hacker bits

May 2016

new bits

Hello from Redmond!

Life as a tech professional ain't easy... we know because y'all wrote to tell us! Among the top issues were keeping up with the onslaught of new technology and maintaining work-life balance. But have no fear, because we are here to help! :)

In this issue of *Hacker Bits*, we chat with John Sonmez on how tech professionals can become better at their jobs and lead rich, fulfilled lives.

Hint: The trick is learning to learn.

If you've gone on the job interview loop lately, chances are you've caught a glimpse of the industry's fickle hiring practices. Dan Luu's well-researched piece reveals how some companies are hiring irrationally by focusing only on particular candidates.

And as always, we have a line-up of all the essential tech articles you should be reading, including the lowdown on writing and organizing good code, and what you can expect from state of the art JavaScript in 2016.

So dig in and enjoy another meaty issue of *Hacker Bits*!

Happy learning!

— Maureen and Ray

us@hackerbits.com

content bits

May 2016

6 State of the art JavaScript in 2016

14 Trackers

16 Kill your dependencies

18 "We only hire the best" means we only hire the trendiest

26 Writing good code: How to reduce the cognitive load of your code

30 My biggest regret as a programmer

32 Interview: How to become a better programmer

36 Four strategies for organizing code

43 Why I switched to Android after 7 years of iOS

48 On asking job candidates to code

52 I tried to virtually stalk Mark Zuckerberg

contributor bits

Francois Ward
Francois is a software engineer and opinionated JavaScript enthusiast. He specializes in front end tooling, infrastructure and is obsessed with all things related to unit testing. Currently, he's on Hubspot on the front end as a service team in Cambridge, MA.

Jacques Mattheij
Born in 1965, just in time for a front-row-seat for the PC revolution, playing with technology since I was old enough to hold a screwdriver, mechanical stuff, electrical, electronics and finally software. Inventor of live-streaming-video on the web, currently mostly active as interim CEO/CTO and doing technical due-diligence for top-flight VCs.

Mike Perham
Mike is CEO of Contributed Systems, purveyors of open source-based software. Their main product is Sidekiq Enterprise, the finest background job framework money can buy. It's artisanally-crafted in Portland, OR.

Dan Luu
Dan has worked on deep learning, CPU verification/test/ucode/RTL/etc, and other performance critical problems for Microsoft, Google, and Centaur. Check out his blog at danluu.com.

Christian Mackeprang
Christian is an independent web developer and worked for two decades in some of the largest websites in Argentina. Now he's focused on giving something back to the community through his blog and writing about software craftsmanship.

Martin Sandin
Martin is a software engineer who likes programming languages, concurrency, distributed systems, and organizing stuff. He has been programming since he was a kid and occasionally writes about it to make sure that at least he understands his own idiosyncrasies.

Henrik Joreteg
Henrik is a progressive Web App developer, consultant, author, and educator. He believes the Web is the future of mobile and IoT.

Alex Kras
Alex is a Software Engineer by day and Online Marketer by night. You can find his blog and learn more about him at alexkras.com.

Phil Calçado

Phil is the Director of Software Engineering at DigitalOcean. Previously, he was the Director of Core Engineering at SoundCloud, and his team was responsible for "keeping the trains running" in our microservices architecture. Before that he was the Director of Product Engineering at SoundCloud, and before joining SoundCloud he was a Lead Consultant for ThoughtWorks in Australia and the UK.

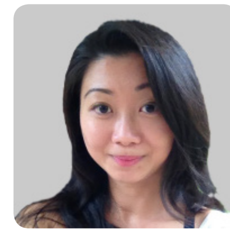
Andrew Wulf

Andrew, "The Codist," is still delivering after 35 years as a programmer. He is currently working in Florida after spending most of his life in Texas.

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.

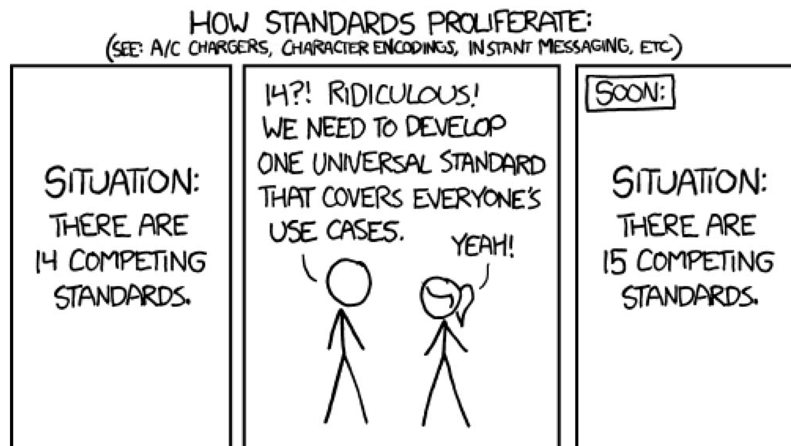
Programming

State of the art JavaScript in 2016

By FRANCOIS WARD

So, you're starting a brand new JavaScript front-end project or overhauling an old one, and maybe you haven't kept up with the breakneck pace of the ecosystem.

Or you did, but there are too many things to choose from. React, Flux, Angular, Aurelia, Mocha, Jasmine, Babel, TypeScript, Flow, oh my! By trying to make things simpler, some fall into a trap captured by one of my favorite XKCD comics.



Credit: www.xkcd.com

Well, the good news is the ecosystem is starting to slow down. Projects are merging. Best practices are starting to become clear. People are building on top of existing stuff instead of building new frameworks.

As a starting point, here's my personal picks for most pieces of a modern web application. Some choices are likely controversial and I will only give basic reasoning behind each choices. Keep in mind they're mostly my opinion based on what I'm seeing in the community and personal experiences. Your mileage may vary.

Core library: React

The clear winner right now is *React*.

- Components all the way down makes your application much easier to reason about.
- The learning curve is very flat. The important APIs would fit on one page.
- JSX is awesome. You get all the power of JavaScript and its tooling when writing your markup.
- It is the natural match for Flux and Redux (more on that later).The React community is amazing, and produced many best-of-breed tools such as Redux (also more on that later). Writing high quality data flow is much easier in large applications than dealing with 2-way data binding (eg: Knockout).
- If you ever need to do server-side rendering, React is where it's at.

There's plenty of monolithic frameworks like Ember, Aurelia and Angular that promise to take care of everything, but the React ecosystem, while requiring a few more decisions (that's why you're reading this!), is much more robust. Many of these frameworks, such as Angular 2.0, are playing catch-up with React.

Picking React isn't a technology decision, it's a business decision.

Bonus points: once you start working on your mobile apps, you'll be ready for it thanks to React Native.

Application life cycle: Redux

Now that we have our view and component layer, we need something to manage state and the lifecycle of our application. Redux is also a clear winner here.

The clear winner right now is React.

Alongside React, Facebook presented a design pattern for one-way data flow called Flux. Flux largely delivered on its promise of simplifying state management, but it also brought with it more questions, such as how to store that state and where to do Ajax requests.

To answer those questions, countless frameworks were built on top of the Flux pattern: Fluxible, Reflux, Alt, Flummox, Lux, Nuclear, Fluxxor and many, many more.

One Flux-like implementation eventually caught the community's attention, and for good reasons: Redux.

In Redux, almost all of the moving parts are pure functions. There is one centralized store and source of truth. Reducer functions are responsible for manipulating data that makes up the store. Everything is much clearer than in vanilla Flux.

More importantly, learning Redux is a snap. Redux's author, Dan Abramov is a fantastic teacher, and his training videos are fantastic. Watch the videos and, become a Redux expert. I've seen a team of engineers go from nearly zero React experience to having a production-ready application with top notch code in a few weeks.

Redux's ecosystem is as top notch as Redux itself. From the nearly magical devtool to the amazing memoization utility reselect, the Redux community got your back.

One thing to be careful is the natural instinct to try and abstract away the Redux boilerplate. There are good reasons behind all those pieces. Make sure you tried it and understand the "why" before trying to blindly improve on it.

Language: ES6 with Babel. No types (yet)

Avoid CoffeeScript. Most of its better features are now in ES6, a standard. Tooling (such as CoffeeLint) is very weak. Its community is also rapidly declining.

ES6 is a standard. Most of it is supported in the latest version of major browsers. Babel is an amazing "pluggable" ES6 compiler. Configure it with the right presets for your target browsers and you're good to go.

What about types? TypeScript and Flow both offer ways to add static typing to JavaScript, enhance tooling and catch bugs without needing tests. With that said, I suggest a wait and see approach for now.

TypeScript tries too hard to make JavaScript like C# or Java, and lacks modern system features such as algebraic data types (and you really want those if you're going to do static types!). It also doesn't handle nulls as well as Flow.

EDIT: It was brought up that TypeScript does have things like union types which cover a lot of use cases. I only meant that I think when it comes to type system in UX-land, you go all the way or not at all.

Flow can be much more powerful, catching a wider variety of bugs, but it can be hard to set up. It's also behind Babel in terms of language features and has poor Windows support.

I'll say something controversial: types are not nearly as critical to front-end development as some will have you believe (the whole argument will have to be in a future blog post). Wait until the type systems are more robust and stick to Babel for now, keeping an eye on Flow as it matures.

Types are not nearly as critical to front-end development as some will have you believe.

Linting & style: ESLint with Airbnb

Another clear winner is ESLint. With its React plugin and awesome ES6 support, one could not ask for more of a linter. JSLint is dated. ESlint does what the JSHint + JSCS combo does in a single tool.

Another clear winner is ESLint.

You do have to configure it with your style preferences. I highly recommend Airbnb's styleguide, most of which can be enforced via the ESlint airbnb config. If your team is the kind that will argue on code style, just use this style guide as gospel and the end-all and be-all of arguments. It isn't perfect, but the value of having consistent code is highly underestimated.

Once you're comfortable with it, I'd suggest enabling even more rules. The more that can be caught while typing (with an ESlint plugin for your favorite editor), the less decision fatigue you'll have and the more productive you and your team will be.

Dependency management: It's all about NPM, CommonJS and ES6 modules

This one is easy. Use NPM. For everything. Forget about Bower. Build tools such as Browserify and Webpack brings NPM's power to the web. Versioning is handled easily and you get most of the Node.js ecosystem. Handling of CSS is still less than optimal though.

One thing you'll want to consider is how to handle building on your deployment server. Unlike Ruby's Bundler, NPM uses wildcard versions, and packages can change between the time you finish coding and you start deploying. Use a shrinkwrap file to freeze your dependencies (I recommend using Uber's shrinkwrap to get more consistent output). Also consider hosting your own private NPM server using something like Sinopia.

Babel will compile ES6 module syntax to CommonJS. You'll get a future proof syntax, and the benefits of static code analysis, such as tree shaking when using a build tool that supports it (Webpack 2.0 or Rollup).

Build tool: Webpack

Unless you fancy adding hundreds of script tags to your pages, you'll need a build tool to bundle your dependencies. You also need something to allow NPM packages to work in browsers. This is where Webpack comes in.

A year ago you had a lot of potential options here. Your environment, such as Rails' sprockets could do it. RequireJS, Browserify and Webpack were the JavaScript-based solutions. Now, RollupJS promises to handle ES6 modules optimally.

After trying them all, I highly recommend Webpack:

- It is more opinionated yet can be configured to handle even the craziest scenarios.
- All main module formats (AMD, CommonJS, globals) are supported.
- It has features to fix broken modules.
- It can handle CSS.
- It has the most comprehensive cache busting/hashing system (if you push your stuff to CDNs).
- It supports hot reload out of the box.
- It can load almost anything.
- It has an impressive list of optimizations.

Webpack is also by far the best for handling extremely large SPA applications with built-in code splitting and lazy loading.

Be warned that the learning curve is brutal! But once you get it, you'll be rewarded with the best build system available.

But what about Gulp or Grunt? Webpack is much better at processing assets. They can still be useful if you need to run other kind of tasks though (usually you won't). For basic tasks (such as running Webpack or ESlint), I recommend simply using NPM scripts.

Testing: Mocha + Chai + Sinon (but it's not that simple)

There are a LOT of options for unit testing in JavaScript, and you can't go wrong with any of them. If you have unit tests, that's already good!

Lodash is by far the king and contains the entire kitchen sink.

Some choices are Jasmine, Mocha, Tape and AVA and Jest. I'm sure I'm forgetting some. They all have something they do better than the rest.

My criteria for a test framework are as follow:

- It should work in the browser for ease of debugging.
- It should be fast.
- It should easily handle asynchronous tests.
- It should be easy to use from the command line.
- It should let me use whatever assertion and mock library I want.

The first criteria knocks out AVA (even though it looks awesome) and Jest (auto-mocking isn't nearly as nice as it sounds, and is very slow anyway).

You can't really go wrong with Jasmine, Mocha or Tape. I prefer Chai asserts because of all the available plugins and Sinon's mocks to Jasmine's built-in construct, and Mocha's asynchronous test support is superior (you don't have to deal with done callbacks).

Chai as Promised is amazing. I highly recommend using Dirty Chai to avoid some headaches though. Webpack's mocha-loader lets you automatically run tests as you code.

For React specific tooling, look at Airbnb's Enzyme and Teaspoon (this isn't the Rails-based Teaspoon).

I really enjoyed Mocha's features and support. If you want something more minimalist, read this article about Tape.

Edit 3/11/2016: Since I posted this, Facebook posted this article on how they make Jest scale. Probably a bit too complex for most people, but if you have the resources and don't care about running things in the browser, you can probably make it awesome.

Additionally, a bunch of people thought I dismissed AVA too quickly. Don't get me wrong, AVA is amazing. One of my criteria, however, is full browser support, so people can run them straight from any browser (to test cross browser support) and debug them easily. If you don't care about that, you can use the fantastic iron-node for debugging. Coupled with the bundled power asserts, it would easily beat my Mocha setup.

Utility library: Lodash is king, but look at Ramda

JavaScript doesn't have a strong core of utilities like Java or .NET does, so you'll most likely want to include one.

Lodash is by far the king and contains the entire kitchen sink. It is also one of the most performant, with features such as lazy evaluation. You don't have to include the whole thing if you don't want to, either: Lodash lets you include only the functions you use (pretty important considering how large it has become). As of 4.x, Lodash also natively supports an optional "functional" mode for the FP geeks among us. See how to use it here.

If you're into functional programming, however, take a look at the fantastic Ramda. If you decide to use it, you might still need to include some Lodash functions (Ramda is focused on data manipulation and functional construct almost exclusively), but you'll get a lot of the power of functional programming languages in a JavaScript-friendly way.

Http requests: Just use fetch!

Many React applications don't need jQuery at all anymore. Unless you're working on a legacy application or have 3rd party libraries that depend on it, there's no reason to include it. That means you need to replace $.ajax.

I like to keep it simple and just use fetch. It's promise based, it's built in Firefox and Chrome, and it Just Works™. For other browsers, you'll need to include a polyfill. I suggest isomorphic-fetch, to ensure you have all your bases covered, including server-side.

There are other good libraries such as Axios, but I haven't

needed much beyond fetch.

For more details about why promises are important, see my post on asynchronous programming.

Styling: Consider CSS modules

This is an area I feel is lagging behind. Sass is the current go to, and node-sass is a great way to use it in your JavaScript project. That said, I feel it's missing a lot to be a perfect solution. Lack of reference imports (a way to import just variables and mixins from a file, without duplicating selectors) and native URL rewriting makes it harder than needed to keep things lean and clean in production. Node-sass is a C library, and will have to be kept in sync with your Node version.

LESS does not suffer from these issues, but has fallen out of favor because it lacks many of Sass' features.

PostCSS is much more promising, allowing you to kind of "make your own CSS processor". I'd recommend using it on its own, or even in ADDITION to your preferred processor for things such as AutoPrefixer instead of importing a big library like Bourbon.

One thing worthy of attention though, are CSS modules. CSS modules prevent the "cascading" part of CSS, allowing us to keep our dependencies explicit, and prevent conflict. You'll never have to worry about overriding classes by accident or having to make ultra explicit names for your classes. It works great with React, too. One drawback: css-loader with CSS modules enabled is REALLY slow, so if you plan on having hundreds of kilobytes of CSS, you may want to avoid it until it gets better.

If I were to start a large project from scratch today, I'd probably just use PostCSS along with pre-compiled versions of my favorite CSS libraries.

Regardless of what you choose, you may want to look at my post on CSS performance with Webpack, especially if you go with Sass.

If I were to start a large project from scratch today, I'd probably just use PostCSS along with pre-compiled versions of my favorite CSS libraries.

Universal (Isomorphic) JavaScript: Make sure you need it.

Universal or Isomorphic JavaScript refers to JavaScript that can be used on both the client and the server. This is primarily used to pre-render pages server-side for performance and SEO purpose. Thanks to React, what was once only the realm of giants such as Ebay or Facebook is now within reach of most development shops. It is still not "free" though, adding significant complexity and limiting your options in term of libraries and tooling.

If you are building a B2C (Business to Customer) website, such as an e-commerce website, you may not have a choice but to go that route. For internal web or B2B (Business to Business) applications however, that kind of initial performance may not be required. Discuss with your product manager to see if the cost:benefit ratio is worth the trouble.

The API: There's still no true answer.

It seems everyone lately is asking themselves what to do for API. Everyone is jumping in the RESTful API bandwagon, and SOAP is a memory of the past. There are various specifications such as HATEOAS, JSON API, HAL, GraphQL among others.

GraphQL gives a large amount of power (and responsibility) to the client, allowing it to make nearly arbitrary queries. Along with Relay, it handles client state and caching for you. Implementing the server-side portion of GraphQL is difficult and most of the documentation is for Node though.

Netflix's Falcor looks like it will eventually give us a lot of what GraphQL/Relay offers, with simpler server requirements. It is however only a developer preview and not ready for prime time.

All the well-known specifications have their quirks. Some are overly complex. Some only handle reads and don't cover update. Some stray significantly from REST. Many people choose to make their own, but then have to solve all the design problems on their own.

I don't think any solution out there is a slam dunk, but here's what I think your API should have:

- It should be predictable. Your endpoints should follow consistent conventions.
- It should allow fetching multiple entities in one round trip: needing 15 queries to fetch everything you need on page load will give poor performance.
- Make sure you have a good update story: many specifications only covers reads, and you'll need to update stuff sometimes.
- It should be easy to debug: looking at the Chrome inspector's network tab should easily let me see what happened.
- It should be easy to consume: I should be able to easily consume it with fetch, or have a well-supported client library (like Relay)

I haven't found a solution that covers all of the above. If there is one, let me know.

Consider looking at Swagger to document your API if you go the standard RESTful path.

Desktop applications: Electron

Electron is the foundation of the great Atom editor and can be used to make your own applications. At its core, it is a version of Node that can open Chrome windows to render a GUI, and has access to the operating system's native APIs without a browser's typical security sandboxing.

You'll be able to package your application and distribute it like any other desktop application, complete with an installer and auto-updates.

This is one of the easiest ways to make an application that can run on OSX, Windows and Linux while reusing all the tools listed above. It is well-documented and has a very active community.

You may have heard of nw.js (formerly node-webkit), which has existed longer (and does almost the same thing), but Electron is now more stable and is easier to use.

Take a look at this great boilerplate to play around with Electron, React and hot reload. You'll probably want to start from scratch if you're serious about making your own application so you'd understand how all the pieces work.

Everyone is jumping in the RESTful API bandwagon, and SOAP is a memory of the past.

Who to follow and where to learn more?

This is a place where I'm falling short, but on Twitter I follow the following people:

The most important thing to remember is to keep it simple and only use what you need.

- Dan Abramov, author of Redux.
- Christopher Chedeau aka Vjeux, a React developer at Facebook very active in the community.
- Jeff Morrison, one of the main Flow contributors.
- Sebastian Markbåge, another React developer at Facebook, involved with the TC39.
- Pete Hunt. Originally from Instagram and Facebook, famous for his Rethinking best practices talk, partly responsible for putting React on the map.
- React, because duh.
- The above are more are aggregated in React Influencers.

While there's many more worth noting, those people retweet almost anything worth looking at, so they're a good start.

Consider reading Pete Hunt's Learning React. Follow the order!

Dan Abramov published the Getting started with Redux video series. I can't overstate how amazing it is at teaching Redux.

The official Redux FAQ.

Dan also published his own list, and it's probably better than mine.

Mark Erikson's collection of React/Redux links is an ever growing gold mine.

Read Removing user interface complexity, or why React is awesome to get a walkthrough of how React is designed and why.

If you don't need it, don't use it

The JavaScript ecosystem is thriving and moving quickly, but there's finally an end at the light of the tunnel. Best practices are no longer changing constantly, and it is becoming increasingly clear which tools are worth learning.

The most important thing to remember is to keep it simple and only use what you need.

Is your application only 2–3 screens? Then you don't need a router. Are you making a single page? Then you don't even need Redux, just use React's own state. Are you making a simple CRUD application? You don't need Relay. Are you learning ES6? You don't need Async/Await or Decorators. Are you just starting to learn React? You don't need Hot reload and server rendering. Are you starting out with Webpack? You don't need code splitting and multiple chunks. Are you starting with Redux? You don't need Redux-Form or Redux-Sagas.

Keep it simple, one thing at a time, and you'll wonder why people ever complained about JavaScript fatigue.

Did I miss anything?

And there you have it, my view of the current state of JavaScript. Do you think I forgot an important category? Do you think I'm objectively wrong on one of these choices? Do you recommend something else? Let me know! ■

Reprinted with permission of the original author. First appeared at medium.com/javascript-and-opinions.

Trackers

By JACQUES MATTHEIJ

A couple of weeks ago I went to the local shopping centre looking for a thermometer. After entering one store and upon leaving without buying anything, a tracker was assigned to me. I didn't think much of it at first, but he followed me dutifully around the shopping centre, and took careful note of how I walked.

Whenever I visited a store, he made a note in his little black book (he kept calling it my profile, and he didn't want to show me what was in it so I assume it was actually his, rather than mine).

Each of those stores of course assigned trackers to me as well and soon enough I was followed by my own personal veritable posse of nondescript guys with little black books making notes.

After doing my shopping I went home. To my surprise, they expected to come into the house with me and stay there, which I objected.

That didn't stop them. Instead of walking in with me through the front door, they forced open the back door and installed themselves at my table. One of them had found my mobile phone and was going through my list of contacts, adding the names and the telephone numbers of the people that I knew to a thing they called a 'social graph.'

It mattered a lot to them, apparently, and even though I took the phone away from him and made him wipe the copy that he'd made, I realized that if they did the same to my friends (who likely would not take as forceful a stance against the trackers as I did), they'd already have most of that information anyway and there was nothing I could do about it.

Every visitor to my house was asked a whole pile of personal questions, and if they didn't answer, their photographs would be used to complete the gaps in their profiles and mine.

During the night they'd been up to something because the next morning my newspaper had been cut up, with all kinds of blank windows between the articles. They assured me that all of this was entirely legal, and that may be so, but it left me with a weird feeling that these anonymous entities would know more about me than my own mother does.

Upset about this, I decided to read my newspaper in the park, where I expected at least a little bit more privacy. I probably should have known better. The trackers of course followed me to the park and set up an impromptu auction of the space created by the holes in my paper.

Every time I would open a page the space on that page would come up for auction, and the tracker that won the auction would quickly glue an advertisement that he'd brought with him over the gap in the page. Mysteriously quite a few of the ads were for thermometers, even though I no longer had a need for one (having found one in

the drawer in the bathroom that same evening).

The guy the newspaper hired to do the auctions thought this was a-ok, and encouraged the trackers to bid even higher based on all the stuff that he'd told them about me.

The information they used included all the stuff in 'my' profile (some of which came as a surprise to me, for instance, he knew roughly what I earned, knew the fact that I had kids and a whole raft of other details that I did not consider to be any of his business, and even today I have no idea how he got that information) as well as our current location on that park bench.

Some of the more enterprising trackers took down the stuff the newspaper guy told them about me in their own little black books and soon were auctioning that information off to other trackers and to anybody that was willing to part with some money. It became quite a crowd (a mini WallStreet, actually) around my chosen park bench and the atmosphere turned bad, to the point where I did not feel as if the park had anything to offer any longer.

I figured maybe I'd go for some lunch at a nearby restaurant. The trackers had long since figured out my mobile phone number. They combined a little embedded code in a game that I'd once installed (and played twice, although it wasn't a very good game). It kept a continuous read-out of my phone and all its sensors, and phoned home to the mother ship whenever it could.

I probably shouldn't have been surprised when just as I was passing by the door of one particular restaurant chain, an ad popped up overlaying my mobile screen indicating that the restaurant had my favorite dish on sale that day. (How did they figure out my favorite dish anyway? Was it because I had searched for the recipe a few weeks ago?).

I resisted the temptation and decided to go *anywhere* but there. (Even if that meant the trackers could still influence me. At least they'd have to add an inversion to all their little plans).

Finally after a couple of weeks of this I decided I'd had enough. The final straw was when a tracker popped up in my bed, between me and my wife when I was looking for the shortest route to my meeting the next morning. Really, that did it.

The advertising industry did not have the right to become involved in my private life to this degree, to track my every move, and to keep detailed information of my daily whereabouts and interactions in the real world. And the last place I expected them to track me to was in the privacy of my bedroom.

I hired a couple of 'tough guys', bouncers, really. Their assignments were simple: keep these advertising types out of my life, hurt them if they have to. I've *really* had enough of it.

Another side benefit of this was that the burglars and other shady types that used the information gathered by the trackers to target me for an entirely different kind of operation were also shut out.

After the first couple of run-ins between the bouncers and trackers, the trackers got the message and mostly left me alone. Some still tried but for the most part it seems I was at least a little bit safer from prying eyes.

Even more powerful methods of getting rid of trackers exist and there are many variations to choose from, with different functionality. For some victims however, even bouncers were not enough to regain their much deserved privacy.

Some of the bouncers were trying to be a little bit too clever for my taste: they actually took the place of the trackers they blocked. Serves me right for hiring tough guys, I thought, so I've been more careful since then and now check out carefully which party gets to place themselves between me and the trackers.

Even with the tough guys in tow there were still ways in which I could be tracked. But at least that seemed to, for now, be beyond the pale for most respectable businesses employing trackers.

Some venues did not allow me to enter with my bouncers in tow, so I stopped visiting. It's not that I minded their advertising to me, but that I minded being followed, and if they wouldn't change their ways then I would have to change mine.

All of this is really too bad, since some of the stores depended to a certain extent on my patronage. But I guess there are not enough people that are concerned about privacy to make a difference. Or are there? ■

Reprinted with permission of the original author. First appeared at jacquesmattheij.com.

Kill your dependencies

By MIKE PERHAM

This post talks about Ruby but it's true of every language community: Python, JavaScript, Java, etc. The scourge of dependencies spares no one.

This is a dependency visualization of every Rails app I've ever used. See figure below for a dependency visualization of every Rails app.

▲ Rails app dependency visualization

Does any of this sound familiar?

- Gemfile with 100s of entries.
- Test gems loading in production.
- Each Rails process takes 100s of megabytes of RAM.

The Rubygems system is commendable for how easy it makes packaging up Ruby for others to reuse. But that very ease means it's also quite easy for those gems to pull in other gems transitively, leading to Rails apps which "download the Internet" and have hundreds of dependencies.

When you publish a Rubygem, every one of your dependencies transitively becomes a dependency for any app using your gem. This multiplies the impact of bugs in those gems.

The curious case of mime-types

The `mime-types` gem [recently optimized its memory usage] and saved megabytes of RAM. Literally every Rails app in existence can benefit from this optimization because Rails depends on the mime-types gem transitively: `rails -> actionmailer -> mail -> mime-types`.

In other words, this gem wasn't used by your app. It wasn't used by Rails directly. It wasn't used by ActionMailer directly. It was used deep in the bowels of the ActionMailer implementation *and it was using far too much memory*. Every single Rails app in existence was using 10MB too much due to this issue.

App developers, listen up!

Every dependency in your application has the potential to bloat your app, to destabilize your app, to inject odd behavior via monkeypatching or buggy native code. When you are considering adding a dependency to your Rails app, it's a good idea to do a quick sanity check, in order of preference:

1. Do I really need this at all? Kill it.
2. Can I implement the required minimal functionality myself? Own it.

If you need a gem:

1. Does the gem have a native extension? Look for pure ruby alternatives.
2. Does the gem transitively pull in a lot of other gems? Look for simpler alternatives.

Gems with native extensions can destabilize your system; they can be the source of mysterious bugs and crashes. Avoid gems which pull in more dependencies than their value warrants. Example of a bad gem: the `fog` gem which pulls in 39 gems, more dependencies than rails itself and most of which are unnecessary.

Lastly, make sure you only load the gem when necessary. Use Bundler's group support to disable test gems when not testing:

```
group :test do
  gem 'rspec'
  gem 'timecop'
  # etc
end
```

Gem developers, listen up!

Part of your job as a library author is to treat your user and their application with respect. You should make an effort to minimize your own dependencies so they don't load unnecessary code or cause issues in the user's application.

You control your own code but you don't control your dependencies. Any bug in a dependency of yours becomes a bug that causes stress for your user and their application.

As a gem developer, for each of your gem dependencies do you:

- Know how much memory each takes?
- Know how long each takes to require?
- Know whether it performs any monkeypatching outside of its own module?

Sidekiq, with all of its functionality, has only 3 runtime dependencies: `concurrent-ruby`, `connection_pool` and `redis`.

Die json, die (German for "The json, the")

So many gems declare a dependency on json, oj, multi_json, or yajl-ruby. There are so many ossified layers of cruft around JSON processing that only one course of action makes sense: remove it all. JSON has been in the stdlib since 1.9 and you don't need to declare any dependencies at all. Just `require 'json'` and let Ruby deal with it.

Rails did it, so can you!

Why choose an HTTP client when you can have them all?

Every Rails app pulls in a half dozen different HTTP clients: faraday, rest-client, httparty, excon, typhoeus, curb, etc. This is because various gems use them internally.

A Rubygem should never use anything but Net::HTTP internally! Learn the Net:HTTP API, kill those dependencies and stop forcing extra HTTP client gems on your users.

Let's say you want to offer an optimized version using curb: ok, but make it optional. Allow the application developer to opt into using curb but net/http should always be the default.

Optimizing Rails 5.0

For the last few weeks, I've been working (in tandem with several other developers, hi @_matthewd, @applerebel!) on minimizing gem dependencies in Rails 5.0.

- Rails 4.2.5 required 34 gems.
- Rails 5.0b1 required 55 gems.
- Rails 5.0b2 required 39 gems.

I expect Rails 5.0 to require 37 gems or less. So far we've removed Celluloid, EventMachine, thread_safe, and json.

Unfortunately there's no more low-hanging fruit. I'd love to drop Nokogiri, it's such a huge dependency with a massive native extension component, but there are some non-trivial dependencies on it. Oga is a nice, simpler alternative. If you ship a gem which depends on Nokogiri, consider making it optional and defaulting to REXML (I know, but at least it's in stdlib) or Oga instead.

Be part of the solution

I can help with Rails 5.0 but I can't fix every gem. If you are a gem developer, audit your own dependencies and remove as many as you can. If you're an app developer, take a look in your Gemfile and see if you can find a gem or two to remove. Simplify, simplify, simplify.

As an example, I think it's possible for the Stripe gem to remove both of its runtime dependencies.

Rules to remember

Some software engineering rules:

- No code runs faster than no code.
- No code has fewer bugs than no code.
- No code uses less memory than no code.
- No code is easier to understand than no code.

Kill those dependencies. Your gems and apps will be better for it. ■

Reprinted with permission of the original author and mikeperham.com.

Opinion

“We only hire the best” means we only hire the trendiest

By DAN LUU

An acquaintance of mine, let's call him Mike, is looking for work after getting laid off from a contract role at Microsoft, which has happened to a lot of people I know. Like me, Mike has 11 years in the industry. Unlike me, he doesn't know a lot of folks at trendy companies, so I passed his résumé around to some engineers I know at companies that are desperately hiring. My engineering friends thought Mike's resume was fine, but most recruiters rejected him in the resume screening phase.

My engineering friends thought Mike's resume was fine, but most recruiters rejected him in the resume screening phase.

When I asked why he was getting rejected, the typical response I got was:

1. Tech experience is in irrelevant tech.
2. "Experience is too random, with payments, mobile, data analytics, and UX."
3. Contractors are generally not the strongest technically.

This response is something from a recruiter that was relayed to me through an engineer; the engineer was incredulous at the response from the recruiter. Just so we have a name, let's call this company TrendCo. It's one of the thousands of companies that claims to have world class engineers, hire only the best, etc. This is one company in particular, but it's representative of a large class of companies and the responses Mike has gotten.

Anyway, (1) is code for "Mike's a .NET dev, and we don't like people with Windows experience."

I'm familiar with TrendCo's tech stack, which multiple employees have told me is "a tire fire." Their core systems top out under 1k QPS, which has caused them to go down under load. Mike has worked on systems that can handle multiple orders of magnitude more load, but his experience is, apparently, irrelevant.

(2) is hard to make sense of. I've interviewed at TrendCo and one of the selling points is that it's a startup where you get to do a lot of different things. TrendCo almost exclusively hires generalists but Mike is, apparently, too general for them.

(3), combined with (1), gets at what TrendCo's real complaint with Mike is. He's not their type. TrendCo's median employee is a recent graduate from one of maybe ten "top" schools with 0-2 years of experience. They have a few experienced hires, but not many, and most of their experienced hires have something trendy on their resume, not a boring old company like Microsoft.

Whether or not you think there's anything wrong with having a type and rejecting people who aren't your type, as Thomas Ptacek has observed, if your type is the same type everyone else is competing for, "you are competing for talent with the wealthiest (or most overfunded) tech companies in the market."

If you look at new grad hiring data, it looks like FB is offering people with zero experience greater than $100k/ salary, $100k signing bonus, and $150k in RSUs, for an amortized total comp greater than $160k/yr, including $240k in the first year.

Google's package has greater than $100k salary, a variable signing bonus in the $10k range, and $187k in RSUs. That comes in a bit lower than FB, but it's much higher than most

It's much easier to hire people who are underrated, especially if you're not paying market rates.

companies that claim to only hire the best are willing to pay for a new grad. Keep in mind that compensation can go much higher for contested candidates, and that compensation for experienced candidates is probably higher than you expect if you're not a hiring manager who's seen what competitive offers look like today.

By going after people with the most sought after qualifications, TrendCo has narrowed their options down to either paying out the nose for employees, or offering non-competitive compensation packages.

TrendCo has chosen the latter option, which partially explains why they have, proportionally, so few senior devs – the compensation delta increases as you get more senior, and you have to make a really compelling pitch to someone to get them to choose TrendCo when you're offering $150k/yr less than the competition. And as people get more experience, they're less likely to believe the part of the pitch that explains how much the stock options are worth.

Just to be clear, I don't have anything against people with trendy backgrounds. I know a lot of these people who have impeccable interviewing skills and got 5-10 strong offers the last time they looked for work.

I've worked with someone like that: he was just out of school, his total comp package was north of $200k/year, and he was worth every penny.

But think about that for a minute. He had strong offers from six different companies, of which he was going to accept at most one. Including lunch and phone screens, the companies put in an average of eight hours apiece interviewing him.

And because they wanted to hire him so much, the companies that were really serious spent an average of another five hours apiece of engineer time trying to convince him to take their offer.

Because these companies had, on average, a one-sixth chance of hiring this person, they have to spend at least an expected (8+5) * 6 = 78 hours of engineer time[1].

People with great backgrounds are, on average, pretty great, but they're really hard to hire. It's much easier to hire people who are underrated, especially if you're not paying market rates.

I've seen this hyperfocus on hiring people with trendy backgrounds from both sides of the table, and it's ridiculous from both sides.

On the referring side of hiring, I tried to get a startup I was at to hire the most interesting and creative programmer I've ever met, who was tragically underemployed for years because of his low GPA in college.

We declined to hire him and I was told that his low GPA meant that he couldn't be very smart. Years later, Google took a chance on him and he's been killing it since then. He actually convinced me to join Google, and at Google, I tried to hire one of the most productive programmers I know, who was promptly rejected by a recruiter for not being technical enough.

On the candidate side of hiring, I've experienced both being in demand and being almost un-hireable. Because I did my undergrad at Wisconsin, which is one of the 25 schools that claims to be a top 10 CS/engineering school, I had recruiters

beating down my door when I graduated.

But that's silly – that I attended Wisconsin wasn't anything about me; I just happened to grow up in the state of Wisconsin. If I grew up in Utah, I probably would have ended up going to school at Utah. When I've compared notes with folks who attended schools like Utah and Boise State, their education is basically the same as mine.

Wisconsin's rank as an engineering school comes from having professors who do great research which is, at best, weakly correlated to effectiveness at actually teaching undergrads. Despite getting the same engineering education you could get at hundreds of other schools, I had a very easy time getting interviews and finding a great job.

I spent 7.5 years in that great job, at Centaur. Centaur has a pretty strong reputation among hardware companies in Austin who've been around for a while, and I had an easy time shopping for local jobs at hardware companies. But I don't know of any software folks who've heard of Centaur, and as a result I couldn't get an interview at most software companies. There were even a couple of cases where I had really strong internal referrals and the recruiters still didn't want to talk to me, which I found funny and my friends found frustrating.

When I could get interviews, they often went poorly. A typical rejection reason was something like "we process millions of transactions per day here and we really need someone with more relevant experience who can handle these things without ramping up."

And then Google took a chance on me and I was the second person on a project to get serious about deep learning performance, which was a 20%-time project until just before I joined.

We built the fastest deep learning system in the world. From what I hear, they're now on the Nth generation of that project, but even the first generation thing we built has better per-node performance and performance per dollar than any other production system I know of today, years later (excluding follow-ons to that project, of course).

While I was at Google I had recruiters pinging me about job opportunities all the time. And now that I'm at boring old Microsoft, I don't get nearly as many recruiters reaching out to me.

I've been considering looking for work[2] and I wonder how trendy I'll be if I do. Experience in irrelevant tech? Check! Random experience? Check! Contractor? Well, no. But two out of three ain't bad.

My point here isn't anything about me. It's that here's this person[3] who has wildly different levels of attractiveness to employers at various times, mostly due to superficial factors that don't have much to do with actual productivity.

This is a really common story among people who end up at Google. If you hired them before they worked at Google, you might have gotten a great deal! But no one (except Google) was willing to take that chance.

There's something to be said for paying more to get a known quantity, but a company like TrendCo that isn't willing to do that cripples its hiring pipeline by only going after people with trendy resumes.

I don't mean to pick on startups like TrendCo in particular. Boring old companies have their version of what a trendy background is, too.

A friend of mine who's desperate to hire can't do anything with some of the resumes I pass his way because his group isn't

And now that I'm at boring old Microsoft, I don't get nearly as many recruiters reaching out to me.

allowed to hire anyone without a degree. Another person I know is in a similar situation because his group won't talk to people who aren't already employed.

Not only are these decisions non-optimal for companies, they create a path dependence in employment outcomes that causes individual good (or bad) events to follow people around for decades. You can see similar effects in the literature on career earnings in a variety of fields.[4]

Thomas Ptacek has this great line about how "we interview people whose only prior work experience is "Line of Business .NET Developer," and they end up showing us how to write exploits for elliptic curve partial nonce bias attacks that involve Fourier transforms and BKZ lattice reduction steps that take 6 hours to run."

If you work at a company that doesn't reject people out of hand for not being trendy, you'll hear lots of stories like this. Some of the best people I've worked with went to schools you've never heard of and worked at companies you've never heard of until they ended up at Google. Some are still at companies you've never heard of.

If you read Zach Holman, you may recall that when he said that he was fired, someone responded with "If an employer has decided to fire you, then you've not only failed at your job, you've failed as a human being." A lot of people treat employment status and credentials as measures of the inherent worth of individuals. But a large component of these markers of success, not to mention success itself, is luck.

Solutions?

I can understand why this happens. At an individual level, we're prone to the fundamental attribution error. At an organizational level, fast growing organizations burn a large fraction of their time on interviews, and the obvious way to cut down on time spent interviewing is to only interview people with "good" qualifications. Unfortunately, that's counterproductive when you're chasing after the same tiny pool of people as everyone else.

Here are the beginnings of some ideas. I'm open to better suggestions!

Moneyball

Billy Beane and Paul Depodesta took the Oakland A's, a baseball franchise with nowhere near the budget of top teams, and created what was arguably the best team in baseball by finding and "hiring" players who were statistically underrated for their price.

The thing I find really amazing about this is that they publically talked about doing this, and then Michael Lewis wrote a book, titled Moneyball, about them doing this. Despite the publicity, it took years for enough competitors to catch on that the A's strategy stopped giving them a very large edge.

You can see the exact same thing in software hiring. Thomas Ptacek has been talking about how they hired unusually effective people at Matasano for at least half a decade, maybe more.

Google bigwigs regularly talk about the hiring data they have and what hasn't worked. I believe they talked about how focusing on top schools wasn't effective and didn't turn up employees that have better performance years ago, but that doesn't stop TrendCo from focusing hiring efforts on top schools.

A large component of these markers of success, not to mention success itself, is luck.

Training/mentorship

You see a lot of talk about *Moneyball*, but for some reason people are less excited about… trainingball? Practiceball? Whatever you want to call taking people who aren't "the best" and teaching them how to be "the best".

This is another one where it's easy to see the impact through the lens of sports, because there is so much good performance data. Since it's basketball season, if we look at college basketball, for example, we can identify a handful of programs that regularly take unremarkable inputs and produce good outputs. And that's against a field of competitors where every team is expected to coach and train their players.

When it comes to tech companies, most of the competition isn't even trying. At the median large company, you get a couple days of "orientation", which is mostly legal mumbo jumbo and paperwork, and the occasional "training," which is usually a set of videos and a set of multiple-choice questions that are offered up for compliance reasons, not to teach anyone anything. And you'll be assigned a mentor who, more likely than not, won't provide any actual mentorship. Startups tend to be even worse! It's not hard to do better than that.

Considering how much money companies spend on hiring and retaining "the best," you'd expect them to spend at least a (non-zero) fraction on training. It's also quite strange that companies don't focus more on training and mentorship when trying to recruit.

Considering how much money companies spend on hiring and retaining "the best," you'd expect them to spend at least a (non-zero) fraction on training.

Specific things I've learned in specific roles have been tremendously valuable to me, but it's almost always either been a happy accident, or something I went out of my way to do. Most companies don't focus on this stuff.

Sure, recruiters will tell you that "you'll learn so much more here than at Google, which will make you more valuable," implying that it's worth the $150k/yr pay cut, but if you ask them what, specifically, they do to make a better learning environment than Google, they never have a good answer.

Process/tools/culture

I've worked at two companies that both effectively have infinite resources to spend on tooling. One of them, let's call them ToolCo, is really serious about tooling and invests heavily in tools. People describe tooling there with phrases like "magical," "the best I've ever seen," and "I can't believe this is even possible." And I can see why.

For example, if you want to build a project that's millions of lines of code, their build system will make that take somewhere between 5s and 20s (assuming you don't enable LTO or anything else that can't be parallelized)[5]. In the course of a regular day at work you'll use multiple tools that seem magical, because they're so far ahead of what's available in the outside world.

The other company, let's call them ProdCo pays lip service to tooling, but doesn't really value

You can get better results by hiring undistinguished folks and setting them up for success, and it's a lot cheaper.

it. People describing ProdCo tools use phrases like "world class bad software" and "I am 2x less productive than I've ever been anywhere else," and "I can't believe this is even possible."

ProdCo has a paper on a new build system; their claimed numbers for speedup from parallelization/caching, onboarding time, and reliability, are at least two orders of magnitude worse than the equivalent at ToolCo. And, in my experience, the actual numbers are worse than the claims in the paper.

In the course of a day of work at ProdCo, you'll use multiple tools that are multiple orders of magnitude worse than the equivalent at ToolCo in multiple dimensions. These kinds of things add up and can easily make a larger difference than "hiring only the best."

Processes and culture also matter. I once worked on a team that didn't use version control or have a bug tracker. For every no-brainer item on the Joel test, there are teams out there that make the wrong choice.

Although I've only worked on one team that completely failed the Joel test, every team I've worked on has had glaring deficiencies that are technically trivial (but sometimes culturally difficult) to fix.

When I was at Google, we had really bad communication problems between the two halves of our team that were in different locations. My fix was brain-dead simple: I started typing up meeting notes for all of our local meetings and discussions, and taking questions from the remote team about things that surprised them in our notes.

That's something anyone could have done, and it was a huge productivity improvement for the entire team. I've literally never found an environment where you can't massively improve productivity with something that trivial. Sometimes people don't agree (e.g., it took months to get the non-version-control-using-team to use version control), but that's a topic for another post.

Programmers are woefully underutilized at most companies. What's the point of hiring "the best" and then crippling them? You can get better results by hiring undistinguished folks and setting them up for success, and it's a lot cheaper.

Conclusion

When I started programming, I heard a lot about how programmers are down to earth, not like those elitist folks who have uniforms involving suits and ties. You can even wear t-shirts to work!

But if you think programmers aren't elitist, try wearing a suit and tie to an interview sometime. You'll have to go above and beyond to prove that you're not a bad cultural fit.

We like to think that we're different from all those industries that judge people based on appearance, but we do the same thing, only instead of saying that people are a bad fit because they don't wear ties, we say they're a bad fit because they do, and instead of saying people aren't smart enough because they don't have the right pedigree… wait, that's exactly the same. ■

Footnotes

[1] This estimate is conservative. The math only works out to 78 hours if you assume that you never incorrectly reject a trendy candidate and that you don't have to interview candidates that you "correctly" fail to find good candidates. If you add in the extra time for those, the number becomes a lot larger. And if you're TrendCo, and you won't give senior ICs $200k/yr, let alone new grads, you probably need to multiply that number by at least a factor of 10 to account for the reduced probability that someone who's in high demand is going to take a huge pay cut to work for you.

By the way, if you do some similar math you can see that the "no false positives" thing people talk about is bogus. The only way to reduce the risk of a false positive to zero is to not hire anyone. If you hire anyone, you're trading off the cost of firing a bad hire vs. the cost of spending engineering hours interviewing.

[2] I consider this to generally be a good practice, at least for folks like me who are relatively early in their careers. It's good to know what your options are, even if you don't exercise them. When I was at Centaur, I did a round of interviews about once a year and those interviews made it very clear that I was lucky to be at Centaur. I got a lot more responsibility and a wider variety of work than I could have gotten elsewhere, I didn't have to deal with as much nonsense, and I was pretty well paid. I still did the occasional interview, though, and you should too! If you're worried about wasting the time of the hiring company, when I was interviewing speculatively, I always made it very clear that I was happy in my job and unlikely to change jobs, and most companies are fine with that and still wanted to go through with interviewing.

[3] It's really not about me in particular. At the same time I couldn't get any company to talk to me, a friend of mine who's a much better programmer than me spent six months looking for work full time. He eventually got a job at Cloudflare, and was half of the team that wrote their DNS, and is now one of the world's experts on DDoS mitigation for companies that don't have infinite resources. That guy wasn't even a networking person before he joined Cloudflare. He's a brilliant generalist who's created everything from a widely used javascript library to one of the coolest toy systems projects I've ever seen. He probably could have picked up whatever problem domain you're struggling with and knocked it out of the park. Oh, and between the blog posts he writes and the talks he gives, he's one of Cloudflare's most effective recruiters.

[4] I'm not going to do a literature review because there are just so many studies that link career earnings to external shocks, but I'll cite a result that I found to be interesting: Lisa Kahn's 2010 Labour Economics paper.

There have been a lot of studies that show, for some particular negative shock (like a recession), graduating into the negative shock reduces lifetime earnings. But most of those studies show that, over time, the effect gets smaller. When Kahn looked at national unemployment as a proxy for the state of the economy, she found the same thing. But when Kahn looked at state level unemployment, she found that the effect actually compounded over time.

The overall evidence on what happens in the long run is equivocal. If you dig around, you'll find studies where earnings normalizes after "only" 15 years, causing a large but effectively one-off loss in earnings, and studies where the effect gets worse over time. The results are mostly technically not contradictory because they look at different causes of economic distress when people get their first job, and it's possible that the differences in results are because the different circumstances don't generalize. But the "good" result is that it takes 15 years for earnings to normalize after a single bad setback. Even a very optimistic reading of the literature reveals that external events can and do have very large effects on people's careers. And if you want an estimate of the bound on the "bad" case, check out, for example, the Guiso, Sapienza, and Zingales paper that claims to link the productivity of a city today to whether or not that city had a bishop in the year 1000.

[5] During orientation, the back end of the build system was down so I tried building one of the starter tutorials on my local machine. I gave up after an hour when the build was 2% complete. I know someone who tried to build a real, large scale, production codebase on their local machine over a long weekend, and it was nowhere near done when they got back.

Reprinted with permission of the original author. First appeared at danluu.com.

Writing good code: How to reduce the cognitive load of your code

By CHRISTIAN MACKEPRANG

Low bug count, good performance, easy modification. Good code is high-impact, and is perhaps the main reason behind the existence of the proverbial 10x developer. And yet, despite its importance, it eludes new developers. Literature on the subject usually amounts to disconnected collections of tips. How can a new developer just memorize all that stuff? [Code Complete](), the greatest exponent in this matter, is 960 pages long!

I believe it's possible to construct a simple mental framework that can be used with any language or library and which will lead to good quality code by default. There are five main concepts I will talk about here. Keep them in mind and writing good code should be a breeze.

```
1
2 // This was useful in C to avoid accidentally
3 // typing variable = null. These days it will
4 // confuse most people, with little benefit.
5 if (null != variable) ...
6
7
```

▲ Don't personalize your work in ways that would require explanations.

Keep your personal quirks out of it

You read some article which blows your mind with new tricks. Now you are going to write clever code and all your peers will be impressed.

The problem is that people just want to fix their bugs and move on. Your clever trick is often nothing more than a distraction. As I talked about in *Applying neuroscience to software development*, when people have to digest your piece of code, their "mental stack" fills up and it is hard to make progress.

Don't code "your way". Just follow the coding standards. This stuff is already figured out. Make your code predictable and easy to read by coding the way people expect.

Divide and conquer it

Complex code can often be clarified through modularization, and there are more ways to do this than just creating more functions. Storing the result of long conditionals into a variable or two is a great way to modularize without the overhead of a function call. This will even allow you to compose them into larger conditionals, or to reuse the result somewhere else.

The approach when breaking a problem down should be to have each section as focused as possible, affecting only local state, without mixing in irrelevant issues, and without side-effects if at all possible.

Programming languages and libraries often come with their issues, and abstracting them away can help your code mind its own business. The [Single Responsibility Principle]() is another example of how focused and localized code leads to good design.

TDD, besides coming with its own benefits when done right, has been forcing people to apply certain principles which were previously not as popular. Stateless code was dismissed as slow and unnecessary (see: most old C/C++ code), and now everyone is talking about pure functions. Even if you don't do TDD, you should learn the principles that drive it. Working under new paradigms will turn you into a resilient developer.

```
1
2 // This can be a good way to modularize without
3 // adding excessive function call overhead.
4 valid_user = loggedIn() && hasRole(ROLE_ADMIN)
5 valid_data = data != null && validate(data)
6
7 if (valid_user && valid_data) ...
8
```

▲ I like taking advantage of variables to compartmentalize logic.

Make it discrete and process-able

Your computer and your tools can have as much of a hard time dealing with your code as you, and there is some correlation

```
// Using magic strings will make it impossible
// for your IDE to follow your code.
ServiceLocator.get('serviceName')
```

▲ Using ServiceLocator is an example of design that leads to poor integration with most IDEs.

between the number of preprocessors and mutations you have to apply and how convoluted the code is.

Let's set aside the possible benefits of those additional build tools for a moment. Chances are that they require you to use domain-specific languages such as custom templating, or complex and dynamic data structures such as hash tables. Your IDE generally isn't going to be good at handling this stuff, and locating relevant pieces of code will become harder.

Avoid using language extensions and libraries that do not play well with your IDE. The impact they will have on your productivity will far outweigh the small benefit of easier configuration or saving a few keystrokes with more terse syntax.

Another way to keep the "integrated" part of your IDE relevant is to avoid magic code. Most languages will provide ways for you to write more dynamic code. Abusing these features such as by using magic strings, magic array indexes and custom template language features will lead to a more disconnected codebase.

Generally any features which only a human will know the meaning of will lead you down this road, and it's a hard road to come back from, because if your IDE doesn't understand the code, any refactoring features it has are going to be useless when you want to move to a more static architecture.

Make it readable

Work towards having a predictable architecture. Your teammates will find it easier to locate things, and this will greatly reduce the time it takes them to get something done. *Once you've agreed on an overall architectural structure for your project, make it obvious where the main elements are located.*

Using MVC? Place models, views and controllers in their own folders, not three folders deep or spread across several different places.

I talked about modularization. There can also be excessive modularization, which will usually make code harder to locate. Your IDE might help some, but often you will be torn between having your IDE ignore a vendor/library folder due to it having too much irrelevant code, or having it indexed and dealing with the problem manually. It's a lose-lose situation. Try to use fewer libraries by choosing the ones that cover as many of your needs as possible.

Libraries and tooling can also be a barrier for new developers. I recently built a project using EcmaScript 7 (babel), only to later realize that our junior dev was getting stuck trying to figure out what it all meant. Huge toll on the team's productivity. I underestimated how overwhelming that can be to someone just starting out. Don't use tools that are still too hard to get a grip on. Wait for a better time.

```
DEV_TRANSFORM = -t [$(BABELIFY)]
PROD_TRANSFORM = -t [$(BABELIFY)]
UGLIFY = uglifyjs - -c --screw-ie8
PROD_TARGET = exorcist $(SEARCH_TO).map | $(UGLI
LOCAL_TARGET = exorcist $(SEARCH_TO).map > $(SEA

watch:
    watchify -dv --poll $(DEV_TRANSFORM) $(SEARC
    sudo compass watch web --sourcemap --poll &
```

▲ Actual code from a makefile I wrote. Junior devs can't handle overuse of new tech.

```
1
2 // Use names that convey purpose, don't take
3 // advantage of the language to look clever.
4 fluent().interfaces().areSometimes()
5   .usedJust().toMakeCode().lookCool()
6
7
```

▲ Fluent interfaces have been abused often in recent times.

Make it easy to digest

If you've made it this far, I have good news: this is probably the most important part. Choosing good names is known to be one of the larger issues in software development. Build tools are unlikely to improve on this, and the reason is that computers can't really know the reasoning behind a solution. *You have to document the why. Relevant and contextual variable and function names are a great way to do this.*

Names that convey purpose will even reduce the need for documentation.

Using prefixes in names is a great way to add meaning to them. It's a practice that used to be popular, and I think misuse is the reason it hasn't kept up. Prefix systems like hungarian notation were initially meant to add meaning, but with time they ended up being used in less contextual ways, such as just to add type information.

You have to document the why.

Finally there is always something to be said about keeping a low cyclomatic complexity. What this means is to keep the number of conditional branches as low as possible. Each additional branch will not only add indentation and hurt readability, but will more importantly increase the number of things you have to keep track of.

Conclusion & further reading

These are five simple and overarching concepts, and my goal here was to make your learning easier by giving you boxes into which you can put all those code organization ideas.

Practice focusing on these aspects while programming to solidify them. If you haven't done so yet, I really recommend *Code Complete*. It comes with a large number of examples and will dissect almost any situation you might run into. ■

Reprinted with permission of the original author. First appeared at chrismm.com/blog/.

My biggest regret as a programmer

By ANDREW WULF

A little over 20 years ago I was at a crossroad. My second company was petering out when our 5 years of building Deltagraph for the publisher ended (they wanted to move into the nascent Internet space). At that point I had 13 years' experience as a programmer but also 9 years or so experience running a company (at the same time).

I no longer wanted to do both. My first company (85-87) not only built a new kind of spreadsheet program but also published it ourselves. I led the company, did all the press interviews, managed the investors, did all the usual business stuff and also was one of the three programmers and the UI designer. After we shipped the product in early 87 I also wound up in the hospital. Trying to be both leader and programmer was simply too much.

So at that point in 1994 I could have gone either into technical management or continued as a programmer. I chose programmer because it was easier. Today I realize how wrong I was despite all the great stuff I've been able to work on and ship over the past 20 years. Going towards the CTO/CIO/VP Engineering route, which was fairly new back then, would have been a much better plan.

I was in the Bay Area for a year around 1995 and worked at Apple for the last half. Apple looked to be falling apart and I'd left to return to Texas as I didn't want to see my favorite company die around me. Big mistake.

Not only did Apple begin a huge turnaround a year later when Steve came back, but the whole Dotcom explosion happened.

Being both an experienced programmer and leader who understood what it took to deliver (we did 9 major releases of the apps during my time, all of which I built the master floppies for, with no need for hot fixes which were hard to do then anyway) I can only imagine how in demand I could have been. Once you get to the level of one of those titles you can keep moving forward and up.

My sister started as a programmer 30 years ago but jumped into management within the first year and has been a VP at a big company for the past 15 or so years. The huge parent of the travel company I worked for a couple years ago had a CEO who started 15 years earlier as a programmer. Of course these types of jobs can be hard and unpleasant but for that the remuneration is way greater. My sister has 10X the assets I have.

Over the years I've seen how little ability you have as a programmer, no matter how good you are, at making a difference or changing things that are broken. I simply didn't realize how little room you have to advance as just a programmer (or even as an architect or the like); the power to change exists at a level not available to you as a mere delivery device.

Add to that the financial benefits, the higher likelihood of substantial IPO participation, and all the other things you gain access to, and being a programmer means you have to be happy with the opportunity to build cool things.

Over the years the worst places I've worked for or helped as a consultant for those 5 or so years were almost always due to inept, incompetent or downright idiotic technology management. There isn't enough room in this blog to list them all.

Take the VP of engineering for a bank who remarked that he didn't need to understand technology as he managed people, yet still made technology decisions.

The CIO at the same place never believed anything his employees told him but believed everything vendors told him. Of course we knew he was taking kickbacks as we kept buying things we had no use for and

I came to a fork in the road and took the one less traveled.

he kept writing articles for them relating how wonderful their products were for us. Yet we used almost none of it. Some time after I left he was fired and perp-walked out of the company, yet immediately he got another similar CIO position.

The worst job I ever had started out as what I thought would be awesome. A post-startup had a successful niche in their industry; both they and their arch-rivals (different niche) both wanted to launch into a broader public market and the market was heating up.

I was hired as a second programmer. The other programmer and manager had been hired to build a new broader online store as the existing one was too inflexible and slow for a big market.

The company had zero technical leadership otherwise, and the CEO and the other two founders had no technical experience or knowledge. The programmer constantly talked about how wonderful his backend code was and the manager supported him. I built a front end piece, put up demos, checked in my source every day. When I thought it a good time to integrate I discovered the other programmer, after 10 months, had checked innothing.

When I pointed this out the manager said “he never checks in anything until it’s perfect”. Yet no one called this out as stupid other than me. I spent the next two months trying desperately to get the 3 founders to bring in people who could actually deliver (I knew several people) but they were afraid to make any changes and admit they had screwed up in hiring these two guys. Eventually I gave up and left.

A year later, after still getting nothing from this guy, they fired both of them. They tried to hire some consulting firm but got nothing from them either. By this time it was too late. The rivals? They became a billion dollar public company and I see their commercials on TV sometimes. I always want to throw a shoe at the TV when I see them. We had everything but a damn store and actual technology leadership. If I had been such a person instead of a programmer I would have had the track record and clout to make it happen. But all I was was a programmer.

I could go on and on but the key is that you can’t make changes in how people do things in a technical sense unless you have the ability, authority and opportunity. Once you make that call and assuming you find the right places to grow, the sky is really the limit.

When I was on TV (Computer Chronicles) in early 1987 showing our product Trapeze, the other presenter was Mike Slade who was product manager of Excel. At the time young me thought he was some random marketing weenie (young people can be pretty stupid). Yet he started all these companies later including ESPN, worked for Apple in various leadership roles, was a good friend of Steve Jobs and started his own VC firm.

And today I am still just a programmer. Who’s the weenie now? I doubt I will ever be able to really retire. Thankfully I am still good at delivery (I was recruited into my present job by a former manager who needed what he knew I can do) but still all I will be until I croak is what I am now.

Being a programmer for nearly 35 years and still being able to get things done and ship is still fun and I’ve been able to work on amazing things over the years. But I can still feel the regret of not seeking the challenge of just leadership. In some ways programming was the easy choice. Given how close I got to the whole Dotcom timeframe, or even the return of Steve to Apple, and still had recent leadership experience, I could have been almost anything.

So yes I regret not taking that choice and seeing where it would have led me, yet I would have missed all the fun of writing code and the soul-draining jobs that often come with it where you can’t really fix anything.

I came to a fork in the road and took the one less traveled. Perhaps now I realize why. ■

Reprinted with permission of the original author. First appeared at thecodist.com.

Interview: How to become a better programmer

John Sonmez is the founder of SimpleProgrammer.com, author of the bestselling book *Soft Skills: The Software Developer's Life Manual* and creator of over 50 professional developer courses ranging from iOS, Android, Game Development, Java and more.

In the past few months, we've gotten a number of emails from readers about how to be an all-around more effective programmer. Learning new technologies faster, becoming more relevant, being more productive and interviewing better all came up as hot button topics.

To help us through these topics this month, we chatted with John Sonmez, who's been helping software developers, programmers and other IT professionals, become better at their jobs.

Without further ado, here's John Sonmez.

Can you fill in some gaps in that intro, and then give us a glimpse into your personal life?

Well, I'd say the main thing about me is that I am always interesting, and growing and developing myself and sharing what I learned in the process with others to help them along the way.

My real mission in life right now is to help as many software developers not just get better at their job and make more money, but to really live better, happier, and more fulfilled lives, and to achieve the goals they set for themselves.

I developed software for 15 years before I realized that the most important skills in life are not technical, but rather what I call soft skills.

Once I realized this, I realized that I could really help other software developers get ahead in their career and reach success by taking a much more holistic approach to their lives, instead of just focusing on the technical aspects.

I believe in being transparent, so I do almost everything in public and share as I learn and grow myself.

I live in sunny San Diego with my wife, Heather, and 5-year-old daughter, Sophia.

From your coaching and consulting, you've come across engineers looking to take things to the next level. Can you go into detail about the challenges and help us understand the steps your clients took to get to that next level?

It's difficult to make a generalization about what is holding engineers back from reaching the next level, because so many people have so many different ideas of what success even looks like, as well as very different mental models and struggles. But I'd say that one of the

The first and most important skill is learning to learn.

biggest things is mindset. When you change someone's mindset, you change his/her life.

It seems very basic, but so many people have extremely limiting mindsets. They believe something about how the world works and place artificial limitations on themselves that prevent them from succeeding. They believe they are somehow not good enough or that they are not "lucky," instead of realizing that there are very few things that can't be achieved by a person who is determined and unwilling to give up.

I've coached (personally and through my blog, books, videos and podcasts) tens of thousands of developers to develop the kind of mindset needed to succeed and really reach that elusive next level.

I've had to go through it myself – in fact, I still am. Every single day I try and expand my mindset, and grow a little more and help others along the way.

Bringing things back down to a very practical aspect, I'd say one of the biggest things that software developers and IT professionals don't know how to do, that can greatly impact their career, is to learn how to market themselves. I sell a course on how to market yourself as a software developer. It teaches developers why marketing themselves is so important and critical to success, and how to build a personal brand and specialize to become an expert in a particular area of software development, which can yield huge results.

I've seen huge success myself and with many developers I've worked with. Multiple stories of 3x income increases, new and better jobs, successful freelancing or consulting practices, and just learning how to sell the value you already have better, as well as to learn how to give free value to others in order to build a brand and reputation.

In general, what are the top 3 essential skills a software engineer should develop? What is the most efficient way to develop these skills?

The first and most important skill is learning to learn.

Developing the ability to teach yourself is by far the most important skill you can develop in life – especially for a software developer who has to constantly face changing technologies and is forced to learn at a rapid pace, just to keep up. This skill was so important that I specifically did a course on it called “10 Steps to Learn Anything Quickly.”

I found so many developers doing this wrong or not doing this at all that I felt I had to address it, because it's so important.

I could talk about how important this single skill is for hours.

A good way to develop this skill is to get in over your head and do things that you are not ready for yet, because it forces you to sink or swim and to learn in a limited time frame, focusing on what is essential.

Next, I'd say is a hybrid of discipline and gumption. It's the ability to do what needs to be done, whether you feel like doing it or not. Most success in life comes from going after what most people would have given up. So many people quit too early or get discouraged when they are just a few feet from gold.

To develop this skill, I practice doing one thing I dislike every day, whether I feel like it or not. Discipline comes from learning to produce results even when motivation is lacking.

Finally, I'd say people skills are very important – essential – for just about anyone.

Learning how to properly interact with people and deal with tricky social situations is a skill that is always valuable, because regardless of what our job is, we are almost always dealing with people.

To develop this skill, I recommend reading Dale Carnegie's famous book: "How to Win Friends and Influence People."

How can programmers balance being productive at work and learning new frameworks, tools and languages to stay up-to-date?

Devote some time each day to self-development. I used to walk on the treadmill and read a technical book of some sort for at least half-an-hour each day. This helped me get in shape and keep my technical knowledge sharp.

Another important thing to do is to not learn what you don't need to learn. Only learn what you are actually going to use.

It's great to read blog posts and technical news to stay up-to-date on trends and developments in the field, but don't learn every new framework that comes along. Instead, focus on knowing enough about it so that if you need to learn it in the future, you can. Again, learning to teach yourself effectively will give you the confidence you need to know that you can do just-in-time learning to learn something when the time comes, instead of potentially wasting hours trying to learn something ahead of time that you may or may not actually use.

Let's dial in on staying up-to-date. How can software developers stay informed of what's truly important with everything evolving so quickly in the industry?

Devote time each day to read blogs and news sites like Hacker News. Not a huge amount of time, but perhaps 30 minutes each day will help keep you up-to-date.

Also realize that there is nothing truly new. Everything is always a reinvention of something else from the past.The pendulum always swings back. First it's thin clients, then it's thick clients, then it's thin clients inside of a thick client and back again.One you realize this, absorbing new information and spotting trends becomes a lot easier.

What is the #1 thing holding back programmers from taking things to the next level?

[See John's video response. If you enjoyed his response, subscrbe to his YouTube channel of 23K subscribers.]

Programming interviews are possibly the worst invention in the software industry (sorry, that's Ray's opinion). However, they are a fact of life if you're looking to get hired. What is the #1 thing programmers can do to improve their odds of nailing that interview and getting hired?

Think of them as a filter, not an actual test of your ability to perform at the job.

Don't fight them; instead realize they are an opportunity to artificially thin the field which will actually increase your chances of getting the job.

With that in mind, become an expert at passing these kinds of interviews.

Practice coding on a whiteboard. Practice at sites like Codility and Top Coder to solve the kind of algorithm and data structure problems that big companies like Microsoft and Google will inevitably ask.

I used to fight these programming interviews and complain about them, until I realized I just needed to get good at them. It didn't take very long, but it was a skill that allowed me to land several really good jobs and to feel more confident in general.

Another important thing to do is to not learn what you don't need to learn.

It's up to you right now – in this very moment – to decide to take action and then to follow through and to keep following through until the job is done.

This is one of those times when you have to study for the test and not for the material.

Plus, even though programming interviews may seem silly, the skills I'd developed to pass them have actually been quite helpful in my career in other areas when I had to solve a difficult problem, and I knew just what algorithm and data structure to use.

And since most developers will not do well on these kinds of interviews, since they won't prepare for them, taking the time to do so will give you a huge advantage.

What is a parting piece of advice to Hacker Bits readers?

Realize that while technical abilities are important, they are not the most important. In fact, probably less than 10% of the success you see in your career will come from your technical abilities. It's really the soft skills that make the difference in your career and life, so actively focus on developing those skills.

Stop limiting yourself so much. You hear it over and over again, but you can literally do anything you want right now with your life and no one can stop you but yourself. We live in a time of such incredible freedom and opportunity, yet so many of us let our minds and attitudes defeat us before we've even given it a shot.

If you would have told me that I'd earn my living by inspiring and helping people develop themselves 5 years ago, I would have told you that you were crazy. I had no idea what my own potential was. I had no idea I could do what I wanted to do. I just had to be willing to work for it.

Stop giving up. Stop accepting less. Stop making excuses for yourself. This is your life right now. No one is coming to fix it for you. No one is coming to save you. It's not going to magically get better. Your problems are not going to magically go away. It's up to you right now – in this very moment – to decide to take action and then to follow through and to keep following through until the job is done. Do not let another day go by living the life you are not supposed to live. Do not say "someday I will do this, or someday I will do that." Do not wait to live your life. Live it now.

Give. Don't worry about what you'll get in return. You can't out-give the universe. The more people you help and the more you give, the more you will get. Create value for others, encourage them, and help them along the way. I guarantee you that if you find a way to help 1 million people, you'll earn more than 1 million dollars.

Where can people find out more about you? And where can they pick up a copy of your book?

Go to simpleprogrammer.com. There you'll find my blog posts, links to my YouTube videos, podcasts,products and more.

You can get Soft Skills at simpleprogrammer.com/softskills. I recorded an audio version of the book on Audible.com as well. ■

Interview was edited for brevity and clarity.

Four strategies for organizing code

By MARTIN SANDIN

This article outlines four different strategies for organizing code:

- *by component*
- *by toolbox*
- *by layer*
- *by kind*

I think these four form a kind of hierarchy with regards to which kind of cohesion they favor and in my experience they cover most of the real-world code I've worked with, pleasurable and not. There are an endless number of possible strategies but I've (thankfully) never encountered anyone who organizes packages into projects by creation date or classes into packages by first letter.

The whys and whats of organizing organizing code

It is a funny thing that most of the advice you will hear and read on how to develop software basically prescribes how you should organize your code, a topic that doesn't matter to the computer. As far as the machine is concerned, all this talk about coupling and cohesion is mostly irrelevant; it doesn't care if you put all your code in a single million line method, sort your classes alphabetically, or give all your variables single letter names.

Code organization is not about communicating with the computer but rather all about trying to make sure that *people* can understand the code well enough that they will be able to maintain and evolve it with some degree of efficiency and confidence.

> *"Programs should be written for people to read, and only incidentally for machines to execute."*
>
> *— Structure and Interpretation of Computer Programs by Abelson and Sussman*

When a unit of code grows too large and contains too many elements, it becomes hard to navigate, hard to get an overview of, and hard to understand: it becomes *complex*.

Our main weapon against this complexity is *divide and conquer*: we split the unit into smaller parts which we can understand in isolation.

For classes it is fairly well understood that this should be done so that we create logical objects which exhibit good cohesion and fit well in the domain model.

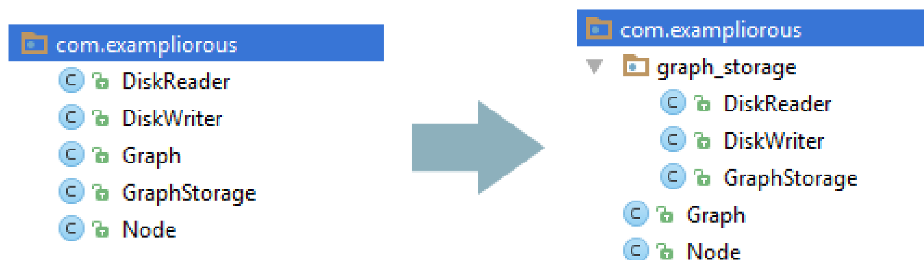
With projects — which are separately compiled — vwe have to break circular dependencies and try to make sure that they expose reasonably logical and stable interfaces to other projects.

On the level in between — *packages* in Java or *namespaces* in C# — there is a lot more variation and in my experience many developers chose a strategy without much consideration given to why that particular strategy should be employed.

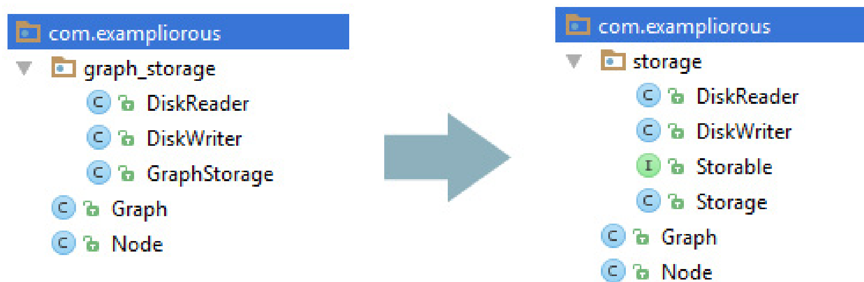
The first three strategies described in this article can be used at either *class*, *package*, or *project* level while the last one — *organization by kind* — is more or less specific to the package level.

Strategy #1 — by component

Organization by component minimizes complexity by emphasizing external and internal cohesion of code units, e.g.



▲ Reduced total complexity doesn't follow unless you take the step of eliminating dependencies



▲ Eliminating dependency

packages. The former means that the package has a minimal interface, which exposes only concepts which that are strongly related to the service the component provides. The latter means that the code in the package is strongly interrelated and thus strongly related to the provided service.

A lot can be and has been written about what constitutes a good unit of abstraction and covering even a sliver of that would make this article too long by far. Suffice to say that the SOLID principles are a great place to start learning, and that practice and reflection on how things are working out and why that might be is paramount.

In this article I will only cover what in my experience is the single most common reason for rampant complexity in code bases where people have actually tried to organize things by divide and conquer: *failure to isolate packages into components*.

New units of code are often created by identifying a subset of the functionality contained in one (or more) existing packages and creating a new abstraction from the corresponding code, resulting in more but smaller units.

This creates code which looks easier to digest but it is mainly window dressing until further steps are taken: the benefit of reduced total complexity doesn't follow unless you then take the step of *eliminating dependencies*.

In my opinion, *packages which have mutual dependencies should not be considered separate units of code at all* as none of them can be understood in isolation from the others.

In the example above. it is easy to imagine that the *Graph* class has a reference to a *GraphStorage* in which it persists itself whenever it has changed.

Not only does the *graph_storage* package depend on a lot of details of the graph package domain model about which it should be rightfully ignorant, the packages also remain mutually dependent. The easiest dependency to eliminate is often that from the new package to the old one (see figure above).

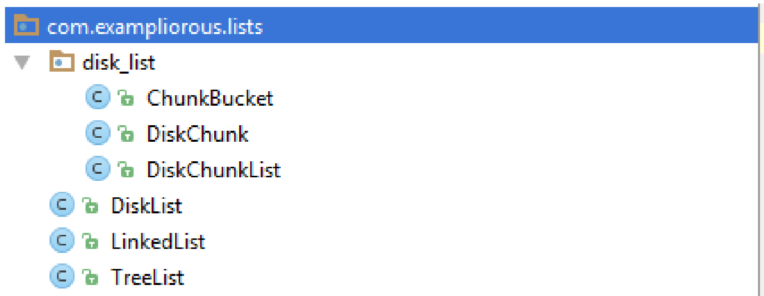
The most important reason that this is an improvement is that when reading the storage code one can now rely on the fact that the only things it need to know about that which it is storing is what is in the *Storable* interface.

> *"No client should be forced to depend on methods it does not use."*
>
> *— The Interface Segregation Principle*

The next step would be to eliminate the direct dependency from the graph package to the storage package. This could for example be done by creating a *GraphPersister* interface in the former and having a higher level package inject an adapter implementation into the *Graph*. And once again the primary benefit would be that the exact set of storage functionality the graph package depends on would become obvious.

> *"...packages which have mutual dependencies should not be considered separate units of code at all..."*

In theory this process might sound fairly easy but it takes a lot of experience to learn to identify suitable components and strategies for isolating them. It is quite common to start the process only to find out that you didn't quite get the abstraction right and need to back out of the change.



▲ Toolbox with a facade for DiskList for the sake of external consistency

The rewards for properly isolating components are great however: *code which is easy to understand, easy to improve, easy to test, and — incidentally — easy to reuse.*

Strategy #2 — by toolbox

Organization by toolbox focuses on external cohesion, providing a consistent toolbox which the consumer can choose from. This strategy is weaker than organizing by component as it drops the requirement for strong internal cohesion, e.g. that the constituents are all strongly interrelated.

The parts of a toolbox are often complementary implementations of the same interface(s) which can be usefully chosen from or combined, rather than sharing a lot in the way of implementation.

- Collection libraries are typically organized as toolboxes with a set of complementary implementations of a set of collection interfaces with varying characteristics, with regards to areas such as time complexity and memory consumption. There might also be a unifying theme to the toolbox, such as only containing disk-based data structures.
- Logging libraries are not necessarily toolboxes in their entirety but often contain a toolbox of e.g. log-writer implementations, which target different destinations.

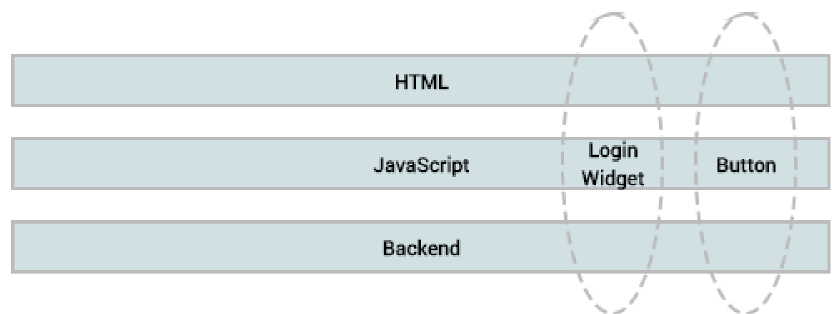
Toolboxes arise because they are convenient to the consumer and each “tool” in the box isn't big enough to warrant its own unit even though they are technically independent.

Each component in a GUI library might for example deserve its own package but giving each its own project is unnecessarily onerous. Similarly each collection implementation might fit in a single class and putting them all in individual packages would be unnecessary bureaucracy.

At least in the latter case a single collection implementation which that grows beyond a couple of classes should get its own package, possibly except for a thin facade for the sake of external consistency.

Strategy #3 — by layer

Organization by layer favors workflow cohesion instead of trying to control complexity by minimizing cross-unit coupling. The code is split along layer boundaries defined by issues such as deployment scenarios or areas of contributor responsibility.



▲ Component coupling across layers

This strategy is different from organization by toolbox in that layers don't present a single, minimal, and coherent interface to the other layers, but instead a wide interface with many constituents which that are accessed piecemeal by the corresponding constituents of the consuming layer.

The typical characteristic of organization by layer is that the logical coupling is stronger within the logical components that span across the layers than within the layers themselves.

The most common failure mode of this strategy is that most changes require touching files across all the layers, basically the textbook definition of tight coupling.

> *"Given two [units of code], A and B, they are coupled when B must change behavior only because A changed."*
>
> *— The C2 wiki*

Under this scenario, logical intra-component dependencies end up like ugly nails driven through your supposedly decoupled layers, pulling them together into a single—often wildly complex—unit.

Organization by layer should be used cautiously as it often increases total system complexity rather than help control it, but there are cases where the benefits it provides outweigh this drawback.

In those cases it's often worth sequestering your layer dependency into a single place in your consumer code rather than having its tendrils reach throughout the entire code base:

- Don't let references to language resource files infiltrate your entire code base, but rather map all results and errors from your internal components to language resource messages in a single place near the presentation layer.
- Don't use the value objects generated from your JSON schema beyond your service layer; , translate them into proper domain objects and calls at the earliest time possible.

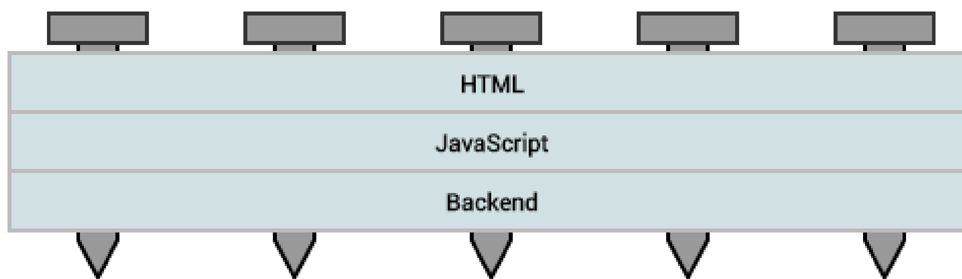
Strategy #4 — by kind

Organization by kind is a strategy which tries to bring order to overly complex units of code by throwing the parts into buckets based on which *kind* of class (or interface, …) it is deemed to be. In doing this it ignores dependencies and conceptual relationships and typically produces packages with names such as *exceptions, interfaces, managers, helpers,* or *entities*.

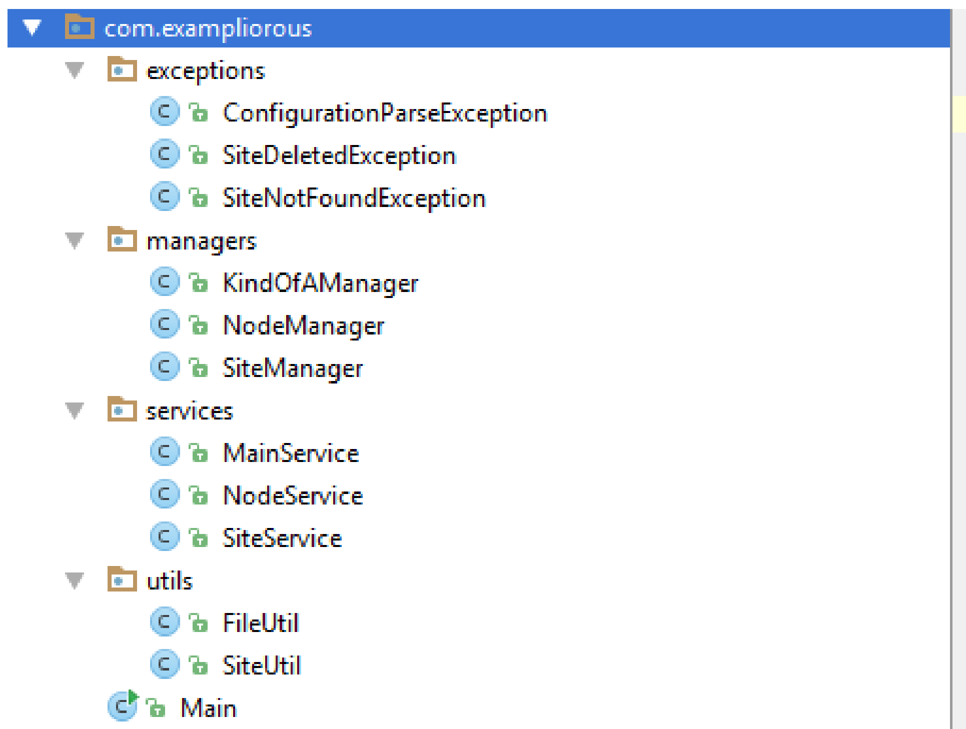
Organization by kind is different from organization by toolbox in that it drops any pretense that the classes in a package are complementary, interchangeable, and/or form any kind of sensible library when put together.

Nobody that I know of is advocating using this strategy for organizing code into separate classes or projects ("*here's the class with all the string members*" or "*here's the project in which we put all our exceptions*").

I consider organizing code by kind harmful as it *hides* the actual problems of complex code and thus make developers feel that they've fixed it while



▲ Layers nailed together into a single — very complex — unit



▲ Project organized by kind

the overall complexity remains the same.

The example above *looks* kind of neat with everything tucked into bite-sized packages, but almost every change requires touching every package, meaning that the packages are in fact tightly coupled.

The other big problem with this strategy is that if it is taken to its extreme, it requires *every* class to be of a clear-cut kind. I've seen this warp entire code-bases as all kinds of strange things get created and designated a *Manager* or a *Helper* just to fit into some package.

> *"...package size isn't the main problem, the number of interdependent parts is."*

I consider organization by kind a code smell but in my experience from commercial projects—mainly in Java and C#—it is quite common. I believe that this happens because it seems to provide an easy way to partition large packages and most people aren't aware that package size isn't the main problem, the number of interdependent parts is.

Summary

Organizing code is a core skill for software developers and as with all skills the most effective way to improve is to reflect on your previous choices and the fallout from them. There are a wide array of different strategies for organizing code and learning to recognize both the useful and the dangerous ones is very important. ■

Reprinted with permission of the original author. First appeared at medium.com/@msandin.

Why I switched to Android after 7 years of iOS

By HENRIK JORETEG

Last Monday I was all excited. I had just gotten the green light to start prototyping a new Progressive Web App for a client I've been working with.

I pulled out an older Android phone that I keep around for development. Then I also got my sleek, new, shiny iPhone 6s out of my pocket, with its smooth curves and speedy OS. But as I looked at my iPhone I was kind of bummed out.

I realized that this slick piece of Apple hardware was less capable as a platform for web applications than my dusty old Android dev phone.

At that point I knew I was over iOS.

So, instead of opening my text editor, I placed an order for a Nexus 6P and signed up for Google fi phone service (which is awesome, btw).

Just like that, after 7+ years, bye bye iOS.

What!? What's wrong with iOS?

Remember the original iPhone announcement? Where Steve introduced the amazing combination of a mobile phone, an iPod, and an *Internet communications device.*

I don't know about you, but the idea of having a fully capable web browser in my pocket was a *huge part* of the appeal.

That's never changed.

Of course I don't know the full backstory, but it sure seemed like the original plan for 3rd party developers on iOS was to have us all just build apps using the web. Safari added support for making web apps that could be added to your home screen as an icon and by including a few magical `<meta>` tags you could use to create something that could sort of be "installed" to your home screen. Then, when you opened it would run in "standalone mode".

These were, in many ways, the original "Progressive Web Apps" and this was sometime around 2009!

Think about it…

1. They started in the browser, but then you could kind of upgrade them to "home screen status".
2. When you ran them from the home screen they'd open with a splash screen and without any visible browser UI.
3. You could pick a loading screen and app icon.
4. You could even pick from a few different status bar colors!

I don't know whether this type of app was actually intended to be the primary mechanism for 3rd party dev to build apps for iOS but regardless,it was way ahead of its time.

Unfortunately, the web platform itself wasn't quite ready for the spotlight yet. It was *sort of* possible to build web apps that looked and performed like native apps, and I was trying to do this 6 years ago using David Kaneda's awesome jQTouch lib. Hilariously, the corny little demo video that I posted led to a call from David and almost got me a job at extjs right as they were rebranding to Sencha and starting to build Sencha Touch. But the story for offline was terrible.

But anyway, as it turned out, the capabilities of the web on iOS were not quite enough to satiate ravenous developers. So developers were left clawing for the ability to build stuff that ran natively on the device to give them better performance and deeper API access.

Enter the iOS SDK and App Store

Apple made what turned out to be a really smart business decision: they released an iOS SDK and an App Store, and the rest is history.

First, I was excited about "apps" just like everyone else seemed to be. Just think, here we had been busy building "applications" when we really should've been building "apps" all along! Who knew?! ;)

Anyway, I quickly found myself hunting for the best apps and switching to whatever bank, social networks, and other services had the best iOS apps. I bought a book on iOS development and built a hello world or two. My old co-worker Ryan Youngman made iSaber to let you swing a fake lightsaber at your friends with your phone. Every developer I knew was talking about iOS development but at some point the fun of all this iOS stuff dried up.

Seeing the hoops you had to jump through to ship an app on iOS didn't seem right.

How quickly developers traded away the wide-open spaces of the web for a walled castle with a monarch enforcing a 30% tax.

So, I decided to focus on building "installable web apps" for iOS instead because surely, the web would catch up.

However, this became problematic

Despite the popularity of native apps, the original idea of these standalone installable web apps has continued to be supported for new versions of iOS. But, they didn't fit into Apple's business model! The App Store turned into a huge business, the term "app" was going mainstream, and every business suddenly felt they needed to have their own "app," whether they had any users or not.

As Apple's app business took off these capabilities very clearly and quickly, and somewhat unsurprisingly were deprioritized. The end result for those of us still trying to build installable web apps for iOS was that with nearly every new iOS release, some key feature that we were depending on broke.

I'm talking about stuff that QA should have caught, stuff that if *anybody* at Apple was actually building apps this way would have noticed before they released.

One quick example that bit me was how they broke the ability to link out to an external website from within an app running in "standalone" mode. `target=_blank` no longer worked, neither did `window.open` or anything else I could think of. So now since our "standalone" app didn't have a URL bar or back button, it would simply take the user to the link they clicked within the same full-screen web view *with no way to return to the app*! The *only* way out was forcibly quitting the app (hopefully the user knew how to do that).

We were running a chat product at the time, so any time someone pasted a URL into chat it was essentially a trap.

These sorts of issues continued to happen release after release. Soon it became obvious that while you can *sort of* build these types of apps on iOS, you can't really depend on them not breaking with the next update.

The message from Apple seemed clear: web apps are second-class citizens on iOS.

What of Android?

I didn't care much at the time, but somewhere in the middle of all of this, Android appeared on the scene. It promised to be a more open alternative as a mobile platform. It was a collaboration between several big companies, and it was their attempt to essentially fight off the fruit-company-comeback-kid-turned-gorilla and its Mighty Joe App Store.

It started gaining traction, but its web experience at the beginning was quite sub-par.

It (Android) is currently the best mobile web app platform.

Fast-forward five years…

1. People are somewhat burnt out on Apps.
2. The vast majority of developers building native iOS apps never made back their expenses. We knew this in 2013.
3. A *few* games are still making money, but that's a lottery.
4. Meanwhile, there are over 1.4 billion active Android users.
5. Android switched to using Chrome as the default browser.
6. Chrome, Opera, and Firefox have added features to allow building actual app experiences via the Web.

And here I am… switching to Android.

So why Android? Isn't it just more of the same?

Yes. It is. Android itself bores me, honestly. There's nothing all that terribly new or exciting here.

Save one very important detail… *IT'S CURRENTLY THE BEST MOBILE WEB APP PLATFORM.*

What do you mean?! Doesn't Safari run my JS faster?

Most people when they say this are referring to this post by Jeff Atwood (a.k.a. codinghorror), which I wrote a whole response post to, if you're interested.

So yeah, Safari runs my JS faster, but guess what… most of your users won't have a shiny new iPhone 6s, and as I've said before, betting on desktop-like performance on the mobile web, or sending huge frameworks like Ember to a mobile device probably isn't a great idea.

With performance, there is such a thing as "good enough." It wouldn't matter if Safari ran JS 50x faster! The only thing that matters is whether my app runs *fast enough*. Beyond that, as a user, I don't care.

As it turns out, it's possible to write web apps that run at 60fps even on older, crappier hardware.

But, all that aside, note this: I said "*better app platform*" not faster JavaScript runtime.

So why not just use Chrome for iOS?!

As I started tweeting about switching, I was surprised to realize that many people don't know that Chrome, Opera, and Firefox for iOS all just use WebKit web views under the hood.

In fact, apps that include a different browser engine are a violation of Apple's terms of service.

They're just different UIs on the same browser engine.

But isn't WebKit getting better?

Yes, it seems like they're picking up some momentum recently.

But, there's a whole lot more to it than just what happens in the browser window. I want the ability to create app-like experiences on the OS with web technology.

Very little seems to be happening in that regard as far as I can tell.

Let's look at Apple and WebRTC

A few years ago, I built SimpleWebRTC and the first version of Talky.io.

I seem to have been one of the early web geeks to get really excited about WebRTC (the browser web technology that now powers Google Hangout video calls). Anyway, I managed to figure out how to build one of the first, possibly *the first* multi-user, peer-to-peer, video calling WebRTC app on the web that worked with more than 2 people and worked between Chrome and FireFox.

This was my first experience with Apple lagging behind in implementing new web APIs. Although Chrome and FireFox were both actively implementing and excited about WebRTC, there was not a peep from Apple. iOS still hasn't added WebRTC support to this day. Though, they've apparently been hiring WebRTC engineers of the Safari team. So here's hoping…

But it kinda makes sense, right? Why would they? They'd rather you use FaceTime, right?

They seem fine with improving *the browser engine*, but seem very slow to do anything that involves increasing the web's reach in the OS.

Anyway, we shipped Talky.io as a web app that worked in

Chrome and FireFox, and eventually @hjon built an iOS app for it too.

But the thing that blew my mind was one day I just downloaded Chrome on Android, opened it to Talky.io and sure enough… IT JUST FRIGGIN' WORKED!

Since then I've been paying much closer attention to what's happening in mobile Chrome and it's very impressive.

Meanwhile on Android

During the last couple of years, a few very bright, persistent, and idealistic developers (many of them at Google) who believed in the web have been at work pushing for, and implementing new web standards that fill the gaps between native and web.

Incredibly cool new stuff is coming, like:

- WebBluetooth (yup, talking to Bluetooth devices from JS on a webpage).
- WebNFC is coming too, apparently.

These things are going to blow the roof off IoT stuff (but that's a whole other blog post).

Just type `chrome://flags` in the URL bar of Chrome for Android and read through all the stuff that's currently in the works. It's amazing!

Anyway, in the past couple of years these fine folks have built a feature that has me more excited than I've been by any web tech for a loooooong time:

ServiceWorker and the concept of Progressive Web Apps.

I believe that the introduction of ServiceWorker and Progressive Web Apps on Android is the most important thing to happen to the mobile web since Steve first introduced the iPhone.

Why?! Because, for the first time, we have a mobile platform with a huge user base, which lets me build a web app that is treated as a first-class citizen by the platform!

(Note: yes, I'm aware there have been other attempts to do this, but none of those had 1.4 billion active users.)

These folks *finally gave us a platform where web apps were first-class citizens*!

And to be clear, I'm not just talking about a way to put a glorified bookmark on the home screen.

I'm talking about a way for us to build web apps that are *indistinguishable* from native apps.

The term that's sticking for these types of apps is "Progressive Web Apps".

In fact, I think Progressive Web Apps (PWAs) actually have a huge leg-up on native apps because you can start using them immediately. You don't have to jump to an app store and wait a minute or two until some huge binary is done downloading. They're just web apps, they have URLs, and they can be built to load super fast. Because… well, we've been optimizing load time performance on the web for a long time.

There's just *so much less friction* for users to start using them. Just think what that would do to your conversion numbers!

Because of the improved on-boarding experience I believe that businesses targeting Android users should be strongly questioning whether they should be building native Android apps at all.

So what are Progressive Web Apps anyway?

Unfortunately, for some reason Google has managed to teach a generation of devs the words "Polymer" and "Angular", while the vast majority of web developers that I meet and talk to today have still have *ZERO* idea what ServiceWorkers or Progressive Web Apps are.

Some of this is because of the newness of it all, and some of this is improving recently. Butsheesh… I hope this changes.

You can think of a progressive web app like this:

It's an app written in HTML, CSS, and JS that can *completely masquerade as a native app*.

This includes:

1. Living on the home screen.
2. Existing in the Android "app switcher" as a separate app (not as part of the browser app).
3. True offline behavior, meaning when you tap the app icon,it will open regardless of current Internet status.
4. The ability to run in the background and triggering OS-level notifications, even when the app and browser is closed.

Instead of starting as a useless web page with a "please install our app" banner, these apps start life running as a tab in your browser. Then *progressively* they become more installed/integrated into the OS.

At first, it's really no different than any other website you visit. But then if you return to that same website/app in your browser again, the browser itself will subtly ask the user if they'd

like to add it to their home screen.

From this moment on it's indistinguishable from a native app to the user.

Also, if you build these correctly there's usually nothing else the user has to download or wait for at all. This means that adding it to the home screen is *effectively an instant app install*. Again, imagine what that'll do to your conversions? Eh? (No, I'm not Canadian)

Luckily, we don't have to entirely guess about the business impact. We actually have some real data from a certain $20 billion dollar online retailer in India called FlipKart, who did launch a PWA and have shared some of their numbers.

Key highlights from FlipKart's experience:

- 40% returning visitors week over week.
- +63% conversions from Home screen visits.
- 3x time spent on FlipKart Lite

That data came from Alex Russel's recent Fluent Keynote on what's next for mobile. I encourage you to watch and share it with product managers and leaders at your company. It does a great job of explaining the how/why of Progressive Web Apps.

For related reading check out:

- Addy Osmani's Getting started with Progressive Web Apps
- Mozilla's service worker examples at: ServiceWorke.rs
- FlipKart's original technical post about their PWA
- Jake Archibald's Offline Cookbook
- Aditya Punjani's post on how they built FlipKart lite

So what does this all mean for us?

We, as web developers, can finally build screaming fast, fully offline-able, and user-privacy-protecting apps that work cross-platform without the need for any friggin' App Store taxes, approval processes, or doorslamming users up front with "please install my app to use this service".

What about iOS support?

Well, the beauty of it is an iOS user can still use your web app even if service worker support doesn't exist.

They just don't get the extra goodies, like offline and push notifications.

But you could also bundle the app with Cordova and use the Service Worker plugin, and that would in theory let you use the same code to do those things but bundled up as an iOS app.

Why should I care? React Native exists now and solves the same problem.

Personally, I actually kind of wish tools like React Native didn't exist. Stay with me, let me explain. React Native is an amazing and very impressive tool that lets us use our JS skills to write native iOS apps.

But as I've been saying, I don't think we *should* be building native apps unless we absolutely have to.

The end result of React Native is that because it exists and because it's largely aimed at *web developers*, we now have web devs flocking to build native apps just because they can!

I fear that this undermines our ability to use our collective bargaining power to encourage Apple to implement support for Progressive Web Apps.

To be clear, I completely understand why it was created and I have a lot of respect for the technical achievement it represents, and the developers behind it.

I just don't want us to stop pushing Apple to improve web support.

In summary

So, all this said, these things led me to finally exercising the only voting power I have as a consumer:I took my money and left.

I don't see this as switching to Android; I'm simply switching to the best mobile web app platform available today.

The web is the only truly open platform we've got. It's the closest thing we have to a level playing field.

This is why I'm focusing all my efforts on building Progressive Web Apps, and I hope you'll do the same. ■

Reprinted with permission of the original author. First appeared at joreteg.com.

On asking job candidates to code

BY PHIL CALÇADO

We truly practiced hire for attitude, not for skill.

At DigitalOcean, we are making some changes to the recruitment process of back-end developers. [We simplified the job description]() and the interview process, and added a step that asks the candidate to write us some code. This is an account of my experience on hiring processes and their use of code reviews.

Around 2005, I had interviewed for a management position at a Globo.com, the Internet arm of the largest Latin American media conglomerate. I was coming back to product development from a couple of years as a consultant, and was super excited about the opportunity.

The week before I started, my new boss sends me an email with what would be my first task: hire four people to join the four others already on my team. It was a weird situation; I couldn't even tell what kind of people I would need as I hadn't started working yet!

I began thinking about what kind of people I knew would be perfect for almost any engineering job, the people I would try to poach. I then thought about what particular traits or habits these people shared.

Soon enough, I realised a common pattern: the people I thought were great for almost any engineering jobs would be always reading books. Not just the reference books everybody reads for their daily jobs, but texts on software architecture, object-oriented design, arcane programming languages, and advice on how to get better at their profession.

I had a somewhat popular programming blog in Brazil, and I wrote a job ad as a blog post. I described the job the best I could, and asked people to send me their résumé and a list of the last three books they had read.

The response was excellent. The temporary email address I had set up was full of résumés, and I was able to confirm my hypothesis: the most interesting résumés correlate with the list containing books from the categories above.

At Globo.com, we've hired a large team following this process, and some of us alumni still use something similar in our new organisations.

Fast forward a couple of years, and I'd applied for a job at ThoughtWorks. I had worked for some international companies in the past where I would be required to speak English when interacting with other offices, but this would be the first time I would speak English continuously for more than one hour. Over Skype. And have people assess me on my coding skills. I was freaking out.

Luckily, ThoughtWorks had a process that was a bit different from the usual standards companies had back then.

After a quick recruiter screening, they sent me three options for a *code challenge* (a small problem I could solve in any programming language I would like to use). My code submission would then be used during follow-up interviews, including a pairing session where a ThoughtWorker and I would try to extend my code adding a new feature.

I spent four years at ThoughtWorks and saw this process produce consistently good results over and over. It was also important that it didn't particularly dictate what languages or tools a candidate would have to use. My experience with ThoughtWorks was that I could *either choose an interesting project or an interesting programming language* (I've written Internet-scale web crawlers in Drupal and timesheet software in F#), so we truly practiced [hire for attitude, not for skill]().

After all those years it was time to move on. I've had mostly a good experience at ThoughtWorks, but my last project there

was probably my worst. As I looked around for a new position, I dreaded having to go through whiteboard coding sessions on some useless puzzle — the kind of stuff my friends reported the big Internet companies had them go through.

Once more I was lucky and ended up applying for a position at SoundCloud. Back then it was just a handful of engineers, and just like ThoughtWorks their process started with a recruiter screen and a coding challenge.

Different from ThoughtWorks, though, their code challenge wasn't a puzzle but something closer to the work one would perform there. SoundCloud wanted me to build an uploader, from the user interface down to any back-end layers I thought I'd need.

There was one particularly challenging part: how to implement a real-time-ish progress bar. As brain-dead as I was after eight hours at my project, I found myself thinking about possible solutions whilst on the train from East Croydon back to London, and immediately heading to my favourite late-night coffee shop in Old Street to work on it. Solving the problem was a lot of fun, and if you want to check out what was in vogue in Clojure in 2011 and how one can pretend to know JavaScript by writing Scheme with curly brackets, you can see the code here.

A couple of weeks later, I got the gig and moved to Berlin. As it happens with small organisations, even before my probation period is over, I'd started reviewing code for new candidates. Unfortunately, the experience was quite disappointing.

It turned out that most people would put together a Rails app with more lines on their Gemfile than lines of actual code they wrote. Worse, way too often people would just get an Adobe Flash uploader from a random Flash component website and not even write a single line of code. As pragmatic as a plugin plus minimal glue code might be, this wasn't a good use of the candidate's time or mine – I needed them to write some code we could read before inviting them for more interviews.

And then we came up with a different challenge. This time, we would be a bit stricter. We would continue to not limit whatever programming language the candidate used (just like with ThoughtWorks, in a small startup like SoundCloud was back then, we needed T-Shaped people more than anything), but we would ask them to use a language's standard library, with no third-party libs or frameworks.

To make it more feasible, we didn't ask for a web app but for something all platforms would offer: a socket-server interface and a simple string protocol. To drive it closer to the needs of a company that grew from 10 to 100 million users in less than one year, we also included the requirement for clients to handle hundreds of clients at once.

But we've also done something else. Something that would improve the candidate's experience by making sure that their code fulfils the functional requirements before they expose it to us. Something that has saved us a lot of time by not having to review code that obviously didn't even work.

Within the problem description sent to candidates, we included a functional test suíte. It consisted of a binary that when started would try to connect to the candidate's server implementation, open lots of sockets, sends lots of messages, and verify the results against what the

I dreaded having to go through whiteboard coding sessions on some useless puzzle.

Flávio Brasil ... not only found a bug in our test harness, but decompiled the (Scala!) code, wrote a patch and submitted it as part of his solution to the challenge.

problem description stated. The candidate was instructed only to send their submission once it passed the functional test on their local box.

The best submission ever was definitely from Flávio Brasil, now at Twitter, who not only found a bug in our test harness but decompiled the (Scala!) code, wrote a patch and submitted it as part of his solution to the challenge. We received code submissions written in every language under the sun. We had so much qualitative data, I even gave a talk on some particular problems we found in code submissions using Node.js.

For this iteration of our recruitment process at DigitalOcean, I am trying to use as leverage these past experiences as much as possible. Overall the code challenge is very similar to the one we developed at SoundCloud, but there are some stark differences.

The first major change is that we are trying a simplified interviewing process, closer to ThoughtWorks'. Previously we had some ad-hoc whiteboarding or pairing sessions. This time, we will be giving the candidate an opportunity to talk about the code they wrote, instead of some generic question already catalogued in a best-selling book.

Another change is that we give more emphasis to *path-to-production*, or in how the application is tested, built, and executed. Irrespective of how product-centric we are, DigitalOcean is an infrastructure company and we break the fourth wall all the time during application development. Showing interest in build and runtime tools is a good indicator of culture fit.

But the most significant change is how the code is sent to review. In previous jobs, as a reviewer, I would receive the code and the candidate's résumé. Over the past few years, the industry and academia have built compelling evidence that our prejudice influences our decisions when it comes to deciding what good or bad code is. To help keep our bias at bay, we are asking candidates not to add any personally identifiable information to the submission.

We will let reviewers know roughly at which level the candidate is and how many years of experience they have, but we will not disclose gender, name, nationality, geographic location, or which schools or companies they have been at. This required a lot of work from our incredibly awesome recruiting team, and a lot of patience from our reviewers, but preliminary results were quite good.

I am super excited about this first iteration of the new process, and more than just finding the right people to grow our team, I hope we have enough data to share with the industry at some point in the near future. ■

Reprinted with permission of the original author. First appeared at philcalcado.com.

I tried to virtually stalk Mark Zuckerberg

By ALEX KRAS

Part 1 – A naive dream

The dream
In late 2015, I finished reading Automate the Boring Stuff with Python and was very inspired to try to automate something in my life.

At the same time, I have always been fascinated by Mark Zuckerberg – the Bill Gates of our time. A lot of people love to hate on Mark, but I actually like the guy. Sure he got lucky in life, but at the same time he is doing something right for Facebook to continue to stay successful.

In any case, one day I had a "brilliant" idea. *What if I wrote a script that would follow Mark's public posts and send me a text whenever he posted something new? Then I can be the first guy to comment on his post.*

After a while Mark would notice my comments and would begin to wonder "Who is this guy that is always posting meaningful responses to all of my posts?" Then he would invite my wife and I to his house for dinner and our kids will become friends. :)

So, without further ado I got to work.

I've briefly considered using Facebook APIs to get notified on Mark's posts. I've had mixed experience with APIs in the past, hitting rate limits pretty quickly and other related problems. Plus, I wanted to use my Automate the Boring Stuff with Python knowledge.

So I went the other route and wrote a Selenium script (which was really easy to do using selenium module in Python) that would:

1. Log in to Facebook.
2. Use current timestamp as the time of last post.
3. Keep checking every 60 seconds if Mark has posted a new post.
4. Send me a Text using Twilio API, with a link to the new post.

I happen to own a small server, so I set the script to run indefinitely in a headless browser (PhantomJS) and began to wait.

What if I wrote a script that would follow Mark's public posts and send me a text whenever he posted something new?

Paradise lost

It took a couple of days for Mark to post something and I began to get worried that my script did not work.

At some point I had to go to the post office. Afterwards, I drove back home, parked my car, checked my phone and saw a new SMS text from my script. My heart started to beat really fast and I rushed to open the link. I soon realized that the post took place 5 minutes ago and I'd missed the notification when I was driving. By now the post already had thousands of comments…

Oh well, I thought, there is always next time. Sure enough, within a day I had another text. This time it was within 1 minute from the original post. I quickly opened the link, only to discover that Mark's post already had close to 100 comments.

Now don't get me wrong, I am not stupid. I knew that Mark's posts were popular and would get a lot of comments.

I even tried to estimate, the rate at which people were posting replies. I've looked through Mark's older posts and saw some posts with tens of thousands of comments. So if you take 10,000 comments and divide by 24 hours, then divide by 60 minutes, you get about 7 posts per minute.

What I didn't realize in my estimate is that those comments were not evenly distributed in time and that I had a very small chance of being the first to comment.

I knew that I was losing my dream and I considered my options. :)

I could set my script to run more often than every 60 seconds, to give myself an early warning. By doing so I would risk showing up on Facebook's radar as a spammer and it just didn't feel right for me to bombard their servers.

Another option that I considered was to try to make an automated reply, in order to be one of the first people to comment. This approach, however, would defeat the purpose of saying something meaningful and would not help me to become friends with Mark.

I've decided against both of these ideas and admitted my defeat. I've also realized that I could turn this (failed) experiment into an interesting Data Exploration project.

Part 2 –Data analysis

Scraping

Having made a big error in my estimate of the rate at which people were replying to Mark, I was curious to explore when and what people were saying. In order to do that, I needed a data set of comments.

Without putting much thought into it, I decided to scrape one of Mark's most recent posts at the time.

My first approach was to try to modify my notification script to:

1. Log in to Facebook.
2. Go to the post that Mark has made.
3. Click on "Show More Comments" link, until all comments were loaded.
4. Scrape and parse the HTML for comments.

Once again I'd underestimated the scale of the operation. There were just too many comments (over 20,000) and it was too much for a browser to handle. Both Firefox and PhantomJS continued to crash without loading all of the comments.

I had to find another way.

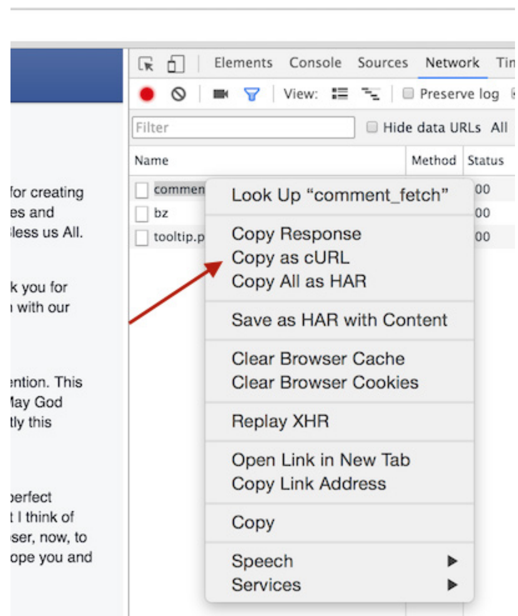
I proceeded to examine how *View more comments* requests were made using the Network Toolbar in Chrome Developer Tools. Chrome allows you to right click on any request and to copy it as CURL via "Copy as cURL" option.

I ran the resulting CURL command in my terminal, and it returned some JSON. BINGO!

At that point all I had to do was to figure out how pagination of comments was done. This turned out to be a simple variable in the query of the request, which acted as a pointer to the next set of comments to fetch.

I've converted the CURL command to a Python request code via this online tool.

After that I wrote a script that would:



1. Start at pagination 0.
2. Make a comments request.
3. Store it in memory.
4. Increment the pagination.
5. Sleep for a random amount of time from 5 to 10 seconds.
6. Repeat the loop, until no more comments were found.
7. Save all of the comments to a JSON file.

I've ended up with an 18Mb minimized JSON file containing about 20,000 comments.

Analyzing the data

First I looked at the distribution of comments over time

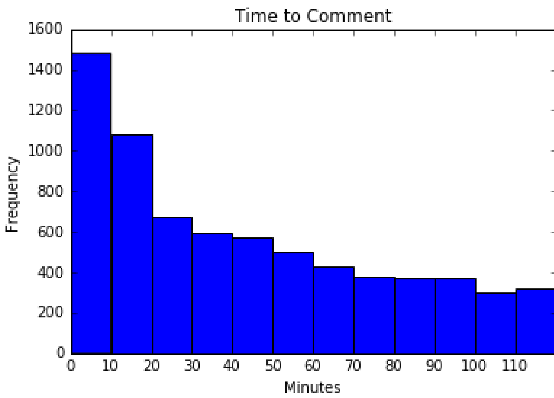
As can be seen in the two plots, it looked a lot like exponential decay, with most of the comments being made in the first two hours.

The first 1,500 comments were made within first 10 minutes. No wonder I had a hard time making it to the top.

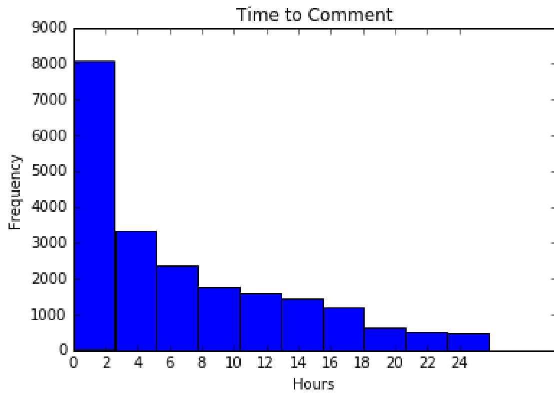
Next I wanted to see what people were saying

I created a word cloud of the most commonly used keywords in comments using a python library called (surprise, surprise) – Word Cloud.

Looking at the word cloud, I realized that I



▲ Number of comments during first 2 hours



▲ Number of comments during first 24 hours

▲ Comment word cloud

might have picked the wrong day to do this experiment. Most of the people responded in kind to Mark's wishes of Merry Christmas and Happy New Year. That was great news for Mark, but kind of boring from the data exploration stand point.

Digging deeper

After I finished the word cloud, I'd spent WAY TOO MUCH TIME trying to gain a deeper understanding of the data.

The data set turned out to be a bit too big for me to iterate on quickly and all of the positive comments created too much noise.

I decided to narrow down the data set by removing all comments with any of the following word stems. A word stem is simply a shortest version of the word that still makes sense. For example, by removing comments that have a word "thank" in it, I was able to remove both the comments with the words "thank you" as well as the comments with the word "thanks." I've used nltk library to help me convert my words to stems.

I organized the stems by a type of comment that they usually belonged to:

Happy New Year Wishes

- new
- happ
- year
- wish
- bless
- congrat
- good luck
- same
- best
- hope
- you too

Comment on Photo of the Family

- photo
- baby
- babi
- beautiful
- pic
- max
- family
- famy
- cute
- child
- love
- nice
- daughter
- sweet

Thanking Mark for creating Facebook
- thank
- connect
- help

After removing all of the typical comments, I ended up with 2887 "unusual" comments.

Digging even deeper

I also recently finished reading Data Smart, from which I learned that Network Analysis can be used to identify various data points that belong together, also known as clusters.

One of the examples in the book used Gephi – an amazing software that makes cluster analysis very easy and fun. I wanted to analyze the "unusual" comments in Gephi, but first I had to find a way to represent them as a Network.

In order to do that, I've:

1. Removed meaningless words such as "and" or "or" (also known as stop words) from every comment using nltk library.
2. Broke remaining words in every comment into an array (list) of word stems.
3. For every comment calculated an intersection with every other comment.
4. Recorder a score for every possible intersection.
5. Removed all intersections with a score of 0.3 or less.
6. Saved all comments as nodes in Gephi graph and every intersection score as an undirected edge.

By now you might be wondering how the intersection score was calculated. You may also wonder what the heck is Gephi graph, but I'll get to it a bit later.

Calculating intersection score

Let say we have two comments:

```
["mark", "love"] # From "Mark, I love you"
# and
["mark", "love", "more"] # From "Mark, I love you
more"
```

We can find the score as follows:

```
def findIntersection(first, second):
    # Find a sub set of words that is present
    # in both lists
    intersection = set(first) & set(second)

    # Words both comments have in common
    intersectionLength = len(intersection)

    # Total length of both comments
    wordCount = len(first) + len(second)

    # Corner case
    if wordCount == 0:
        return 0
    else:
        # Intersection score between two comments
        return (intersectionLength/wordCount)
```

So for our example above:

1. Intersection between two comments is `["mark", "love"]` which is 2 words.
2. Total length of both comments is 5 words.
3. Intersection score is 2/5 = 0.4.

Note: I could have used average length of two comments (so 2+3/2 = 2.5) instead of total length (5), but it would not have made any difference since the score was calculated similarly for all of the comments. So I decided to keep it simple.

Once I had all of intersection calculated, I saved all comments in the nodes.csv file, that had the following format:

```
Id;Label
1;Mark, I love you
```

I've saved all intersection in the edges.csv file, that had the following format:

```
Source;Target;Weight;Type
1;2;0.4;"Undirected"
```

Analyzing the network

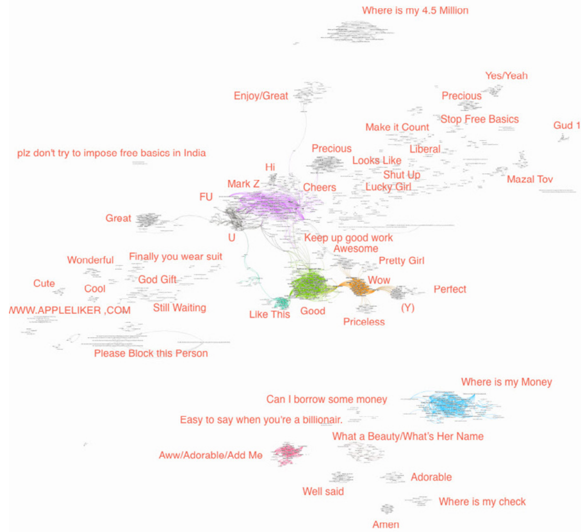
This was all that was needed to import my data into Gephi as a Network Graph. You can read more about Gephi file formats here and this video provides a good introduction to Gephi and how it can be used.

Once I imported my data into Gephi, I ran a network analysis algorithm called "Force Atlas 2", which resulted in the following network graph.

I've manually added the text in red to summarize some of the clusters. If you click on the image, you will be taken to a full screen representation of the graph. *It is pretty big, so you might*

have to zoom out and scroll for a while before you see some data.

Some notes on the results



I was really happy to see my approach finally working (after many days of trying).

I have been staring at those comments for a long time and I've seen some references to "money". Therefore, I was not surprised to see a couple of clusters asking Mark "Where is my Money?"

I was very surprised, however, to see a cluster of comments mentioning a specific number – 4.5 million to be exact. I had no idea where this number was coming from, but a quick Google search pointed me to this hoax. Turns out a lot of people were duped into believing that Mark would give away 4.5 million to 1000 lucky people. All you had to do was to post a "Thank you" message of sorts.

Other than that, I didn't see anything very interesting. There were some spammers and some people asking Mark to ban some people from Facebook. Some aggression towards Mark and a lot more of the general types of comments that I did not filter out.

I've also noticed some weaknesses in my approach. For example, there were two clusters around the word "precious". It was probably caused by removing relationships that did not have intersection score of at least 0.3. Since I did not use the average length for two comments, the threshold of 0.3 really meant that the two comments were at least 60% similar, and it was probably too high and caused the error. On the flip side, it has helped to reduce the number of edges, focusing on the most important connections.

Please let me know in the comments if you find anything else noteworthy or if you have suggestions on how intersection scores can be improved.

Conclusion

It is hard being a celebrity.

I started this journey naively assuming that I can get Mark's attention by simply posting a comment on his timeline. I did not realize the amount of social media attention an average celebrity gets.

It would probably take a dedicated data scientist working fulltime just to get insight into all of the comments that Mark receives. While Mark can afford to hire such a person, my bet is that he is using his resources for more meaningful things.

That being said, this has been a great learning experience for me. Gephi is a magical tool, and I highly recommend checking it out.

If you want some inspiration for automating things, I highly recommend reading Automate the Boring Stuff with Python.

If you are looking for a good entry-level text on data science, I found Data Smart to be an informative read, although hard to follow at times.

Also note that I've destroyed all of my data sets to comply as best as I can with Facebook's Terms of Service. Scraping content without permission is also against Facebook's Terms of Service, but I've avoided thinking about it until after I've done all of my analysis.

I am hoping that Facebook will overlook my transgression, but I wanted to make sure I don't send anybody else down the wrong path without a proper warning.

If all else fails, you can always follow me on Twitter. :) ■

Reprinted with permission of the original author. First appeared at alexkras.com.

food bit *

Ketchup's fishy origins...

Believe it or not, this ubiquitous sauce was once made from pickled fish parts. Ketchup began life as a Southern Chinese condiment made from fish, salt and spices thousands of years ago. The sauce was a hit with 18th century British sailors, who brought it to Europe as a prized delicacy. British cooks soon came up with their own version of ketchup, which included savory ingredients like oysters, mushrooms and nuts. After countless iteration, ketchup eventually evolved into the tomato-based version that we know and love.

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

HACKER BITS is the monthly magazine that gives you the hottest technology and startup stories crowdsourced by the readers of Hacker News. We select from the top voted stories for you and publish them in an easy-to-read magazine format.

Get HACKER BITS delivered to your inbox every month! For more, visit hackerbits.com.