroTaskQueue          @ zone.js:357
    ZoneTask.invoke              @ zone.js:297
```



```
<html>
  <head>
    <base href="/">
    <!--<script>document.write('<base href="' + document.location + '" />');</s
    <title>Angular 2 QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="s
    <link href="https://fonts.googleapis.com/css?family=Playfair+Display:400,40

    <!-- Polyfill(s) for older browsers -->
    <script src="node_modules/core-js/client/shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>


    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
```

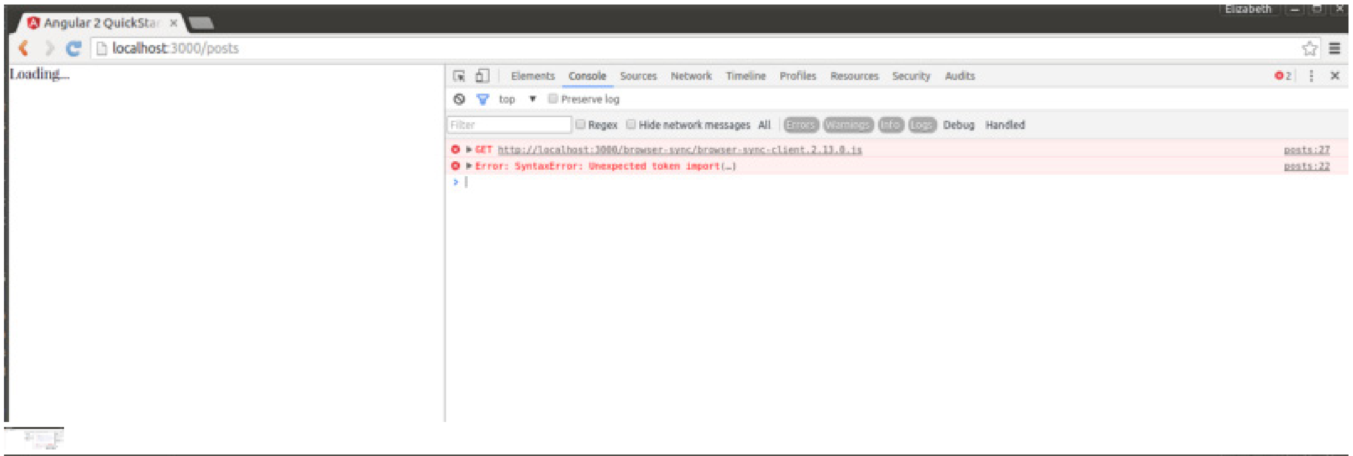
```
Error: Error: XHR error (404 Not Found) loading http://localhost:3000/traceur
at XMLHttpRequest.wrapFn [as _onreadystatechange] (http://localhost:3000/node_modules/zone.js/dist/zone.js:636:29)
at ZoneDelegate.invokeTask (http://localhost:3000/node_modules/zone.js/dist/zone.js:225:37)
at Zone.runTask (http://localhost:3000/node_modules/zone.js/dist/zone.js:125:47)
at XMLHttpRequest.ZoneTask.invoke (http://localhost:3000/node_modules/zone.js/dist/zone.js:293:33)
Error loading http://localhost:3000/traceur
Error loading http://localhost:3000/node_modules/@angular/router/index.js as "@angular/router" from http://localhost:3000/app/app.module.js
```

After some troubleshooting (by now I've gotten okay at debugging System JS's useless error messages), the issue was caused by this errant line in my systemjs.config.js file:

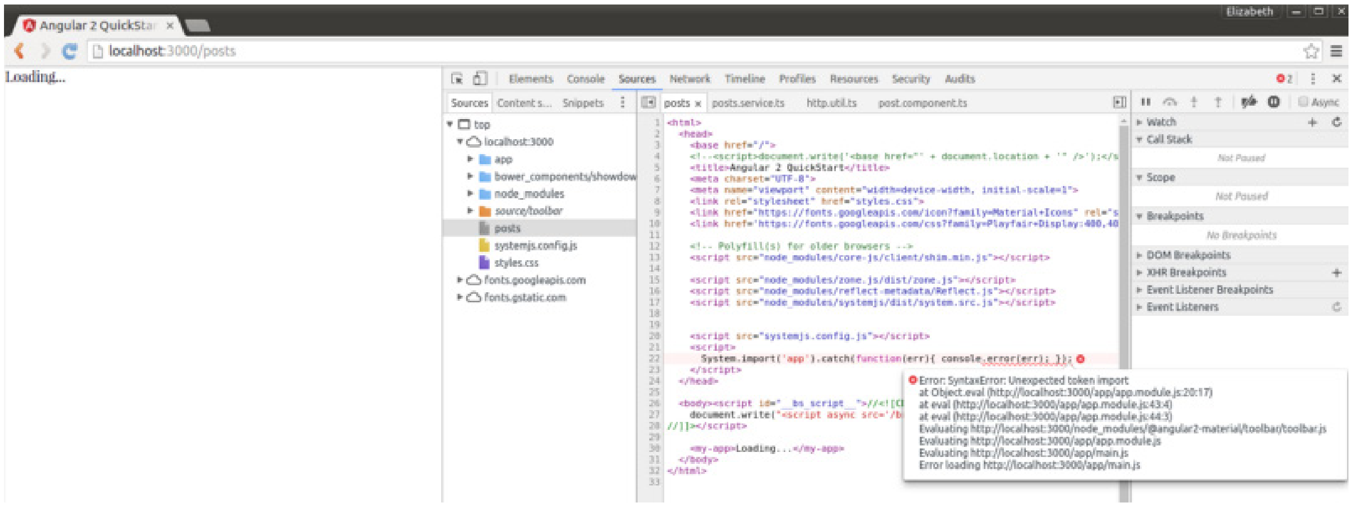
```
// No umd for router yet
packages['@angular/router'] = { main: 'index.js', defaultExtension: 'js' };
```

I guess @anguler/router has a umd now. Whatever a umd is………

The next issue I encountered after that was this:



```
Loading...
GET http://localhost:3000/browser-sync/browser-sync-client.2.13.0.js     posts:27
Error: SyntaxError: Unexpected token import(…)                          posts:22
```



```
<html>
  <head>
    <base href="/">
    <!--<script>document.write('<base href="' + document.location + '" />');</s
    <title>Angular 2 QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="s
    <link href='https://fonts.googleapis.com/css?family=Playfair+Display:400,40

    <!-- Polyfill(s) for older browsers -->
    <script src="node_modules/core-js/client/shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>


    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>

  <body><script id="__bs_script__">//<![C
    document.write("<script async src='/b
//]]></script>

    <my-app>Loading...</my-app>
  </body>
</html>
```

```
Error: SyntaxError: Unexpected token import
  at Object.eval (http://localhost:3000/app/app.module.js:20:17)
  at eval (http://localhost:3000/app/app.module.js:43:4)
  at eval (http://localhost:3000/app/app.module.js:44:3)
  Evaluating http://localhost:3000/node_modules/@angular2-material/toolbar/toolbar.js
  Evaluating http://localhost:3000/app/app.module.js
  Evaluating http://localhost:3000/app/main.js
  Error loading http://localhost:3000/app/main.js
```

Great, a freaking syntax error from a random file in angular-material. There is no helpful error message, no line numbers. There is precious little to help me figure out what to do next. I don't know whether to upgrade dependencies, downgrade dependencies, install new dependencies, change the syntax of my tsconfig.js file (which undoubtedly changed when I upgraded to typescript 2.0). I'm lost in a sea of confusion and frustration.

Now, those of you who are seasoned web developers can probably troubleshoot an issue like this fairly easily. That's not what I'm getting at. It's not about the particular issue I just faced. It's that the Javascript ecosystem is utter chaos. Every new version of every new library comes with a slew of breaking changes. New libraries will be released before their APIs are nailed down. Libraries in beta are now old news because only alpha libraries are new and interesting Stackoverflow posts that are older than 6 weeks old are no longer relevant and probably deal with an issue from an old version that I'm no longer using.

The Java engineer in me is screaming with the frustration of this ecosystem. What sort of madness has web development devolved into? What the f**cking f**ck is going on with Javascript these days???

Okay, I think it's time to take a step back and say that I actually love Angular 2. When I get to a point where all the dependencies have been upgraded, and everything is working, when Intellij is underlining the correct things and my typescript compiler is hooked up right, Angular 2 is awesome.

To be fair, I have been using versions of the library that have thus far not been officially released. Maybe, you say, it’s my fault for trying to download and use a version of the library that is still in alpha/beta/release candidate and expecting it to work and be relatively easy to use. Perhaps you are right. But, considering the fact that hundreds of thousands of developers are already using Angular 2, we should ask ourselves the question: is it responsible to release libraries that are still very much a work in progress? Does it make sense to announce a release candidate and then make tons and tons of breaking changes over the course of evolving from rc.1 to rc.6? How many hundreds of thousands of human hours were wasted dealing with the pain of upgrading a version of Angular 2 and all of its related dependencies? And, as most engineers will attest to, developer hours are a precious commodity in today’s job market. How many developers have been utterly burned by the experience of trying to use Angular 2 and have sworn off Angular forevermore in favor of React? How many other Javascript libraries are out there that have also caused such frustration and undue anguish for their users?

Perhaps the Javascript ecosystem is just experiencing *intense* growing pains. Maybe developers are quickly understanding the errors in their initial designs and brazenly correcting them as they blaze on in iterating through better and better versions of their libraries. Maybe the Javascript world is where the vast majority of software engineers will live soon, since the tools there will have evolved rapidly and effectively. Perhaps people will accomplish more in Javascript and the tools written in Javascript will be easier to use than the tools in any other software engineering landscape. Or, maybe, just maybe, the Javascript world will always be sort of a joke, a place where engineering hipsters go to waste time and feel like they’re on the cutting edge of innovation when really, they are living in a world of madness and chaos, throwing hours of productivity down the drain so that they can use the latest and ‘greatest’ tools out there.

But I am just a humble Java engineer. What I’m most interested in is: what do you think? ■

On DRY and the cost of wrongful abstractions

By VLAD ZELINSCHI

This article has been on my to do list for quite some time now. But it seems that only today I found the energy and time to take it from idea to implementation. Coincidence or not, I'm at the same coffee shop I posted my first article, a while ago. Must be that they put something in the drinks they serve...

So the good, old advice of following best practices, right? We hear about them all the time. We've somehow made acronyms such as *DRY* or *KISS* defaults in our technical conversations. We follow the concepts religiously and if, by chance, desire or lack of knowledge, someone strays away from them, we make sure we rain a shit storm of criticism upon them. We're caught in this dogma and we refuse to look away.

For sure I don't want to imply that principles such as DRY are bad. Definitely not. I just think context matters. A lot. In regards to DRY specifically, this brings in the following logical conclusion: I actually am the guy that will sometimes advise in favour of duplication rather than abstraction.

Yes, you read that right. Duplicating code (a.k.a copy + paste) can be a good thing. Namely when the abstraction that would replace repetitive portions of your codebase is a pain to understand.

How programming time is divided

When I tell people what I do for a living, they all imagine I must be some kind of weirdo who keeps slapping the keyboard ten hours a day or more.

Although, I'm not sure I'm not a little weird, I'm pretty sure I don't code for ten hours straight. The truth is that we, as programmers, spend far more time **reading code** than writing code. I'm not sure if you ever tracked this in one form or another, but research and Robert C. Martin claim to have some sort of ratio on this. And I, for one, find the difference to be staggering. It turns out that for every hour we code, we spend other ten hours reading code (ours or someone else's).

This is extremely important. The effort we put in a day of work goes mostly towards reading code. Of course, reading is not enough. We need to also understand. Which means we have to do our best to produce code that is clear, concise and easy to read. For everyone's benefit, including ours in the long term. Keep this in mind as we'll get back to this idea later on.

A note on DRY

For those who are not familiar with what DRY means, it basically stands for *don't repeat yourself*. This programming principle or best practice if you'd like, is advocating for creating abstractions on top of every repetitive portion of your codebase.

DRY has tons of advantages. For one, abstractions mean that, if changes will be required in the future, you will have to deal with them in one single place - the abstraction itself.

Consuming another module's functionality, someone else's API, etc. you're only concerned with how the interface (or abstraction) looks like. You don't care about the underlying implementation. Thus, software design patterns such as the **facade pattern** allow for easy refactoring of the implementation, without hindering the abstraction used by others.

So abstractions are good and DRY totally makes sense. Then why am I still insisting on repeating code in some scenarios?

Well, it's because...

The cost of abstractions

Every abstraction comes at a cost. A cost that may not be immediately visible. But it reveals itself in time.

Abstractions carry an extra layer of knowledge. Knowledge that you may not necessarily understand why it exists in the first place (especially if you didn't write it). And every new piece of information places cognitive load on your brain. Which, in turn, increases the time spent on reading that piece of code.

Down the rabbit hole

The problem with DRY and religious following of this principle is not visible in small projects. But in mid-sized and large ones.

In those kind of projects, it's very rare that you will write only one abstraction per duplication. This is because, as the project evolves and new requirements come in, old code must always adjust.

Imagine this. You enter a new project and scan the codebase for the first time. After getting used to what's there and learning your way around, you start implementing features, changing old ones, etc. You get to touch existing code and existing abstractions. You didn't write those, but they're there. And probably there's a very good reason for that.

As Sandi Metz said:

> *Existing code exerts a powerful influence. Its very presence argues that it is both correct and necessary.*

Which makes you not wanting to touch it.

Now the new feature that barely came in could definitely make use of that nice abstraction that you have there. But, as it turns out, the abstraction needs a bit of tweaking. It wasn't thought for this particular use case. If only you could change it a bit... Or maybe write a new abstraction on top of it that could encapsulate some other repetitive logic as well? Yeah, that seems to be the answer, right? That's what DRY says anyway...

It's easy to see how we can drive this madness to absolute extremes. Extremes where everything is encapsulated in abstractions. And those, in turn, have their own abstractions on top. And so on and so forth...

In these cases, the abstraction lost its value. Its existence was determined by dogmas we follow blindly. And this makes the abstraction a **wrongful one**. It exists just because we can.

As mentioned, abstractions have a cognitive cost associated with them. If anything and alongside the added benefits, that cost almost always hinders and increases the time we need to understand it (remember we're spending more time reading code than writing it). But even worse than a good abstraction, is a wrongful one.

> *Because wrongful abstractions not only kick you in the nuts, they also laugh while doing it.*

To DRY or not to DRY

So when do you encapsulate repetitive code and when do you not?

The answer is simple in itself. It's just hard to get it right from a practical point of view. But this does come with experience as well.

> *Always abstract if the cost of abstraction does not surpass the cost of duplicating code.*

That is, if the abstraction you wrote requires someone new on the project to spend hours understanding it, then you're probably doing something wrong. Don't abstract just because you can. Predict whether code duplication will happen again on that particular portion or not and decide accordingly.

Sometimes, repeating code might hinder much less than following a tree of nested calls to different methods and keeping track of passed parameters, possible side effects, etc.

Closing thought — in defence of myself

Hopefully this article does not scream "To hell with DRY and other shit!". I absolutely think that is a very good programming principle. But I also urge you to not follow it blindly. Put everything you learned in context and always question the validity of your ideas and actions. This is the only sane way towards becoming a better professional.

Looking forward to your comments. Remember to smile often and always question best practices. ■

Not just any old geek

By JOHN WHEELER

It's not easy getting a job at a Silicon Valley startup of twenty-somethings when you're a 37 year old programmer like me. I have a long, unglamorous work history mostly in government. That my last job was managing other programmers at Xerox, inspires as much technical credibility as… well… Xerox.

But I'm not just any old geek. In my spare time, I've published a dozen articles for IBM and O'Reilly. I taught myself App Engine and used it to build software with thousands of paying customers. I study and work on open source Python. I've been using Angular 2 since beta 6 and Bootstrap 4 since alpha 1. I've also read almost 40 business books, and I live and breathe Hacker News.

I'm only getting better as I grow older, but my perceived market value is worsening. I know first-hand. Recently, two early twenty year olds at a startup I interviewed with told me I wasn't a cultural fit—and I really wasn't. Even though their job advertisement called for the passionate, opinionated, and entrepreneurial-type, they really wanted someone who could *become* that under *them*. Both were solid, but according to LinkedIn, neither had much experience outside college. At any rate, I can appreciate their mindset even if I don't agree with it.

But, when Tim Bray paraphrased James Gosling about employers making age-based concessions with him, I saw a harsh glimpse into my future.

Basically, it's all downhill.

Gosling has a PhD from CMU, wrote versions of Emacs and Unix, and invented the Java programming language. You don't get to that level working in a vacuum, and actually, you probably never to get to that level period. But, if you care about what you do and ascend to your own new heights, the ranks of those who can influence you diminishes. The flip-side is the immense value you place on the ones who can because of their increasing scarcity—your productivity compounds when you know your stuff and work with others who do too.

That's why after I read Bray's post last week, I immediately created OldGeekJobs.com. It's not a place for COBOL programmers to go and die; it's for us lifelong autodidacts to find jobs using the technology we love. If you're an employer, post your jobs free. I'm pulling in StackOverflow jobs, highlighting the jobs posted on my site green, and improving the UX, all in attempts to overcome the chicken-and-egg problem inherent in two-sided marketplaces. I'm also writing about my journey, so follow me on Twitter for updates. ■

"Snowball" by Kamyar Adl is licensed under CC BY 2.0

Don't start big, start a little snowball

By JUSTIN VINCENT

Little Snowballs

When myself and my co-host interviewed Travis Kalanick on our podcast, he had recently co-founded a little snowball called UberCab. It was so early in Uber's existence he didn't even mention it.

I notice Uber falls into a category of companies I call little snowballs. There are some fundamental features these companies have in common. I thoguht it might be helpful to list a few little snowballs and then talk about how you can go about starting your own.

Uber

Travis Kalanick and Garrett Camp commissioned the creation of a simple app that enabled users to hail a black car on demand. To validate they asked friends and colleagues to install the app. They hired a driver & watched what happened. After a few months of hustle the app hit 10 rides in one day.

Airbnb

Brian Chesky and Joe Gebbia bought a few airbeds and put up a static site called "Air Bed and Breakfast". They expanded the concept so that other people could offer airbeds in their homes and make money too. Then they expanded the idea to rooms, then to entire apartments. Their journey was a hustle from hell, but they made it happen.

Google

Sergey Brin and Larry Page built a web search engine. It was one page with a search box and submit button. The backend was a simple database search returning ranked results based on the number of backlinks. It did a really good job at providing search results, and took off.

Slack

Stewart Butterfield worked with his Tiny Speck group to build a better team messaging app. Slack makes it really easy for people to signup and start chatting as a team. At it's core Slack is a simple IM client.

Other examples

AT&T, Dollar Shave Club, Buffer, Shazam, DropBox

What is a Little Snowball?

"A startup that is founded to do-one-simple-thing-well, and that can be massively grown outward from that starting point."

Typical markers:

- Very simple idea with a single facet
- Extremely low product surface area (frontend and backend)
- Can generally be prototyped and brought to market in months
- Is instantly useful and quickly gains early-adopter traction

Each of these markers makes it considerably easier for you as a founder to start and grow the first version of your business.

A Simple Single Facet

Just about every part of growing your business becomes easier if the foundational idea is do-one-thing-well. I've been part of a number of startups where it was almost impossible to explain what we do in a single sentence. This wreaks havok during sales meetings and investor pitches.

However, when your startup is a single cell organism, customers and investors are not confused about what you do. Word of mouth and viral loop growth becomes significantly easier to achieve. You can focus on growing the business, rather than trying to understand what the business is.

A Low Product Surface Area

This is a huge, almost unfair, advantage. Dollar Shave Club is the ultimate example. An idea you can build and launch in a weekend. Then, from Monday forward, spend all your time, effort and money on marketing and learning about your customers.

I learnt this lesson the hard way. I lunched my tool Pluggio, a tool that buffers social content, at the same time as Buffer. Buffer focused on doing one-thing-well while I built Pluggio out to become a social media dashboard. Ironically, no matter how many features I added, Pluggio's main selling point continued to be social buffering.

So, while I was adding 100k lines of code to make a really slick single page web app, buffer was marketing and integrating their very simple app. Due to my high product surface area my time was sucked up by support requests. Sales were harder because the product was harder to explain. Integrations were a non starter because the product was to complex.

After 3 years Pluggio's yearly revenue was $50k and Buffer's was $1m.

Fast to Prototype and Launch

The sooner you can get to market the sooner you can validate and start learning about how your customers make purchase or adoption decisions.

Often times you can quickly validate a business by taking very manual steps that don't scale. For example, if you wanted to validate the business hypothesis – *People will order coffee from a delivery app* – you could get 250-500 business cards printed out saying – *We deliver coffee. Call us on 123-123-1234.* – then hand them out at the mall at breakfast time.

The very next day you will have some great information about your basic hypothesis.

Build Something People Want

The final little snowball component is to make sure your core idea is something people *actually* want. Without that je ne sais quoi your rolling stone will gather no moss.

How Can I Find My Own Little Snowball?

Full disclosure, I am founder of Nugget (startup incubator & community). Our primary purpose is to help you find, and grow, your own little snowball.

That said, here are some tips to help you come up with off the cuff ideas:

Think Small

This sounds easier than it is. Many founders think at large scale because they want to build a Google or Uber. They feel it's too much of a gamble to think about smaller things. Another common trap is to get caught in the mindset of thinking more features equals more sales.

To think small, you should notice details of everyday problems going on around you and try to find do-one-thing-well opportunities.

Notice Problems

Problems are everywhere. The trick is to open your consciousness to the stream of problems that life presents us with. You would be surprised at how capable we are of drowning out regularly annoying tasks that become part of our daily life. Now it's time to start to notice those things we've trained ourself to ignore.

One note about this, it's very important you solve a problem that other people have. That sounds like obvious advice, but if no one else has the problem, you're dead in the water.

Look For Optimizations

Optimization essentially means – to make something better – and is really just another way to solve a problem. However, thinking through the mental model of optimization, can lead you to come up with some interesting product ideas.

For example the core essence of Buffer is an optimization of Twitter.

As a side note. Question: Why didn't Twitter simply add that feature when they saw it take off in Buffer, Hootsuite and other apps? Answer: They were too busy staying on task and building their own little snowball.

Explore Old Ideas

You probably already have a number of ideas you've been thinking about and possibly even products you've launched. Have another look at them through the lens of what we've discussed here. With any luck you might find a single facet that can be extracted from that old work and turned into a do-one-thing-well little snowball. ■

I'm building Nugget (startup incubator & community). We send a new SaaS startup idea to your inbox every single day. Signup for free.

"Typing" by Sebastien Wiertz is licensed under CC BY 2.0

Lessons learned from software rewrites

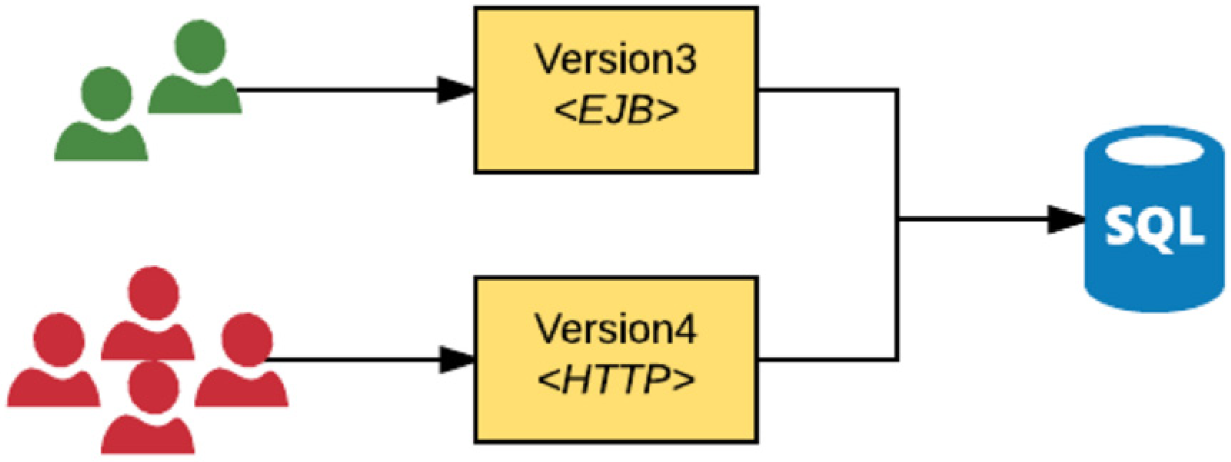
By ALEX MARTINS

A couple of years ago I joined a team working on one of Globo's core products, the platform responsible for user identification and authorization within all our products. With the increasing necessity to know more about our users, to be able to offer them a better experience on our products, my main challenge was to help this team improve the current software architecture, which was old and unreliable, impacting the cost of developing new features.

The primary service, called "Version3", with all functionalities was implemented back in 2004, using Java and the EJB stack. All services talking to it were restricted to use Java, as the only integration point were the EJB clients. As time passed and development teams started creating systems using technologies other than Java, the company felt they needed a better solution, one that was agnostic to any programming language, so they decided to build "Version4", a feature compatible RESTful implementation of "Version3."

Pretty much all features were rewritten on the new stack, some of them didn't even make sense to the business anymore, but due to the lack of service usage metrics, they were also added to the scope, a massive waste of development effort.

With the release of the new HTTP implementation, most of the clients started using it right away, but a few other ones ended up continuing using the EJB implementation, as the cost of migrating all integrations points from EJB to HTTP was quite high. As a result, we ended up with two implementations of the same set of functionalities.



This approach worked pretty well for some time until requests to new features started to pop out. The team had to implement them on both versions for compatibility, and those who have worked with EJB knows how hard and costly it is to test implementations, especially running acceptance tests when you need to spin up the container which is quite slow. That was affecting considerably the team's ability to extend the system.

Problems with duplicate systems

The new service implementation brought many benefits to the organization, as HTTP is a language agnostic protocol, development teams could explore even more technologies and use the right ones for their problems at hand. But when I joined the team, with this scenario at hand, I started to realize that keeping these two implementations alive was introducing too many bottlenecks in the development process, as described below.

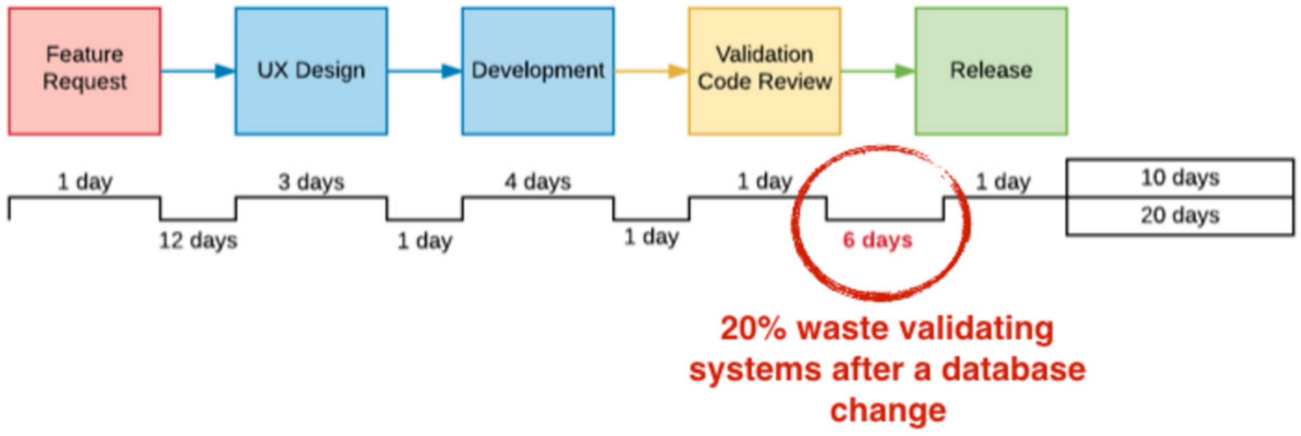
Duplicate effort vs. partial feature availability

For every new feature request to the platform, the development team had to check with the users of "Version3" if they would also need that new feature, and based on that decide if they were implementing it on both systems or not. Bug fixes were a waste as developers had to fix and test them on both systems. We needed to find of getting rid of this duplicate effort.

Both systems sharing the same database

With both systems sharing that same database, it became more expensive to maintain them, as for every change on any of the systems that triggered a change in the database, that would potentially break the other one, causing a ripple effect through every application using them. As a consequence, the development process became more bureaucratic and less responsive to business demands, as developers needed to double check that both systems were working as expected.

For one of the features, I decided to write down all the times spent on every development stage and create a value stream map to understand how this delay was impacting our cycle time. I found out that this extra effort in validating systems after a database change took **20% of the whole development process**!



To make things even more exciting, after adding some performance tests to the HTTP implementation, we found out that it had serious performance issues, mostly because it was running on top of a blocking server architecture. Also, some of the core features were tightly coupled with some outdated services that were slowing down significantly response times.

Our plan to improve this architecture

With that scenario in hand, it was clear that we needed to simplify the overall design to start gaining development speed. Our goal was to have a single implementation, with a straightforward and robust architecture, but we knew we couldn't have it straight away, so we decided to use an approach inspired on Toyota's improvement katas, to step-by-step head towards that direction.

Understanding how people are using our services

Before getting our hands dirty we wanted to understand the scope of the work, so we started collecting system usage metrics to find out how our clients were using our services, and from that discover the features that were being used and the others no one cared. The result was going to be used as input for our subsequent efforts, both to eliminate dead features and to prioritize the ones we were going to start implementing the new architecture. We used Codahale Metrics for collecting these metrics, but there are plenty of other alternatives out there. After a few weeks, we've already had enough data to start electing the first features to be removed.

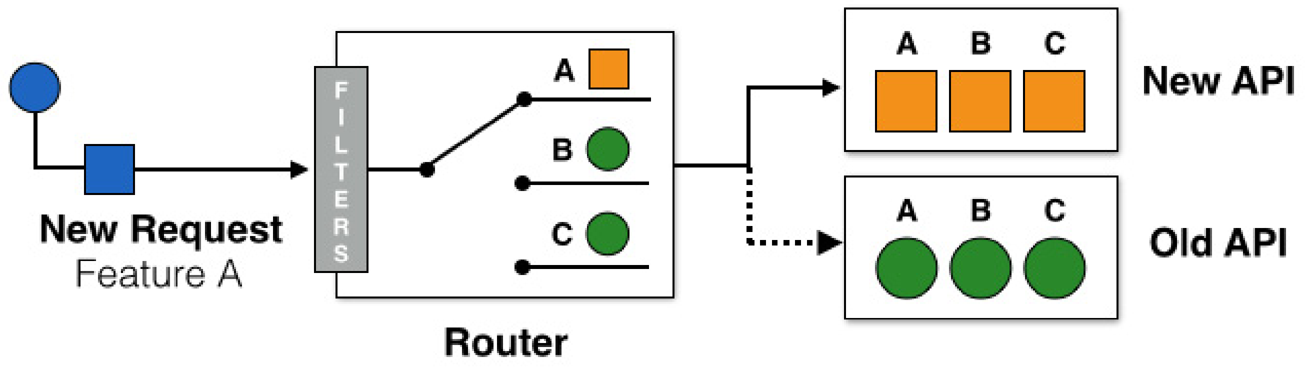
Reducing workload by eliminating "dead" features

While working on this project, I met my friend Joshua Kerievsky in one of his visits at Globo.com. I was sharing with him some of our learnings and challenges we were facing, then he mentioned a refactoring strategy he teaches in one of his workshops, that could be helpful for us to get where we wanted to be. The goal is to remove dead code from your codebase as they introduce unnecessary complexity, in particular on the parts that are integrating with them. I remember once pairing with my colleague-Andre; we were tracing down the process of creating a new user, which was slow at the time. To our surprise we found out that's because it was querying the database 31 times within a single execution, mostly doing work related to dead features. Once potential candidates were identified, they were put into a code quarantine for a few weeks to make sure no one was using them, and then deleted completely. We spent a few months doing that and **ended up reducing the number of features by ~40%**, which was a good return on investment.

Creating a Strangler Application to slowly get rid of the old systems

The metrics we'd collected were also useful to rank the most important features and from that select the ones we would start tackling. For every piece of functionality, we created a suite of characterization tests where we verified exactly how it was supposed to behave, and we also took advantage of that to start using a technique called continuous integration, that allowed us to automatically check, on every build, that it was still working as expected. That increased the level of confidence of the team to implement the functionality on the new stack, which was based on Scala and Finagle.

We used a technique called Strangler Application to rewrite every feature into the new stack. As Martin Fowler describes in his post, *it works by gradually creating a new system around the edges of the old one, letting it grow slowly over several years until the old system is strangled*. The reason why we decided to go with that approach was that the whole process needed to be extremely transparent to the users of our services. To make that happen we've implemented a HTTP Content-Based Router that intercepted every request to our services, and where we could tell which implementation it was going to use to handle specific endpoints. So once a functionality was implemented on the new stack, with all characterization tests passing, all we needed to do was to tell the router to use this new implementation, instead of the old one, and everyone consuming this endpoint would be automatically using the new implementation! This strategy would also allow us to rollback to the old version if something wrong happened.



Takeaways

Software rewrite is a quite painful process, no matter which strategy you choose to use. On another experience, we decided to rebuild the whole thing in parallel, while the old system was in production, and I remember we dedicating a lot of effort trying to catch up with the old system, as it continued growing. The Strangle Application strategy is the one that worked for me, although it's a long process to get there. I left the project a year ago and last time I heard from them, they were still working on that. It all will depend on your scenario. ■

Understanding /proc

By FREDERICO BITTENCOURT

Last week I created a small `ps` clone in ruby. This was done purely out of curiosity, just wondering howdoes `ps` works and how it knows all about current running processes. You can find the project here.

Cool things I learned in the process:

- `procfs` is a virtual filesystem that stores process data!
- `ps` clone is merely file reading (really).

Exploring procfs

At first, I went around the web and read about `ps` and linux processes. This happened to be the first time I was introduced to procfs reading at TLDP (http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html) (awesome documentation btw).

In summary, all linux's processes can be found in `/proc` folder. This folder is of *procfs* type which, like I said before, is a virtual filesystem and most of its file descriptors point to in-memory data. This is why if you run a `ls /proc -l` you'll notice that most files and folders are of size 0.

```
ls -l /proc
dr-xr-xr-x.  9 root          root                         0 Sep 25 22:10 1
dr-xr-xr-x.  9 root          root                         0 Oct  1 10:38 10
dr-xr-xr-x.  9 root          root                         0 Oct  1 12:46 101
dr-xr-xr-x.  9 root          root                         0 Oct  1 12:46 102
...
```

Inside '/proc' there is one folder for **each** process running with its pid as name. So I opened one of the folders to see what I could learn about a running process just by reading these filed.

```
ls -l /proc/<pid>
total 0
dr-xr-xr-x. 2 fredrb fredrb 0 Sep 28 23:15 attr
-rw-r--r--. 1 root   root   0 Oct  1 10:46 autogroup
-r--------. 1 root   root   0 Oct  1 10:46 auxv
-r--r--r--. 1 root   root   0 Sep 28 23:15 cgroup
--w-------. 1 root   root   0 Oct  1 10:46 clear_refs
-r--r--r--. 1 root   root   0 Sep 28 22:41 cmdline
-rw-r--r--. 1 root   root   0 Oct  1 10:46 comm
-rw-r--r--. 1 root   root   0 Oct  1 10:46 coredump_filter
...
```

Ok, now I have a bunch of files like `autogroup`, `gid_map` and `maps` that I have no idea what they're for. A good starting point would be checking for their documentation. But why on earth shouldn't I just open them?

So I started looping through the files one by one and most of them were completely unreadable for me, until I ran into the golden pot:

```
cat /proc/<pid>/status
Name:  chrome
State: S (sleeping)
Tgid:  3054
Ngid:  0
Pid:   3054
PPid:  2934
TracerPid:     0
Uid:   1000    1000    1000    1000
```

```
Gid:   1000    1000    1000    1000
FDSize:        64
Groups:        10 1000 1001
VmPeak:         1305996 kB
VmSize:         1232520 kB
...
```

This is great! Finally something human readable. It contains general data about the process, like its state, memory usage and owner. But is this all I need?

Not satisfied with '/proc' file exploration, I decided to run `ps` against `strace` to see if it's accessing any of the files I found.

```
strace -o ./strace_log ps aux
```

Strace returns all system calls executed by a program. So I filter strace result by 'open' system call and as I suspected (maybe I didn't) the files being open by operating system were the same I first checked:

```
cat ./strace_log | grep open
[...]
open("/proc/1/stat", O_RDONLY) = 6
open("/proc/1/status", O_RDONLY) = 6
[...]
open("/proc/2/stat", O_RDONLY) = 6
open("/proc/2/status", O_RDONLY) = 6
open("/proc/2/cmdline", O_RDONLY) = 6
open("/proc/3/stat", O_RDONLY) = 6
open("/proc/3/status", O_RDONLY) = 6
open("/proc/3/cmdline", O_RDONLY) = 6
[...]
```

Ok, so we have `stat`, `status` and `cmdline` files to check, now all we need to do is to parse this and extract what we need.

The code

The implementation turned out to be fairly simple and it comes down to reading files and display its content in an organized matter.

Process data structure

We want to display our data in a tabular way; where each process is a record on this table. Let's take the following class as one of our table records:

```
class ProcessData
  attr_reader :pid
  attr_reader :name
  attr_reader :user
  attr_reader :state
  attr_reader :rss

  def initialize pid, name, user, state, rss
    @pid = pid
    @name = parse_name name
    @user = user
    @state = state
    @rss = rss
  end
end
```

Finding Pid's for running processes

Take into account what we know so far:

- `/proc` folder contains sub-folders with all processes
- All process folders have their pid as name

So gathering a list of all current pids should be easy:

```
def get_current_pids
  pids = []
  Dir.foreach("/proc") { |d|
    if is_process_folder?(d)
      pids.push(d)
    end
  }
  return pids
end
```

In order to be a valid process folder it must fulfill two requirements:

- It's a folder (duh?)
- It's name contains only number (this is why we have to cast folder name to int)

```
def is_process_folder? folder
  File.directory?("/proc/#{folder}") and (folder.to_i != 0)
end
```

Extracting process data

Now that we know every pid in the system we should create a method that exposes data from `/proc/<pid>/status` for any of them.

But first, let's analyze the file.

```
cat /proc/<pid>/status
Name:  chrome
State: S (sleeping)
...
Uid:   1000    1000    1000    1000
```

This file is organized in the following way: `Key:\t[values]`. This means that for **every** piece of data in this file we can follow this same pattern to extract it. However, some lines will have an individual value and others will have a list of values (like `Uid`)

```
def get_process_data pid
  proc_data = {}
  File.open("/proc/#{pid}/status") { |file|
    begin
      while line = file.readline
        data = line.strip.split("\t")
        key = data.delete_at(0).downcase
        proc_data[key] = data
      end
      file.close
    rescue EOFError
      file.close
    end
  }
  return proc_data
end
```

The method above results in the following structure:

```
get_process_data 2917
=> {"name:"=>["chrome"],
 "state:"=>["S (sleeping)"],
 "tgid:"=>["2917"],
 "ngid:"=>["0"],
 "pid:"=>["2917"],
 "ppid:"=>["1"],
 "tracerpid:"=>["0"],
 "uid:"=>["1000", "1000", "1000", "1000"],
 ...
```

Reading user data

User `uid` and name association is kept in `/etc/passwd` file, so in order to show the correct username we must also read this file and parse it.

For the sake of simplicity, let's just read the whole file and save it in a `Hash` with key as `Uid` and value as name.

```
def get_users
  users = {}
  File.open("/etc/passwd", "r") { |file|
    begin
      while line = file.readline
        data = line.strip.split(":")
        users[data[2]] = data[0]
      end
      file.close
    rescue EOFError
      file.close
    end
  }
  return users
end
```

Creating process records

So far we have found the `pids` in the system, read the `status` file and extracted the data. What we have to do now is to filter and organize this data into a *single record* that will be presented to the user.

```
current_processes = filesystem.get_current_pids

current_processes.each { |p|
  process = create_process p
  puts "#{process.name}\t#{process.user}\t#{process.state}\t#{process.command}\t#{process.rss}"
}

def create_process pid
  data = get_process_data pid

  name = data["name:"][0]
  user_id = data["uid:"][0]
  state = data["state:"][0]

  if data["vmrss:"] != nil
```

```
    rss = data["vmrss:"][0]
  end

  user = get_users[user_id]

  return ProcessData.new(pid, name, user, state, rss)
end
```

The reason why we get `VMRss` value is because we want to check resident memory values, this means, only what's stored in the physical memory and not what's sitting in our disk.

Extra (formatting)

You can format ProcessData text in a tabular way to get a prettier output.

```
format="%6s\t%-15s\t%-10s\t%-10s\t%-10s\n"
printf(format, "PID", "NAME", "USER", "STATE", "MEMORY")
printf(format, "------", "---------------", "----------", "----------", "----------")
current_processes.each { |p|
  process = create_process p
  printf(format, process.pid, process.name, process.user, process.state, process.rss)
}
```

Result:

```
PID    NAME                   USER          STATE          MEMORY
------ ---------------        ----------    ----------     ----------
   1   systemd                root          S (sleeping)       8444 kB
   2   kthreadd               root          S (sleeping)
   3   ksoftirqd/0            root          S (sleeping)
   ...
```

Conclusion

There is a lot of information that you can find under `/proc` folder. This post only covers basic data like name, state and resident memory. But if you dig deep into those files you will find a lot more, like memory mapping and CPU usage.

It was very interesting exploring this part of Linux and hopefully you learned something new with this. ■

Reprinted with permission of the original author. First appeared at fredrb.github.io.

Bezoar orientalis, Orientalischer Bezoar.
Bezoarsteine galten als sehr wertvoll, oft wurden sie
in Gold gefasst. Der Name Bezoar bedeutet auf persisch
Gegengift. Der orientalische Bezoar stammt aus dem
Magen der Bezoarziege, Capra aegagrus. Pulverisiert
wurde der Stein gegen Vergiftungen eingenommen, doch

food bit *

Bezoar

A bezoar is a mass of foreign matter that is trapped in the gastrointestinal system of humans and animals. These lumps of food, hair and indigestible materials become compacted into hard stones after some time, requiring surgical removal.

Despite its revolting origins, bezoars were highly prized in the ancient world as antidote, and a bezoar stone is said to have the power to neutralize any poison.

Interestingly, eating unripe persimmons is a common cause of bezoars, as the fruit contains a substance that forms a sticky mass when it comes in contact with a weak acid.

On the bright side, doctors have discovered that Coca-Cola can shrink or remove these pesky persimmon-bezoars, eliminating the need for surgical removal.

* FOOD BIT is where we, enthusiasts of all edibles, sneak in a fun fact about food.