

hacker **bits**

June 2016





new bits

Hello from sunny (finally!) Redmond!

Can it really be June already?! Time does indeed fly...just ask Adrian Kosmaczewski who shows us how to navigate an industry littered with forgotten technologies and has-beens. Find out more in his blast-through-the-past account of “Life as a developer after 40.”

Curious about Progressive Web Apps? Then don't miss this issue's interview with Henrik Joreteg, expert on all things PWA, who gives us the lowdown on this exciting new mobile technology.

As always, our objective at *Hacker Bits* is to help readers like you learn and grow, and that's why we are rolling out a new feature called Spotlight where we get tech experts to reveal their professional secrets.

Lastly, congratulations to the winners of our giveaway!

Time is precious so let's dive into another wonderful issue of *Hacker Bits*!

Peace and plenty of ice cream!

— Maureen and Ray

us@hackerbits.com

content bits

June 2016

6 Fingerprints are usernames, not passwords

8 Am I really a developer or just a good Googler?

10 Develop the three great virtues of a programmer: laziness, impatience, and hubris

16 Being a developer after 40

26 Implementers, solvers, and finders

30 20 lines of code that will beat A/B testing every time

34 Are Progressive Web Apps the future of the Internet?

38 19 tips for everyday git use

48 It takes all kinds

52 When to rewrite from scratch: autopsy of a failed software

56 Clojure, the good parts

contributor bits



Dustin Kirkland

Dustin is an Ubuntu dev and product manager at Canonical. Formerly, CTO of Gazzang, he created an innovative management system for cloud apps. At IBM, Dustin contributed to many Linux security projects and filed 70+ patents. He is the author of 20+ open source projects, including Byobu, eCryptfs, ssh-import-id, and entropy.ubuntu.com. Twitter [@dustinkirkland](https://twitter.com/dustinkirkland).



Scott Hanselman

Scott is a web developer and has blogged at hanselman.com for over a decade. He works in Open Source on ASP.NET and the Azure Cloud for Microsoft out of his home office in Portland, OR. Scott has 3 podcasts, hanselminutes.com, thisdeveloperslife.com and ratchetandthegEEK.com. He's written a number of books and spoken in person to almost 500K devs worldwide.



Reginald Braithwaite

Reg is the author of [JavaScript Allongé](http://JavaScriptAllongé.com), [CoffeeScript Ristretto](http://CoffeeScriptRistretto.com) and raganwald.com. He develops user experiences at PagerDuty. His interests include constructing surreal numbers, deconstructing hopelessly egocentric nulls, and celebrating the joy of programming. His other works are on [GitHub](https://GitHub.com) and [Leanpub](http://Leanpub.com), and you can follow him on Twitter [@raganwald](https://twitter.com/raganwald).



Adrian Kosmaczewski

Adrian is a writer, software developer and teacher. He is the author of two books about mobile software development, and has shipped mobile, web and desktop apps for iOS, Android, Mac OS X, Windows and Linux since 1996. Adrian holds a Masters in Information Technology from the University of Liverpool.



Randall Koutnik

Randall is a Senior UI Engineer at Netflix and holds a lot of strong opinions about Star Wars. He'd love to hear from you via hackerbits@rkoutnik.com.



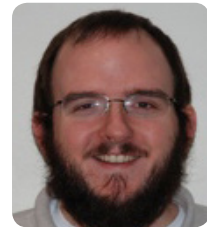
Steve Hanov

Steve can be found at various coffee shops in Waterloo, Ontario, where he writes code and occasionally responds to emails from customers of his web businesses websequencediagrams.com and zwibbler.com. He has three children, one wife, and two birds.



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.



Justin Etheredge

Justin is the cofounder of [Ecstatic Labs](http://EcstaticLabs.com), a small consulting company based out of Richmond, Virginia. His goal is to make software more friendly, one application at a time.



Umer Mansoor

Umer is a software developer, living in San Francisco, CA. He currently works for [Glu Mobile](#) as Platform Manager, building a cloud gaming backend. He previously served as the Head of Software for Starscriber where he built high performance telecommunications software. He blogs at [CodeAhoy.com](#).



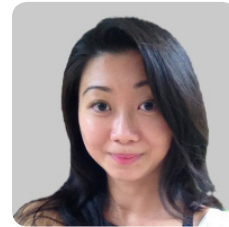
Allen Rohner

Allen is the founder of Rasterize and CircleCI. He is a Clojure contributor, and has commits in `clojure.core`, `contrib`, `lein`, `ring`, `compojure`, `noir`, and about a dozen more libraries. He blogs at [rasterize.io](#) and you can follow him on Twitter [@arohner](#).



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.



Fingerprints are usernames, not passwords

By DUSTIN KIRKLAND

As one of the maintainers of [eCryptfs](#), and a long time Thinkpad owner, I have been asked many times to add support to [eCryptfs](#) for Thinkpad's fingerprint readers.

I actually captured this as a wishlist [bug](#) in Launchpad in August 2008, but upon thinking about it a bit more, I later closed the bug as "won't fix" in [February 2009](#), and discussed in a [blog post](#), saying:

Hi, thanks so much for the bug report. I've been thinking about this quite a bit lately. I'm going to have to mark this "won't fix" for now.

The prevailing opinion from security professionals is that fingerprints are perhaps a good replacement for usernames. However, they're really not a good replacement for passwords.

Consider your laptop... how many fingerprints of yours are there on your laptop right now? As such, it's about as secret as your username. You don't leave your password on your spacebar, or on your beer bottle :-)

*This Wikipedia entry (although it's about Microsoft Fingerprint Readers) is pretty accurate: * http://en.wikipedia.org/wiki/Microsoft_Fingerprint_Reader*

So, I'm sorry, but I don't think we'll be fixing this for now.

I'm bringing this up again to highlight the work released by [The Chaos Computer Club](#), which has [demonstrated how truly insecure](#) Apple's [TouchID](#) is.

There may be civil liberties at issue as well. While this [piece is satire](#), and Apple says that it is not sharing your fingerprints with the government, we've been kept in the dark about such things before. I'll leave you to draw your own conclusions on that one.

But let's just say you're okay with Apple sharing your fingerprints with the NSA, as I've already told you, they're not private at all. You leave them on everything you touch. And let's

you need a password or passphrase. Something that can be independently chosen, changed, and rotated. I will continue to advocate this within the Ubuntu development community, as I have since 2009.

Once your fingerprint is compromised (and, yes, it almost certainly already is, if you've crossed an international border or registered for a driver's license in some US states and countries), how do you change it? Are you starting to see why this is a really bad idea?

There are plenty of inventions that exist, but turned out to be bad ideas. And I think fingerprint readers are another one of those.


Biometrics...cannot authenticate a person or a thing alone.

say you're insistent on using fingerprint (biometric) technology because you can. In that case, your fingerprints might identify you, much as your email address or username identifies you, perhaps from a list.

I could see some value, perhaps, in a tablet that I share with my wife, where each of us have our own accounts, with independent configurations, apps, and settings. We could each conveniently *identify* ourselves by our fingerprint. **But biometrics cannot, and absolutely must not, be used to authenticate an identity.** For authentication,

This isn't a knock on Apple, as Thinkpad have embedded fingerprint readers for nearly a decade. My intention is to help stop and think about the place of biometrics in security. Biometrics can be used as a lightweight, convenient mechanism to establish identity, but they cannot authenticate a person or a thing alone.

So please, if you have any respect for the privacy your data, or your contacts' information, please don't use fingerprints (or biometrics, in general) for authentication. ■



Am I really a developer or just a good Googler?

By SCOTT HANSELMAN

I got a very earnest and well-phrased email from a young person overseas recently.

Some time in my mind sounds come that is that I am really a developer or just a good googler. I don't know what is the answer I am googler or I am developer. Scott Please clear on my mind on this please.

This is a really profound question that deserved an answer. Since I only have so many [keystrokes left](#) in my life, I am blogging my thoughts and emailing a link.

I've felt the same way sometimes when playing a video

game. It'll get hard as I progress through the levels, but not crushingly hard. Each level I squeak by I'll find myself asking, "Did I deserve to pass that level? I'm not sure I could do it again."

You get that feeling like you're in over your head, but just a bit. Just enough that you can feel the water getting into your nose but you're not drowning yet.

First, remember [you are not alone](#). I think that we grow when we are outside our comfort zone. If it's not breaking you down, it's not building you up.

Second, anything that you want to be good at is worth practicing. Do [Code Katas](#). Do a [Project Euler problem](#) every few

weeks, if not weekly.

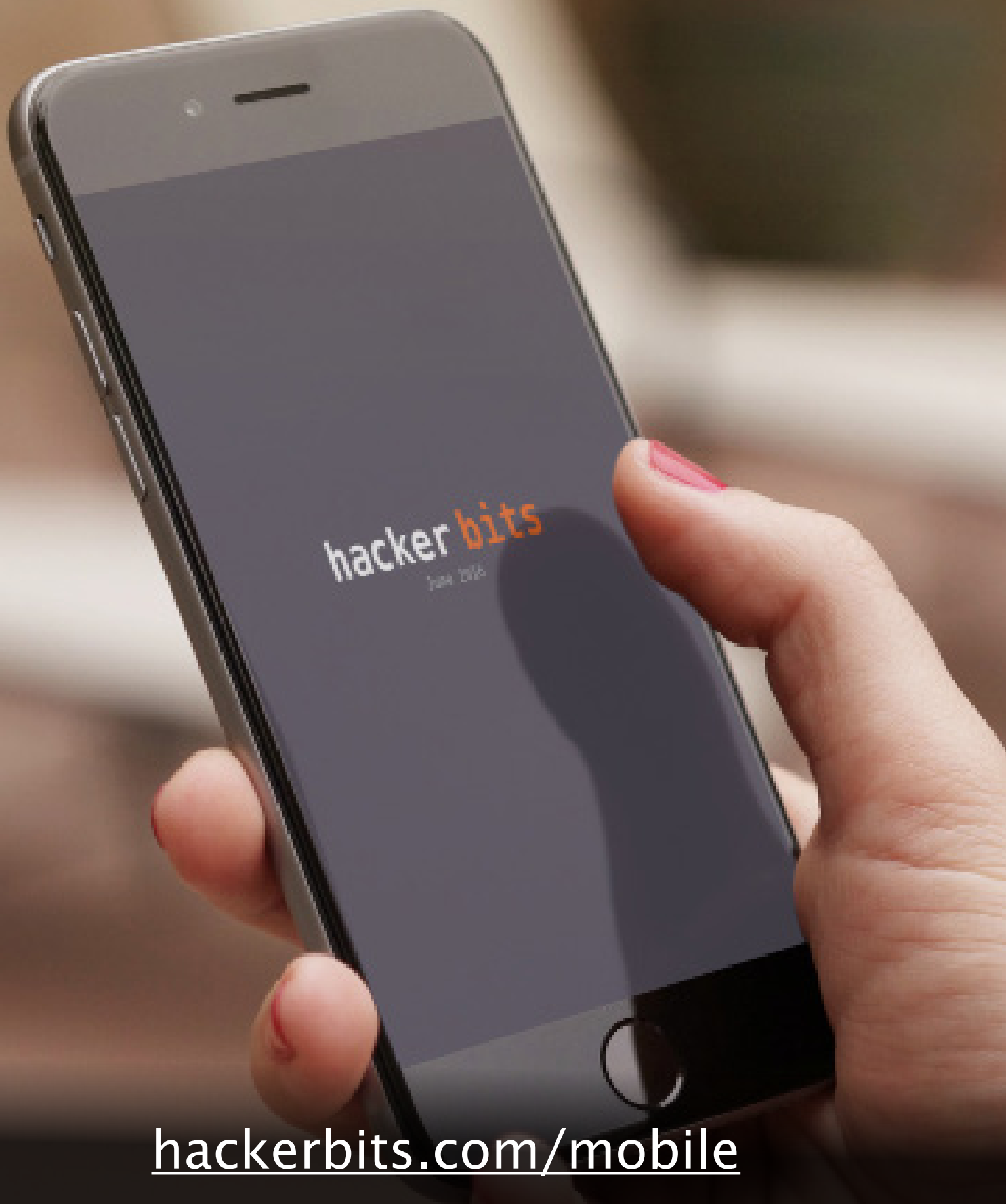
Third, try programming for a day without Googling. Then two days, maybe a week. See how it feels. Remember that there was a time we programmed without copying our work.

Fourth, [think about the problem](#), deeply. Read about algorithms, read [Programming Pearls](#), read about [Design Patterns](#). Rather than copying code from Stack Overflow, copy patterns from the greats.

Fifth, [get involved](#). Go to User Groups, Nerd Dinners, meet with others who feel the same way you do about technology. Stretch.

What do you think? ■

Reprinted with permission of the original author. First appeared at [hanselman.com](#).



hackerbits.com/mobile



Develop the three great virtues of a programmer: laziness, impatience, and hubris

By REGINALD BRAITHWAITE

Laziness and eagerness

In computing, “laziness” is a broad term, generally referring to not doing any work unless you need it. Whereas its opposite is “eagerness,” doing as much work as possible in case you need it later.

Consider this JavaScript:

```
function ifThen (a, b) {  
  if (a) return b;  
}  
  
ifThen(1 === 0, 2 + 3)  
//=> undefined
```

Now, here’s the question: Does JavaScript evaluate $2+3$? You probably know the answer: Yes it does. When it comes to passing arguments to a function invocation, JavaScript is *eager*, it evaluates all of the expressions, and it does so whether the value of the expression is used or not.¹

If JavaScript was *lazy*, it would not evaluate $2+3$ in the expression `ifThen(1 === 0, 2 + 3)`. So is JavaScript an “eager” language? Mostly. But not always! If we write: `1 === 0 ? 2 + 3 : undefined`, JavaScript does *not* evaluate $2+3$. Operators like `?:` and `&&` and `||`, along with program control structures like `if`, are lazy. You just have to know in your head what is eager and what is lazy.

And if you want something to be lazy that isn’t naturally lazy, you have to work around JavaScript’s eagerness. For example:

```
function ifThenEvaluate (a, b) {  
  if (a) return b();  
}  
  
ifThenEvaluate(1 === 0, () => 2 + 3)  
//=> undefined
```

JavaScript eagerly evaluates `() => 2 + 3`, which is a function. But it doesn’t evaluate the expression in the body of the function until it is invoked. And it is not invoked, so $2+3$ is not evaluated.

Wrapping expressions in functions to delay evaluation is a longstanding technique in programming. They are colloquially called [thunks](#), and there are lots of interesting applications for them.

Generating laziness

The bodies of functions are a kind of lazy thing: They aren’t evaluated until you invoke the function. This is related to `if` statements, and every other kind of control flow construct: JavaScript does not evaluate statements unless the code actually encounters the statement.

```
function containing(value, list) {  
  let listContainsValue = false;  
  
  for (const element of list) {  
    if (element === value) {  
      listContainsValue = true;  
    }  
  }  
  
  return listContainsValue;  
}
```

Consider this code:

You are doubtless chuckling at its naïveté. Imagine this list was the numbers from one to a billion – e.g. `[1, 2, 3, ..., 999999998, 999999999, 1000000000]` – and we invoke:

```
const billion = [1, 2, 3, ..., 999999998, 999999999, 1000000000];  
  
containing(1, billion)  
//=> true
```

We get the correct result, but we iterate over every one of our billion numbers first. Awful! Small children and the otherwise febrile know that you can `return` from anywhere in a JavaScript function, and the rest of its evaluation is abandoned. So we can write this:

```
function containing(list, value) {  
  for (const element of list) {  
    if (element === value) {  
      return true;  
    }  
  }  
  
  return false;  
}
```

This version of the function is lazier than the first: It only does the minimum needed to determine whether a particular list contains a particular value.

From `containing`, we can make a similar function, `findWith`:

```
function findWith(predicate, list) {
  for (const element of list) {
    if (predicate(element)) {
      return element;
    }
  }
}
```

`findWith` applies a predicate function to lazily find the first value that evaluates truthily. Unfortunately, while `findWith` is lazy, its argument is evaluated eagerly, as we mentioned above. So let's say we want to find the first number in a list that is greater than 99 and is a palindrome:

```
function isPalindromic(number) {
  const forwards = number.toString();
  const backwards = forwards.split('').reverse().join('');

  return forwards === backwards;
}

function gt(minimum) {
  return (number) => number > minimum;
}

function every(...predicates) {
  return function (value) {
    for (const predicate of predicates) {
      if (!predicate(value)) return false;
    }
    return true;
  };
}

const billion = [1, 2, 3, ..., 999999998, 999999999, 1000000000];

findWith(every(isPalindromic, gt(99)), billion)
//=> 101
```

It's the same principle as before, of course, we iterate through our billion numbers and stop as soon as we get to 101, which is greater than 99 and palindromic.

But JavaScript eagerly evaluates the arguments to `findWith`. So it evaluates `isPalindromic, gt(99)` and binds it to `predicate`, then it eagerly evaluates `billion` and binds it to `list`.

Binding one value to another is cheap. But what if we had to *generate* a billion numbers?

```
function NumbersUpTo(limit) {
  const numbers = [];
  for (let number = 1; number <= limit; ++number) {
    numbers.push(number);
  }
  return numbers;
}

findWith(every(isPalindromic, gt(99)), NumbersUpTo(1000000000))
//=> 101
```

`NumbersUpTo(1000000000)` is eager, so it makes a list of all billion numbers, even though we only need the first 101. This is the problem with laziness: We need to be lazy all the way through a computation.

Luckily, we just finished working with generators² and we know exactly how to make a lazy list of numbers:

```
function * Numbers () {
  let number = 0;
  while (true) {
    yield ++number;
  }
}

findWith(every(isPalindromic, gt(99)), Numbers())
//=> 101
```

Generators yield values lazily. And `findWith` searches lazily, so we can find 101 without first generating an infinite array of numbers. JavaScript still evaluates `Numbers()` eagerly and binds it to `list`, but now it's binding an iterator, not an array. And the `for (const element of list) { ... }` statement lazily takes values from the iterator just as it did from the `billion` array.

The sieve of Eratosthenes

We start with a table of numbers (e.g., 2, 3, 4, 5, . . .) and progressively cross off numbers in the table until the only numbers left are primes. Specifically, we begin with the first number, *p*, in the table, and:

1. Declare *p* to be prime, and cross off all the multiples of that number in the table, starting from *p* squared, then;
2. Find the next number in the table after *p* that is not yet crossed off and set *p* to that number; and then repeat from step 1.

Here is the [sieve of Eratosthenes](#), written in eager style:

```
function compact (list) {
  const compacted = [];

  for (const element of list) {
    if (element != null) {
      compacted.push(element);
    }
  }

  return compacted;
}

function PrimesUpTo (limit) {
  const numbers = NumbersUpTo(limit);

  numbers[0] = undefined; // '1' is not a prime
  for (let i = 1; i <= Math.ceil(Math.sqrt(limit)); ++i) {
    if (numbers[i]) {
      const prime = numbers[i];

      for (let ii = i + prime; ii < limit; ii += prime) {
        numbers[ii] = undefined;
      }
    }
  }

  return compact(numbers);
}

PrimesUpTo(100)
//=> [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89]
```

Let's take a pass at writing the sieve of Eratosthenes in lazy style. First off, a few handy things we've already seen in this blog, and in [JavaScript Allongé](#):

```
function * range (from = 0, to = null) {
  let number = from;

  if (to == null) {
    while (true) {
      yield number++;
    }
  } else {
    while (from <= to) {
      yield number++;
    }
  }
}

function * take (numberToTake, iterable) {
  const iterator = iterable[Symbol.iterator]();

  for (let i = 0; i < numberToTake; ++i) {
    const { done, value } = iterator.next();
    if (!done) yield value;
  }
}
```

With those in hand, we can write a generator that maps an iterable to a sequence of values with every *n*th element changed to `null`:³

```
function * nullEveryNth (skipFirst, n, iterable) {
  const iterator = iterable[Symbol.iterator]();

  yield * take(skipFirst, iterator);

  while (true) {
    yield * take(n - 1, iterator);
    iterator.next();
    yield null;
  }
}
```

That's the core of the “sieving” behaviour: Take the front element of the list of numbers, call it *n*, and sieve every *n*th element afterwards.

Now we can apply `nullEveryNth` recursively: Take the first unsieved number from the front of the list, sieve its multiples out, and yield the results of sieving what remains:

```
function * sieve (iterable) {
  const iterator = iterable[Symbol.iterator]();
  let n;

  do {
    const { value } = iterator.next();

    n = value;
    yield n;
  } while (n == null);

  yield * sieve(nullEveryNth(n * (n - 2), n, iterator));
}
```

With `sieve` in hand, we can use `range` to get a list of numbers from 2, sieve those recursively, then we `compact` the result to filter out all the `null`s, and what is left are the primes:

```
const Primes = compact(sieve(range(2)));
```

Besides performance, did you spot the full-on bug? Try running it yourself, it won't work! The problem is that at the last step, we called `compact`, and `compact` is an eager function, not a lazy one. So we end up trying to build an infinite list of primes before filtering out the `null`s.

We need to write a lazy version of `compact`:

```
function * compact (list) {
  for (const element of list) {
    if (element != null) {
      yield element;
    }
  }
}
```

And now it works!

```
take(100, Primes())
//=>
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431,
433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
499, 503, 509, 521, 523, 541]
```

When we write things in lazy style, we need lazy versions of all of our usual operations. For example, here's an eager implementation of `unique`:

```
function unique (list) {
  const orderedValues = [];
  const uniqueValues = new Set();

  for (const element of list) {
    if (!uniqueValues.has(element)) {
      uniqueValues.add(element);
      orderedValues.push(element);
    }
  }
  return orderedValues;
}
```

Naturally, we'd need a lazy implementation if we wanted to find the unique values of lazy iterators:

```
function * unique (iterable) {
  const uniqueValues = new Set();

  for (const element of iterable) {
    if (!uniqueValues.has(element)) {
      uniqueValues.add(element);
      yield element;
    }
  }
}
```

And so it goes with all of our existing operations that we use with lists: We need lazy versions we can use with iterables, and we have to use the lazy operations throughout: We can't mix them.

It comes down to types

This brings us to an unexpected revelation.

Generators and laziness can be wonderful. Exciting things are happening with using generators to emulate synchronized code with asynchronous operations, for example. But as we've seen, if we want to write lazy code, we have to be careful to be consistently lazy. If we accidentally mix lazy and eager code, we have problems.

This is a [symmetry](#) problem. And at a deeper level, it exposes a problem with the “duck typing” mindset: There is a general idea that as long as objects handle the correct interface – as long as they respond to the right methods – they are interchangeable.

But this is not always the case. The eager and lazy versions of `compact` both quack like ducks that operate on lists, but one is lazy and the other is not. “Duck typing” does not and cannot capture difference between a function that assures laziness and another that assures eagerness.

Many other things work this way, for example escaped and unescaped strings. Or obfuscated and native IDs. To distinguish between things that have the same interfaces, but also have semantic or other contractual differences, we need *types*.

We need to ensure that our programs work with each of the types, using the correct operations, even if the incorrect operations are also “duck compatible” and appear to work at first glance. ■

The full source

```
1 function * nullEveryNth (skipFirst, n, iterable) {
2   const iterator = iterable[Symbol.iterator]();
3
4   yield * take(skipFirst, iterator);
5
6   while (true) {
7     yield * take(n - 1, iterator);
8     iterator.next();
9     yield null;
10  }
11 }
12
13 function * sieve (iterable) {
14   const iterator = iterable[Symbol.iterator]();
15   let n;
16
17   do {
18     const { value } = iterator.next();
19
20     n = value;
21     yield n;
22   } while (n == null);
23
24   yield * sieve(nullEveryNth(n * (n - 2), n, iterator));
25 }
26
27 const Primes = compact(sieve(range(2)));
28
29 // General-Purpose Lazy Operations
30
31 function * range (from = 0, to = null) {
32   let number = from;
33
34   if (to == null) {
35     while (true) {
36       yield number++;
37     }
38   }
39   else {
40     while (number <= to) {
41       yield number++;
42     }
43   }
44 }
45
46 function * take (numberToTake, iterable) {
47   const iterator = iterable[Symbol.iterator]();
48
49   for (let i = 0; i < numberToTake; ++i) {
50     const { done, value } = iterator.next();
51     if (!done) yield value;
52   }
53 }
54
55 function * compact (list) {
56   for (const element of list) {
57     if (element != null) {
58       yield element;
59     }
60   }
61 }
```

Notes

¹ A few people have pointed out that a [sufficiently smart compiler](#) can notice that $2+3$ involves two constants and a fixed operator, and therefore it can be compiled to 5 in advance. JavaScript does not *necessarily* perform this optimization, but if it did, we could substitute something like $x + y$ and get to the same place in the essay.

² “[Programs must be written for people to read, and only incidentally for machines to execute.](#)”

³ This is the simplest and most naïve implementation that is recognizably identical to the written description. In [The Genuine Sieve of Eratosthenes](#), Melissa E. O’Neill describes how to write a lazy functional sieve that is much faster than this implementation, although it abstracts away the notion of crossing off multiples from a list.

Reprinted with permission of the original author and [raganwald.com](#).

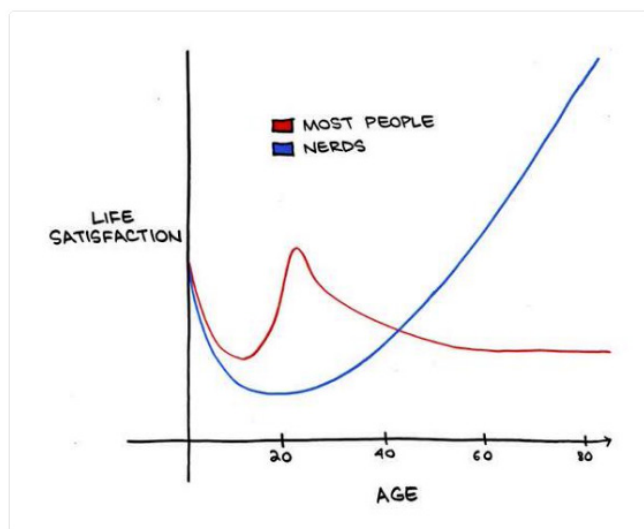
Being a developer after 40

By ADRIAN KOSMACZEWSKI

Hi everyone, I am a forty-two year old self-taught developer, and this is my story.

A couple of weeks ago I came by the tweet below, and it made me think about my career, and those thoughts brought me back to where it all began for me (see figure).

I started my career as a software developer at precisely 10am, on Monday October 6th, 1997, somewhere in the city of [Olivos](#), just north of [Buenos Aires, Argentina](#). The moment was Unix Epoch [876142800](#). I had recently celebrated my 24th birthday.



RETWEETS 2,952 LIKES 3,534



11:09 PM - 17 Mar 2015

The world in 1997

The world was a slightly different place back then.

Websites did not have cookie warnings. The future of the web was portals like [Excite.com](#). AltaVista was my preferred search engine. My e-mail was kosmacze@sc2a.unige.ch, which meant that my first personal website was located in <http://sc2a.unige.ch/~kosmacze>.

We were still mourning [Lady Diana](#). Steve Jobs had taken the role of CEO and convinced Microsoft to inject [\\$150 million](#) into Apple Computer. Digital Equipment Corporation was suing Dell. The remains of Che Guevara had just been brought back to Cuba. The [fourth season of "Friends"](#) had just started. [Gianni Versace](#) had just been murdered in front of his house. [Mother Teresa](#), [Roy Lichtenstein](#) and [Jeanne Calment](#) (the world's oldest person ever) had just passed away. People were playing [Final Fantasy 7](#) on their [PlayStation](#) like crazy. BBC 2 started broadcasting the [Teletubbies](#). James Cameron was about to release [Titanic](#). The Verve had just released their hit "[Bitter Sweet Symphony](#)" and then had to pay most of the royalties to the Rolling Stones.

Smartphones looked like the [Nokia 9000 Communicator](#); they had 8 MB of memory, a 24 MHz i386 CPU and run the GEOS operating system.

Smartwatches looked like the [CASIO G-SHOCK DW-9100B](#). Not as many apps but the battery life was much longer.

IBM Deep Blue [had defeated Garry Kasparov for the first time](#) in a game of chess.

A hacker known as "_eci" published the C code for a Windows 3.1, 95 and NT ex-

plot called "[WinNuke](#)," a denial-of-service attack on TCP port 139 (NetBIOS) that causes a Blue Screen of Death.

Incidentally, 1997 is also the year [Malala Yousafzai](#), [Chloë Grace Moretz](#) and [Kylie Jenner](#) were born.

Many film storylines take place in 1997, to name a few: [Escape from New York](#), [Predator 2](#), [The Curious Case of Benjamin Button](#), [Harry Potter and the Half-Blood Prince](#), [The Godfather III](#) and according to [Terminator 2: Judgement Day](#), Skynet became self-aware at 2:14 am on August 29, 1997. That did not happen; however, in an interesting turn of events, the domain google.com was registered on September 15th that year.

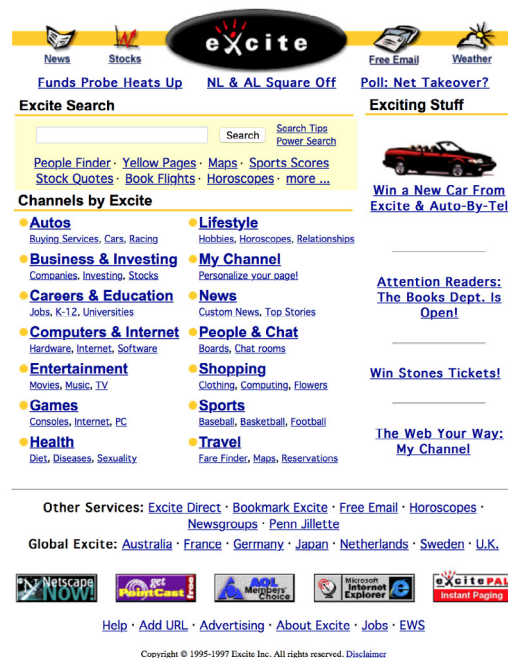
We were two years away from [Y2K](#) and the media was starting to get people nervous about it.

My first developer job

My first job consisted of writing ASP pages in various editors, ranging from [Microsoft FrontPage](#), to [HotMeTaL Pro](#) to [Edit-Plus](#), managing cross-browser compatibility between Netscape Navigator and Internet Explorer 4, and writing stored procedures in [SQL Server 6.5](#) powering a commercial website published in Japanese, Russian, English and Spanish—without any [consistent UTF-8 support](#) across the software stack.

The product of these efforts ran in a [Pentium II](#) server hosted somewhere in the USA, with a stunning 2 GB hard disk drive and a whopping 256 MB of RAM. It was a single server running [Windows NT 4](#), [SQL Server 6.5](#) and [IIS 2.0](#), serving around ten thousand visitors per day.

My first professional programming language was this mutant called [VBScript](#), and of



▲ Excite in 1997, courtesy of the Internet Archive

course a little bit of JavaScript on the client side, sprinkled with lots of “if this is Netscape do this, else do that” because back then I had no idea how to use JavaScript properly.

Interestingly, it’s 2016 and we are [barely starting](#) to understand how to do anything in JavaScript.

Unit tests were unheard of. The [Agile Manifesto](#) had not been written yet. Continuous integration was a dream. XML was not even a buzzword. Our QA strategy consisted of restarting the server once a week, because otherwise it would crash randomly. We developed our own [COM+](#) component in [Visual J++](#) to parse JPEG files uploaded to the server. As soon as [JPEG 2000](#)-encoded files started popping up, our component broke miserably.

We did not use source control, not even [CVS](#), [RCS](#) or, God forbid, [SourceSafe](#). [Subversion](#) did not exist yet. Our [Joel Test](#) score was minus 25.

6,776 days

For the past 6,776 days, I had a cup of coffee in the morning and wrote code with things named VBScript, JavaScript, Linux, SQL, HTML, Makefiles, Node.js, CSS, XML, .NET, YAML, Podfiles, JSON, Markdown, PHP, Windows, Doxygen, C#, Visual Basic, Visual Basic.NET, Java, Socket.io, Ruby, unit tests, Python, shell scripts, C++, Objective-C, batch files, and lately Swift.

In those 6776 days, lots of things happened; most importantly, my wife and I got married. I quit 6 jobs and was fired twice. I started and closed my own business. I finished my Master’s degree. I published a few

open source projects, and one of them landed me an article on [Ars Technica by Erica Sadun herself](#). I was featured in Swiss and Bolivian TV shows. I watched live keynotes by Bill Gates and by Steve Jobs in Seattle and San Francisco. I [spoke at](#) and co-organised conferences in four continents. I wrote and published [two books](#). I burned out twice (not the books, myself,) and lots of other things happened, both wonderful and horrible.

I have often pondered about leaving the profession altogether. But somehow, code always calls me back after a while. I like to write apps, systems and software. To avoid burning out, I have had to develop strategies.

In this talk I will give you my secrets, so that you too can reach the glorious age of 40 as an experienced developer, willing to continue in this profession.

Advice for the young at heart

Some simple tips to reach the glorious age of 40 as a happy software developer:

1. Forget the hype

The first advice I can give you all is do not pay attention to hype. Every year there is a new programming language, framework, library, pattern, component architecture or paradigm that takes the blogosphere by storm. People get crazy about it. Conferences are given. Books are written. [Gartner hype cycles](#) rise and fall. Consultants charge insane amounts of money to teach, deploy or otherwise fuck up the lives of people in this industry. The press will support these horrors and will make

you feel guilty if you do not pay attention to them.

- In 1997 it was [CORBA](#) & [RUP](#).
- In 2000 it was [SOAP](#) & XML.
- In 2003 it was [Model Driven Architecture](#) and [Software Factories](#).
- In 2006 it was [Semantic Web](#) and [OLPC](#).
- In 2009 it was [Augmented Reality](#).
- In 2012 it was [Big Data](#).
- In 2015...virtual reality? Bots?

Do not worry about hype.

Keep doing your thing, keep learning what you were learning, and move on. Pay attention to it only if you have a genuine interest, or if you feel that it could bring you some benefit in the medium or long run.

The reason for this lies in the fact that as the Romans said in the past, *nihil novi sub sole* (nothing new under the sun). Most of what you see and learn in computer science has been around for decades, and this fact is purposely hidden beneath piles of marketing, books, blog posts and questions on Stack Overflow. Every new architecture is just a re-imagination and a re-adaptation of an idea that has been floating around for decades.

2. Choose your galaxy wisely

In our industry, every technology generates what I call a “galaxy.” These galaxies feature stars but also black holes; meteoric changes that fade in the night; many planets, only a tiny fraction of which harbour some kind of life; and lots of cosmic dust and dark matter.

Examples of galaxies are .NET, Cocoa, Node.js, PHP, Emacs, SAP, etc. Each of these

features evangelists, developers, bloggers, podcasts, conferences, books, training courses, consulting services, and inclusion problems. Galaxies are built on the assumption that their underlying technology is the answer to all problems. Each galaxy, thus, is based on a wrong assumption.

The developers from those different galaxies embody the prototypical attitudes that have brought that technology to life. They adhere to the ideas, and will enthusiastically wear the t-shirts and evangelize others about the merits of their choice.

Actually, I use the term “galaxy” to avoid the slightly more appropriate if not less controversial term “religion,” which might describe this phenomenon better.

In my personal case, I spent the first ten years of my career in the Microsoft galaxy, and the following nine in the Apple galaxy.

I dare say, one of the biggest reasons why I changed galaxies was Steve Ballmer. I got tired of the general attitude of the Microsoft galaxy people against open source software.

On the other hand, I also have to say that the Apple galaxy is a delightful place, full of artists, musicians and writers who, by chance or ill luck, happen to write software as well.

I attended conferences in the Microsoft galaxy, like the Barcelona TechEd 2003, and various Tech Talks in Buenos Aires, Geneva or London. I even spoke at the Microsoft DevDays 2006 in Geneva. The general attitude of developers in the Microsoft galaxy is unfriendly, “corporate” and bound in secrecy, NDAs and cumbersome IT processes.

The Apple galaxy was to me, back in 2006, exactly the

opposite; it was full of people who were musicians, artists and painters; they would write software to support their passion, and they would write software with passion. It made all the difference, and to this day, I still enjoy tremendously this galaxy, the one we are in, right now, and that has brought us all together.

And then the iPhone came out, and the rest is history.

So my recommendation to you is: choose your galaxy wisely, enjoy it as much or as little as you want, but keep your telescope pointed towards other galaxies, and prepare to make a hyperjump to other places if needed.

3. Learn about software history

This takes me to the next point: learn how your favorite technology came to be. Do you like C#? Do you know who created it? How did the .NET project come to be? Who was the lead architect? Which were the constraints of the project and why did the language turned out to be what it is now?

Apply the same recipe to any language or CPU architecture that you enjoy or love: Python, Ruby, Java, whatever the programming language; learn their origins, how they came to be. The same goes for operating systems, networking technologies, hardware, anything. Go and learn how people came up with those ideas, and how long they took to grow and mature. Because [good software takes ten years](#), you know. (see Figure)

The stories surrounding the genesis of our industry are fascinating, and will show you two things: first, that [everything is a remix](#). Second, that you could

be the one remixing the next big thing. No, scratch that: you *are going to be* the creator of the next big thing.

And to help you get there, here is my (highly biased) selection of history books that I like and recommend:

- [Dealers of Lightning](#) by Michael A. Hiltzik
- [Revolution in the Valley](#) by Andy Hertzfeld
- [The Cathedral and the Bazaar](#) by Eric S. Raymond
- [The Success of Open Source](#) by Steven Weber
- [The Old New Thing](#) by Raymond Chen
- [The Mythical Man Month](#) by Frederick P. Brooks Jr.

You will also learn to value those things that stood the test of time: [Lisp](#), [TeX](#), [Unix](#), [bash](#), [C](#), [Cocoa](#), [Emacs](#), [Vim](#), [Python](#), [ARM](#), [GNU make](#), [man pages](#). These are some examples of long-lasting useful things that are something to celebrate, cherish and learn from.

4. Keep on learning

Learn. Anything will do. Wanna learn Fortran? Go for it. Find Erlang interesting? Excellent. Think COBOL might be the next big thing in your career? Fantastic. Need to know more about [Functional Reactive Programming](#)? Be my guest. Design? Of course. UX? You must. Poetry? [You should](#).

Many common concepts in Computer Science have been around for decades, which makes it worthwhile to learn old programming languages and frameworks; even “arcane” ones. First, it will make you appreciate the current state of the industry

(or hate it, it depends) and second, you will learn how to use the current tools more effectively — if anything, because you will understand its legacy and origins.

Tip 1: *learn at least one new programming language every year*. I did not come up with this idea; The Pragmatic Programmer book did. And it works.

One new programming language every year. Simple, huh? Go beyond the typical “Hello, World” stage, and build something useful with it. I usually build a simple calculator with whatever new technology I learn. It helps me figure out the syntax, it makes me familiar with the APIs or the IDE, etc.

Tip 2: *read at least 6 books per year*. I have shown above a list of six must-read books; that should keep you busy for a year. Here goes the list for the second year:

- [Peopleware](#) by Tom DeMarco and Tim Lister
- [The Psychology of Software Programming](#) by Gerald M. Weinberg
- [Facts and Fallacies of Software Engineering](#) by Robert L. Glass
- [The Design of Everyday Things](#) by Don Norman
- [Agile!: The Good, the Hype and the Ugly](#) by Bertrand Meyer
- [Rework](#) by Jason Fried and David Heinemeier Hansson
- [Geekonomics](#) by David Rice

(OK, those are seven books.)

Six books per year looks like a lot, but it only means one every 2 months. And most of the books I have mentioned in this presentation are not that long, and even better, they are out-

standingly well written, they are fun and are full of insight.

Look at it this way: if you are now 20 years old, by the age of 30 you will have read over 60 books, and over 120 when you reach my age. And you will have played with at least 20 different programming languages. Think about it for a second.

Some of the twelve books I’ve selected for you have been written in the seventies, others in the eighties, some in the nineties and finally most of them are from the past decade. They represent the best writing I have come across in our industry.

But do not just read them; take notes. Bookmark. Write on the pages of the books. Then re-read them every so often. [Borges](#) used to say that a bigger pleasure than reading a book is re-reading it. And also, please, buy those books you really like in paper format. Believe me. eBooks are overrated. Nothing beats the real thing.

Of course, please know that

as you will grow old, the number of things that qualify as new and/or important will drop dramatically. Prepare for this. It is OK to weep silently when you realise this.

5. Teach

Once you have learnt, *teach*. This is very important.

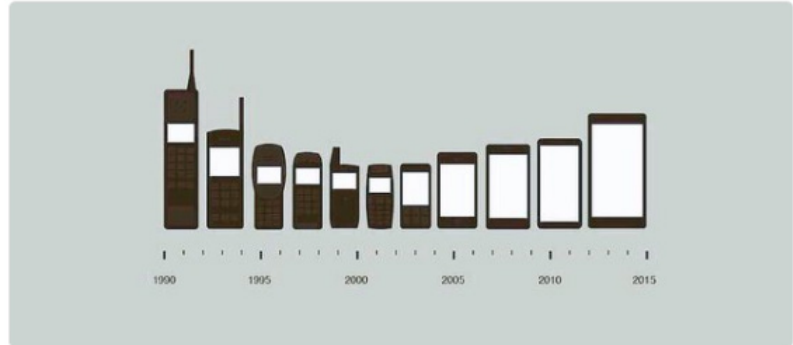
This does not mean that you should set up a classroom and invite people to hear your ramblings (although it would be awesome if you did!) It might mean that you give meaningful answers to questions in Stack Overflow; that you write a book; that you publish a podcast about your favorite technology; that you keep a blog; that you write on Medium; that you go to another continent and set up programming schools using Raspberry Pis; or that you help a younger developer by becoming their mentor (do not do this before the age of 30, though.)

Teaching will make you more humble, because it will



JM Alvarez-Pallete
@jmalvpal

The mobile phone evolution @ValaAfshar



RETWEETS
620

LIKES
420



11:34 PM - 25 Nov 2015

painfully show you how limited your knowledge is. *Teaching is the best way to learn.* Only by testing your knowledge against others are you going to learn properly. This will also make you more respectful regarding other developers and other technologies; every language, no matter how humble or arcane, has its place within the [Tao of Programming](#), and only through teaching will you be able to feel it.

And through teaching you can really, really make a difference in this world. Back in 2012 I received a mail from a person who had attended one of my trainings. She used to work as an Adobe Flash developer. Remember ActionScript and all that? Well, [unsurprisingly](#) after 12 years of working as a freelance Flash developer she suddenly found herself unemployed. Alone. With a baby to feed. She told me in her message that she had attended my training, that she had enjoyed it and also learnt something useful, and that after that she had found a job as a mobile web developer. She wrote to me to say thank you.

I cannot claim that I changed the world, but I might have nudged it a little bit, into something (hopefully) better. This

thought has made every lesson I gave since then much more worthwhile and meaningful.

6. Workplaces suck

Do not expect software corporations to offer any kind of career path. They might do this in the US, but I have never seen any of that in Europe. This means that you are solely responsible for the success of your career. Nobody will tell you “oh, well, next year you can grow to be team leader, then manager, then CTO...”

Not. At. All. Quite the opposite, actually: you were, are and will be a software developer, that is, a relatively expensive factory worker, whose tasks your managers would be happy to offshore, no matter what they tell you.

Do not take a job just for the money. [Software companies have become sweatshops](#) where you are supposed to justify your absurdly high salary with insane hours and unreasonable expectations. And, at least in the case of Switzerland, there is no worker union to help you if things go bad. Actually there are worker unions in Switzerland, but they do not really care about situations that will not land them some kind of media exposure.

Even worse, in most work-

places you will be harassed, particularly if you are a woman, a member of the LGBT community or from a non-Caucasian ethnic group. I have seen developers threatened to have their work visas not renewed if they did not work faster. I have witnessed harassment of women and gay colleagues.

Some parts of our industry are downright disgusting, and you do not need to be in Silicon Valley to live it. You do not need Medium to read it. You could experience that right here in Switzerland. Many banks have atrocious workplaces. Financial institutions want you to vomit code 15 hours a day, even if the Swiss working laws explicitly forbid such treatments. Pharmaceutical companies want you to write code to cheat test results and to help them bypass regulations. Startups want your skin, working for 18 hours for no compensation, telling you bullshit like “because we give you stock options” or “because we are all team players.”

It does not matter that you are Zach Holman and that you can claim in your CV that you literally wrote Github from scratch: [you will be fired](#) for the pettiest of reasons.

It does not matter that the app brings more than half of your employer traffic and revenues; the API team will treat you and your ideas with contempt and sloppiness.

I have been asked to work for free by very well-known people in the industry, some of them even featured in Wikipedia, and it is simply appalling. I will not give out their names, but I will prevent any junior from getting close to them, because people working without ethics *do not deserve anyone's brain.*



Jeff Atwood
@codinghorror

If I could go back in time and tell the younger me exactly one and only one thing, it would be "learn UNIX"

RETWEETS
467

LIKES
597



6:33 PM - 3 Feb 2016

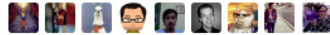


Cat Swart
@Jexx

Every day, with every action and choice, you're either a teacher and an inspiration, or a lesson and a reminder.

RETWEETS
135

LIKES
148



2:04 PM - 25 May 2015

Whenever an HR manager tells you “you must do this (whatever wrong thing in your frame of reference) because we pay you a salary,” remember to answer the following: “you pay me a salary, but I give you my brain in exchange, and I refuse to comply with this order.”

And to top it all, they will put you in an open space, and for some reason they will be proud about it. *Open spaces are a cancer.* They are without a doubt the worst possible office layout ever invented, and the least appropriate for software development—or any type of brain work for that matter.

Remember this: the fact that you *understand* something does not imply that you have to *agree* to it.

Disobey authority. Say “[fuck you, I won't do what you tell me](#)” and change jobs. There are fantastic workplaces out there; not a lot, but they exist. I have been lucky enough to work in some of them. Do not let a bad job kill your enthusiasm. It is not worth it. Disobey and move on.

Or, better yet, become independent.

7. Know your worth

You have probably heard about the “10x Software Engineer”

myth, right? Well here is the thing: it is not a myth, but it does not work the way you think it works.

It works, however, from the employer's point of view: a “10x Software Engineer” generates worth 10 times whatever the employer pays. That means that you, she or he gets 100 KCHF per year, but she or he are actually creating a value worth over a million francs. And of course, *they* get the bonuses at the end of the fiscal year, because, you know, capitalism. Know your worth. Read [Karl Marx](#) and [Thomas Piketty](#). Enough said.

Keep moving; be like the shark that keeps on swimming, because your skills are extremely valuable. Speak out your salary, say it out loud, blog about it, so that your peers know how much their work is worth. Companies want you to shut up about that, so that women are paid 70% of what men are paid. So speak up! Blog about it! Tweet it!

I am making 135 KCHF per year. That was my current salary. How about you? And you? The more we speak out, the less inequality there will be. Any person doing my job with my experience should get the same money, regardless of race, sex,

age or preferred football team. End of the story. But it is not like that. It is not.

8. Send the elevator down

If you are a white male, remember all the privilege you have enjoyed since birth just because you were born that way. It is your responsibility to change the industry and its bias towards more inclusion.

It is your *duty* to send the elevator down.

Take conscious decisions in your life. Be aware of your actions and their effect. Do not blush or become embarrassed for changing your opinions. Say “I'm sorry” when required. Listen. Do not be a hotshot. Have integrity and self-respect.

Do not criticize or make fun of the technology choices of your peers; for other people will have their own reasons to choose them, and they must be respected. Be prepared to change your mind at any time through learning. One day you might like Windows. One day you might like Android. I am actually liking some parts of Android lately. And that is OK.

9. LLVM

Everybody is raving about Swift, but in reality what I pay more attention to these days is LLVM itself.

I think LLVM is the *most* important software project today, as measured in its long-term impact. Objective-C blocks, Rust & Swift (the two most loved, strongly typed and compiled programming languages in the [2016 StackOverflow developer survey](#)), [Dropbox Pyston](#), the Clang Static Analyser, ARC, [Google Souper](#), [Emscripten](#), [LLVM-Sharp](#), [Microsoft LLILC](#), [Rubymotion](#), [cheerp](#), watchOS apps, the



Jochen Wolters
@jochenWolters

Myth: Open offices result in massive collaboration.

Reality: 2 people loudly collaborate; 30 must wear headphones to get any work done.

RETWEETS	LIKES	
7,320	6,526	

1:34 PM - 7 Apr 2016

[Android NDK](#), [Metal](#), all of these things were born out or powered by LLVM.

There are compilers using LLVM as a backend for pretty much all the most important languages of today. The .NET CLR will eventually [interoperate](#) with it, and Mono [already](#) uses it. Facebook has [tried to integrate LLVM with HHVM](#), and WebKit recently [switched from LLVM to the new B3 JIT JavaScript compiler](#).

LLVM is [cross-platform](#), cross-CPU-architecture, cross-language, cross-compiler, cross-eyed-tested, free as in gratis and free as a bird.

Learn all you can about LLVM. This is the galaxy where true innovation is happening now. This is the foundation for the next 20 years.

10. Follow your gut

I had the gut feeling .NET was going to be big when I watched its [introduction in June 2000](#). I had the gut feeling the iPhone was going to be big when I watched its [introduction in 2007](#).

In both cases people laughed

at my face, literally. In both cases I followed my gut feeling and I guess things worked out well.

Follow your gut. You might be lucky, too.

11. APIs are king

Great APIs enable great apps. If the API sucks, the app will suck, too, no matter how beautiful the design.

Remember that *chunky is better than chatty*, and that clients should be dumb; push as much logic as you can down to the API.

Do not invent your own security protocols.

Learn a couple of server-side technologies, and make sure Node is one of those.

Leave REST aside and embrace Socket.io, ZeroMQ, RabbitMQ, Erlang, XMPP; explore realtime as the next step in app development. Realtime is not only for chat apps. Remove polling from the equation forever.

Oh, and start [building bots](#) around those APIs. Just saying.

12. Fight Complexity

Simpler is better. Always. Remember the KISS principle. And I do not mean only at the UI level, but all the way until the deepest layers of your code.

Refactoring, unit tests, code reviews, pull requests, all of these tools are at your disposal to make sure that the code you ship is the simplest possible architecture that works. This is how you build resilient systems for the long term.

Conclusion

The most important thing to remember is that your age does not matter.

One of my sons said to me, "Impossible, Dad. Mathematicians do all their best work by the time they're 40. And you're over 80. It's impossible for you to have a good idea now."



Daniel Méndez
@mendezfe

A customer walks into a bar. He asks for a beer made out of wine. The project manager agrees. Both question the bartender's competence.

RETWEETS	LIKES	
5,003	3,115	

1:24 AM - 22 Mar 2015



Chris Lattner
@clattner_llvm

@owensd Java is 20 years old, C# is 15 - I think it is better to see Swift as a response to that sort of managed language (the next step?)

RETWEETS 11 LIKES 26

11:21 AM - 18 Feb 2015

If you're still awake and alert mentally when you're over 80, you've got the advantage that you've lived a long time and you've seen many things, and you get perspective. I'm 86 now, and it's in the last few years that I've had these ideas. New ideas come along and you pick up bits here and there, and the time is ripe now, whereas it might not have been ripe five or 10 years ago.

Michael Atiyah

Fields Medal and Abel Prize winner Mathematician, quoted in a *Wired* article.

As long as your heart tells you to keep on coding and building new things, you will be young, forever.

In 2035, exactly 19 years from now, somebody will give a talk at a software conference similar to this one, starting like this: "Hi, I am 42 years old, and this is my story."

Hopefully one of you will be giving that presentation; otherwise, it will be an AI bot.

You will provide some anecdotal facts about 2016, for example that it was the year when [David Bowie](#), [Umberto Eco](#), [Gato Barbieri](#) and [Johan Cruyff](#) passed away, or when SQL Server was made [available in Linux](#), or when [Google AlphaGo](#) beat a Go champion, or when the [Panama Papers](#) and the [Turkish Citizenship Database](#) were leaked the same day, or when [Google considered using Swift for Android for the first time](#), or as the last year in which people enjoyed this useless thing called privacy.

We will be three years away



Matthew Jewell
@mattisfrommars

"Ok, let's make a rails app"
"Oh, I need rails first"
"Oh, I need rbenv first"
"Oh I need brew"
"Oh I need xcode tools"
... rage.

RETWEETS 270 LIKES 309

1:19 PM - 22 Jan 2015

from the [Year 2038 Problem](#) and people will be really nervous about it.

Of course I do not know what will happen 19 years from now, but I can tell you three things that will happen for sure:

1. Somebody will ask a question in Stack Overflow about how to filter email addresses using regular expressions.
2. Somebody will release a new JavaScript framework.
3. Somebody will build something cool on top of LLVM.

And maybe you will remember this talk with a smile.

Thank you so much for your attention. This is the talk I gave at [App Builders Switzerland](#) on April 25th, 2016. ■

Spotlight



Adrian Kosmaczewski

Adrian is a writer, a software developer and a teacher. He is the author of two books about mobile software development, and has shipped mobile, web and desktop apps for iOS, Android, Mac OS X, Windows and Linux since 1996. Adrian holds a Master in Information Technology from the University of Liverpool.

But lately I'm more interested on how these technologies enable social inclusion and solve actual problems in society.

What technology has you excited today?

There are many things that I keep an eye on, for example Socket.io, LLVM and also the latest revisions of C++. But lately I'm more interested on how these technologies enable social inclusion and solve actual problems in society.

What are 1-2 blogs, podcasts, newsletters, etc. that you use to stay on top of the fast-changing and ever evolving industry?

I'm not into podcasts, I'm more of a reading guy so I have lots of entries in my RSS reader every

day. The ones I find most interesting are [Daring Fireball](#), [Apple World Today](#), [All About Micro-soft](#), [The Monday Note](#), [asymco](#), [NSHipster](#), [Presentation Zen](#) and [Rands in Repose](#). I also enjoy Swift and iOS newsletters, such as Dave Verwer's [iOS Dev Weekly](#), Natasha Murashev's [Natasha The Robot](#) and [This Week in Swift](#).

Do you have an Internet resource that you recommend, such as Google Docs? Why do you recommend it?

I'm a heavy [iCloud](#) user. Lately it works much better than it used to and I like the integration with the iOS and OS X app ecosystem. I also find [Trello](#) very useful.

What is a personal habit that contributes to your success?

Am I successful? That's a good question. In any case, I guess that if I do not enjoy what I'm doing, I move on. That's all.

If there's one book you'd recommend, what is it and why?

[Ficciones](#) by Jorge Luis Borges. An absolutely astonishing collection of short stories that has something for everyone.

Where can people find out more about you?

The article I published in Medium gives a fair amount of details about me. Medium: [@akosma](#), Twitter: [@akosma](#) ■



Implementers, solvers, and finders

By RANDALL KOUTNIK

I was talking to a former student when he brought up [an article written by a well-seasoned programmer regretting his choice of career](#). This fellow had rejected the management path in order to stay in the coding trenches and as a result, ended up in some absolutely crummy situations. He writes about management antipattern

after antipattern that left him sitting with the bill.

Hearing about a dismal career like that is depressing. So much so, that my student who previously was one of my most enthusiastic programmers had turned morose. Who'd blame him? All he had to look forward to was a life filled with misery caused by other folks' poor

decision-making skills. Learning that sub-par managers will dictate your whole life is almost as traumatizing as dealing with said managers.

To make matters worse, his team had recently mulled over their future career plans at lunch. None of them anticipated that they'd remain programmers. As soon as they could,

they were going to take the leap into management, avoiding what our blogging friend referred to as his “biggest regret”.

I’ve talked to a lot of people who live, eat and breathe programming. It’s hard to stand next to them without hearing about the latest library or tool they’re checking out, and how what they’re building is awesome and is going to revolutionize everything (or is useless but _ aren’t neural networks cool!?!_). Almost all of them echoed the earlier sentiment – programming as a profession wasn’t for them.

How could this be? A group of people, granted the ability to do what they love for great pay and perks, all wanting to move on? It all comes down to:

People want to make decisions rather than execute them.

Turns out science agrees on this: [People want power because they want autonomy](#). Most of the time, folks desire to move up the career ladder not for pay, better title, or keys to the executive washroom (are those still a thing?) but because they wish to be able to exercise greater autonomy over their lives.

[Psychologist Daniel Pink agrees](#) — he’d found that the three qualities that contribute most to workplace satisfaction and overall productivity are autonomy, mastery and purpose.

I too, was bitten by the “management = autonomy” bug. My original career goal was to rise up in the ranks of programmers, eventually crowning myself king of my own startup, the ultimate in autonomy (or so I thought). I carefully chose the work I did at each job to align

with this goal – and succeeded, eventually earning the rank of a lead position (before the company ultimately collapsed, but that’s another story).

During my last job hunt, I had two fantastic options:

- Leader, with a few constraints
- Not Leader, with full autonomy

This was it! I could capitalize on my previous work and finally...not program. Something was wrong here – what did I ultimately want? When it came down to it, I wanted to write code. I love building things and I didn’t want to give that up. Sure, startups mean doing a little bit of everything but the success case everyone’s working towards means outsourcing all of your own work until you’ve forgotten all of the shortcuts to your favorite IDE. I chose the latter job, the one that would grant me the most autonomy.

Implementers, solvers, and finders

Could it be that we’ve utterly mischaracterized how career development as a programmer should work? The guiding trifecta of Junior, Regular, and Senior is [incredibly easy to game](#) (a misguided company offered me a senior level job just under a year into my career). A word ceases to be useful when we can’t agree on its purpose – title relativism means a given title can convey entirely separate messages to different companies. Which is more impressive, a “Senior UI Engineer” or a “JavaScript Architect?”

If we’re to escape this situa-

tion where fantastic fanatic programmers can’t see themselves programming in three years, we need to couple our words to real world meaning. Instead of the Jr/Sr nonsense (which already reeks of the years-of-experience antipattern), why don’t we talk about what the job will actually entail? Let’s define your job title by how much autonomy your day-to-day work gives you.

Do you find that most of your time is simply closing tickets, and your team rarely considers your input? Your title is *Solution Implementer*.

Are you given general problems and left to your own devices on how they’re fixed? When brainstorming, is your input considered by your teammates? You’re working as a *Problem Solver*.

Are you given near-total autonomy in choosing what you work on? Can you tell your boss “That’s an interesting idea but my time would be better spent elsewhere” (and not get fired on the spot)? You’re a *Problem Finder*.

Our regretful fellow author of the post spent most of his time as an Implementer. People (often non-technical) assigned concrete tasks without room for feedback or innovation. Every one of his stories shares the same problem: *He wasn’t given autonomy*.

I don’t mean to say that everyone who’s at the Solution Implementer level should immediately quit their jobs, or that life as an Implementer de facto means you’re a bad programmer.

Beginning programmers who are still learning the basic concepts will thrive as Implementers if handed solutions matching the challenge level they’re

willing to accept. The mere task of implementing something provides plenty of opportunities to learn (in fact, I'd say it's the only way to truly understand something). Beginners of all skill levels will get lost if thrown at a problem with too big of a scope for them to handle.

One can't remain a beginner forever. There needs to be a way for these implementers to level up. We, as a community, don't have an agreed-upon way to take budding programmers and hand them continually-increasing challenges. The standard practice in leveling up seems to be to quit and start hunting for a new job at the next level. This is why my standing advice to beginning programmers is to find a new job after six months (hopefully in the same company).

It can be difficult to tell if a company is actually looking for a Problem Solver or the work is just Implementation in disguise. One org smell to look out for is the ratio of programmers to project/product managers. I've fallen for this, thinking I'd get to take charge of a frontend when what they really wanted me to do was slog through 17,000 issues in JIRA. Determining the true culture of a company is troublesome, but you can start by [interviewing your interviewer](#).

On the other hand, if our programmer does land that Solver job, they will start spreading their wings beyond the basics. At this point their opinion starts to mean something, and the challenges they face take on a new shape as their sphere of responsibility grows.

There's an entirely new set

of skills to learn – suddenly effective communication means a whole lot more than “following the JIRA process correctly”. Solvers need to learn how to evangelize their favorite solutions and defend them against other Solvers who have different preferences.

Solver is an enormous growth area and the typical Solver may spend years making mistakes, learning, and causing *new* catastrophes until their knowledge grows to the point where they're ready to make the final leap to Finder.

Mid-size startups (> 50 people) are a great place for Solvers — there's plenty of problems begging for solutions and often Solvers can branch out into areas that would be verboten to them at a larger, established company. As the startup grows, so does their responsibility — this is a common way for a Solver to grow into a Finder. Another case is for a company to create fertile ground for Finders and recruit them directly.

The final stage of programmer evolution is the Finder. These folks are considered experts in their chosen domain (and are prudent about others). Writing Finder job descriptions is an exercise in futility. As my boss puts it: “I can't tell you the specifics of what you'll be doing here because your first task will be to figure that out.”

A Finder will be able to anticipate problems before they happen, usually because they've been in that situation before. Finders are the canonical “Done and get things smart” that Steve Yegge likes to talk about. Empathy is a critical key in a Finder's

toolbox. Finders need to work with a variety of other folks without swinging the “[I'm smart and because I said so](#)” hammer (otherwise they're [Brilliant Jerks](#)).

Finders need autonomy, by definition. Any job that puts significant restraints on what a programmer can and can't do is a poor fit for a Finder. Poorly-managed startups love to hire smart people and then tell them what to do in precise detail. No one wins.

My hope is this post will create shared vocabulary. This triumvirate of concepts can help great folks find the job that's best for them. A company may be honestly seeking an Implementer, and effectively communicating that to a Solver or Finder will save both parties a lot of time.

More importantly, I want programmers everywhere to realize that it's possible to have autonomy while still writing code for a living. Some may find fulfillment in leadership (I know I do, the siren song has abated but is not gone) but plenty of hackers out there just want to make great things. There's hope for us yet!

Postscript: If this article rang true to you and you feel like you're ready to level up, I'd love to hear from you. What brought you to your current job? What changed? How are you looking for the job that fits you?

Let me know at: isf@rkoutnik.com. ■

Spotlight



Randall Koutnik

Randall is a Senior UI Engineer at Netflix and holds a lot of strong opinions about Star Wars. He'd love to hear from you via hackerbits@rkoutnik.com.

I'm biased to real-time data processing, so RxJS is my favorite here.

What technology has you excited today?

I work in JavaScript, so there's a lot of exciting things going on. Angular 2 brings improvements to every area of frontend dev, compiling lessons learned from across the spectrum. [RxJS](#) and async iterators bring new power to some of the most difficult JS tasks. I'm biased to real-time data processing, so [RxJS](#) is my favorite here.

What are 1-2 blogs, podcasts, newsletters, etc. that you use to stay on top of the fast-changing and ever evolving industry?

I'm subscribed to a few regarding JS/Node but the best tool I've found for keeping up on things is joining chat rooms

(best I've found are on [Slack](#)). The interactive nature means I can learn about things as well as get feedback on my current work. It's the best way to learn (it's ok to ask dumb questions!).

Do you have an Internet resource that you recommend, such as Google Docs? Why do you recommend it?

Not really an Internet tool, but I love [git](#). As a simple commit/push/merge thing, it works fine but once you dive into bisect, rebasing, revert and more, it's one of the best developer productivity tools out there.

What is a personal habit that contributes to your success?

Ask questions, especially dumb ones. We get all wrapped up in

our concern to seem smart that we keep ourselves from making the effort to learn the things that'll make us smart.

If there's one book you'd recommend, what is it and why?

Oh wow, a zillion suggestions just popped into my head. Everyone should read [The Hitchhiker's Guide to the Galaxy](#) and realize that life isn't as serious as we make it out to be.

Where can people find out more about you?

I write at rkoutnik.com tweet as [@rkoutnik](https://twitter.com/rkoutnik) and respond to reader email at blog@rkoutnik.com. ■



20 lines of code that will beat A/B testing every time

By STEVE HANOV

A/B testing is used far too often, for something that performs so badly. It is defective by design: Segment users into two groups. Show the A group the old, tried and true stuff. Show the B group the new whiz-bang design with the bigger buttons and slightly different copy. After a while, take a

look at the stats and figure out which group presses the button more often. Sounds good, right?

The problem is staring you in the face. It is the same dilemma faced by researchers administering drug studies. During drug trials, you can only give half the patients the lifesaving treatment. The others get sugar

water. If the treatment works, group B lost out. This sacrifice is made to get good data. But it doesn't have to be this way.

In recent years, hundreds of the brightest minds of modern civilization have been hard at work not curing cancer. Instead, they have been refining techniques for getting you and me

to click on banner ads. It has been working. Both [Google](#) and [Microsoft](#) are focusing on using more information about visitors to predict what to show them. Strangely, anything better than A/B testing is absent from mainstream tools, including Google Analytics, and Google Website optimizer. I hope to change that by raising awareness about better techniques.

With a simple 20-line change to how A/B testing works, *that you can implement today*, you can *always* do better than A/B testing — sometimes, two or three times better. This method has several good points:

- It can reasonably handle more than two options at once.. e.g. A, B, C, D, E, F, G...
- New options can be added or removed at any time.

But the most enticing part is that *you can set it and forget it*. [If your time is really worth \\$1000/hour](#), you really don't have time to go back and check

how every change you made is doing and pick options. You don't have time to write rambling blog entries about how you got your site redesigned and changed this and that and it worked or it didn't work. Let the algorithm do its job. These 20 lines of code automatically finds the best choice quickly, and then uses it until it stops being the best choice.

The multi-armed bandit problem

The multi-armed bandit problem takes its terminology from a casino. You are faced with a wall of slot machines, each with its own lever. You suspect that some slot machines pay out more frequently than others. How can you learn which machine is the best, and get the most coins in the fewest trials?

Like many techniques in machine learning, the [simplest strategy](#) is [hard to beat](#). More complicated techniques are worth considering, but they may eke out only a few hundredths

of a percentage point of performance.

One strategy that has been shown to perform well time after time in practical problems is the *epsilon-greedy method*. We always keep track of the number of pulls of the lever and the amount of rewards we have received from that lever. 10% of the time, we choose a lever at random. The other 90% of the time, we choose the lever that has the highest expectation of rewards.

Why does this work?

Let's say we are choosing a colour for the "Buy now!" button. The choices are orange, green, or white. We initialize all three choices to 1 win out of 1 try. It doesn't really matter what we initialize them too, because the algorithm will adapt. So when we start out, the internal test data looks like this.

Orange	Green	White
1/1 = 100%	1/1=100%	1/1=100%

```
def choose():
    if math.random() < 0.1:
        # exploration!
        # choose a random lever 10% of the time.
    else:
        # exploitation!
        # for each lever,
            # calculate the expectation of reward.
            # This is the number of trials of the lever divided by the total reward
            # given by that lever.
        # choose the lever with the greatest expectation of reward.
    # increment the number of times the chosen lever has been played.
    # store test data in redis, choice in session key, etc..

def reward(choice, amount):
    # add the reward to the total for the given lever.
```

▲ Epsilon-greedy method

Then a web site visitor comes along and we have to show them a button. We choose the first one with the highest expectation of winning. The algorithm thinks they all work 100% of the time, so it chooses the first one: orange. But, alas, the visitor doesn't click on the button.

Orange	Green	White
1/2 = 50%	1/1=100%	1/1=100%

Another visitor comes along. We definitely won't show them orange, since we think it only has a 50% chance of working. So we choose Green. They don't click. The same thing happens for several more visitors, and we end up cycling through the choices. In the process, we refine our estimate of the click through rate for each option downwards.

Orange	Green	White
1/4 = 25%	1/4=25%	1/4=25%

But suddenly, someone clicks on the orange button! Quickly, the browser makes an Ajax call to our reward function `$.ajax(url: "/reward?testname=buy-button");` and our code updates the results:

Orange	Green	White
2/5 = 40%	1/4=25%	1/4=25%

When our intrepid web developer sees this, he scratches his head. What the F*? The orange button is the *worst* choice. Its font is tiny! The green button is obviously the better one. All is

lost! The greedy algorithm will always choose it forever now!

But wait, let's see what happens if Orange is really the sub-optimal choice. Since the algorithm now believes it is the best, it will always be shown. That is, until it stops working well. Then the other choices start to look better.

Orange	Green	White
2/9 = 22%	1/4=25%	1/4=25%

After many more visits, the best choice, if there is one, will have been found, and will be shown 90% of the time. Here are some results based on an actual web site that I have been working on. We also have an estimate of the click through rate for each choice.

Orange	Green	White
114/4071 = 2.8%	205/6385=3.2%	59/2264=2.6%

Edit: What about the randomization?

I have not discussed the randomization part. The randomization of 10% of trials forces the algorithm to explore the options. It is a trade-off between trying new things in hopes of something better, and sticking with what it knows will work.

There are several variations of the epsilon-greedy strategy. In the epsilon-first strategy, you can explore 100% of the time in the beginning and once you have a good sample, switch to pure-greedy. Alternatively, you can have it decrease the amount of exploration as time passes.

The epsilon-greedy strategy that I have described is a good balance between simplicity and

performance. Learning about the other algorithms, such as UCB, Boltzmann Exploration, and methods that take context into account, is fascinating, but optional if you just want something that works.

Wait a minute, why isn't everybody doing this?

Statistics is hard for most people to understand. People distrust things that they do not understand, and they especially distrust machine learning algorithms, even if they are simple. Mainstream tools don't support this, because then you'd have to educate people about it, and about statistics, and that is hard. Some common objections might be:

- Showing the different options at different rates will skew the results. (No it won't. You always have an estimate of the click through rate for each choice).
- This won't adapt to change. (Your visitors probably don't change. But if you really want to, in the reward function, multiply the old reward value by a forgetting factor).
- This won't handle changing several things at once that depend on each-other. (Agreed. Neither will A/B testing.)
- I won't know what the click is worth for 30 days so how can I reward it? ■



Steve Hanov

Steve can be found at various coffee shops in Waterloo, Ontario, where he writes code and occasionally responds to emails from customers of his web businesses websequencediagrams.com and zwibbler.com. He has three children, one wife, and two birds.

I am excited by the incredible change that everyone has the Internet in their pockets.

What technology has you excited today?

I am excited by the incredible change that everyone has the Internet in their pockets. I am always connected to my wife, and when they are old enough, my kids, not to mention the sum total of human knowledge. The way we interact is changing. I'm increasingly talking to my phone, asking it for quick answers. I hate when I have to look at the screen to do something.

What are 1-2 blogs, podcasts, newsletters, etc. that you use to stay on top of the fast-changing and ever evolving industry?

I listen to the [Startups For The Rest of Us](#) podcast to get motivation to work on things. When I have a rare spare moment, I

check the [best of Hacker News](#) to get a sense of new developments, but this is pure decadence.

Do you have an Internet resource that you recommend, such as Google Docs? Why do you recommend it?

I run my business through the spreadsheets in [Google Docs](#). [Google Keep](#) is wonderful for personal notes. I love [Stripe](#), because I can watch my money flow through it.

What is a personal habit that contributes to your success?

It is not habit, but circumstance that contributes to my success. Before having children, I would waste a lot of time. I read a lot, but produced little. With three children, the moment I have

dispersed them all to school and daycare, I need to be productive, getting everything I can done before I get the inevitable call that one has a fever or a tummy ache.

If there's one book you'd recommend, what is it and why?

I recommend two books: [The Willpower Instinct](#) and [Predictably Irrational](#). With the two of them, you will understand yourself and other people.

Where can people find out more about you?

I have written many interesting tech articles on my blog steve-hanov.ca. My secret is every time I write one, I delete one, and so by now many of them are good. ■



Are Progressive Web Apps the future of the Internet?

An interview with Henrik Joreteg



Henrik Joreteg is a PWA developer, consultant, and educator. He is the author of [Human JavaScript](#) and creator of [Ampersand.js](#), [SimpleWebRTC](#), [Talky.io](#) and [over 200 JavaScript libraries](#). He's also spoken at [O'Reilly's FluentConf](#) and [FFConf](#).

Here at *Hacker Bits*, Progressive Web Apps (PWA) have consistently come up as a hot button topic among you, our readers, and on social media. But is PWA the real deal? Or is it just another fad that'll soon be forgotten?

To find out, we chat with Henrik Joreteg, a PWA expert who's been working with his clients to architect, build and train teams on building performant mobile web apps.

In last month's issue of *Hacker Bits*, you [briefly covered Progressive Web Apps](#). Can you go over the basics and explain what is a Progressive Web App (PWA)?

To steal from Alex Russell, it's just a website that took all the right vitamins. It starts life in a browser tab like any other site, but can *progressively* become an app.

A bit more specifically, if it meets the technical requirements and a user visits the app again, they'll be prompted *by the browser* if they want to add it to their home screen. From that point on, when a user opens the app from the home screen it receives the same vi-

sual treatment by the Operating System as if it were a fully native app. Meaning it has a home screen icon, shows a splash screen when opening, runs from cache first, etc.

You should also read [Alex's post](#) in which he coins the phrase.

What are the top 3-4 things that excite you about PWAs?

1. We now have a way to write an app that's treated as a first-class citizen by an operating system with over a billion active users.
2. The tech required to do this is [either supported, or being built by every browser vendor other than Safari](#).
3. The web can now be made to be completely reliable. And network connection is optional. With Service-Worker the web can now implement the same caching strategies previously reserved for native apps. Some people really focus on the "offline-first" thing. But it's not about being fully functional without a network connection - it's about knowing that when

you tap my app icon, *it will open immediately, every time, regardless of network conditions*.

4. Native push notifications.

What technologies and tools are used to build PWAs?

1. [Application manifest](#)
2. [Service Worker](#)
3. HTTPS... yup. Gotta do it or this stuff won't work :)

What are the top 3 challenges preventing PWAs from being more broadly adopted today? How far are we along?

1. Awareness. There's nothing in the way at this point. I think we're going to see them popping up all over the place in the latter half of this year.
2. Confusion. These things are hard to talk about: is it an app? Is it a web app? Do users like them?
3. It's early but I think we're going to see an absolute explosion of them this year and next.

The reason I'm so bullish about this tech is because of how dramatically it seems to improve on-boarding and decrease cost of customer acquisition.

Can you tell us about one of your experiences building PWAs for your clients or yourself? What is working well and not so well?

I can't talk specifics about my current client. But I can say that the biggest challenge is the mind-shift required by developers and product alike. A lot of product leads don't know they want one yet or what it can mean for them.

Also, PWAs put the web solidly in the realm of the "app" in terms of architecture. Frankly, most web developers are not used to building apps as fully self-contained client side applications. This will take a bit of time.

Ultimately, the reason I'm so bullish about this tech is because of how dramatically it seems to improve on-boarding and decrease cost of customer acquisition.

Rather than showing spammy app banners and "please install our app" door slams, it just asks the users at a point when they're likely to want it and the "install" is non-existent because they're already using the app!

There's no, "please install". Instead of shipping them off to an app store and hoping they'll

install your 40mb app and log back in there, you've already got 'em. This is why I think businesses will flock to this tech once we see more data about business impact.

Lots of articles spotlight the ServiceWorker. What is a ServiceWorker, and how does it fit into the world of PWAs?

ServiceWorker is a separate JavaScript file that your web app can "register." Once registered it acts as a proxy. All requests your app makes from that point on (even to external domains) go through that ServiceWorker proxy first! From that script you can control caches and choose whether to answer from cache or network or both, or whatever else you can dream up.

In addition, ServiceWorker runs in the background, which means you can do interesting things like send it a push notification that will be shown to the user even when the browser is closed!

Other interesting features are being added via ServiceWorker too. One such example is background sync where your service worker can keep trying a request in the background until it resolves. Which means actions

queued up while fully offline could be re-tried and completed long after your app and the browser has been closed.

In the future this script can be extended to do things like background geolocation tracking, etc. Imagine being able to build an app like "Runkeeper" entirely with web tech.

How are data storage and offline capabilities dealt with by PWAs?

There's a new "CacheStorage" API (which is also available as "window.caches" outside the worker). This is a Promise-based API optimized for storing/retrieving Request and Response objects and matching them to their related URLs.

Since they are technically web apps, how are search engine indexing and bookmarking handled by PWAs?

The story here is still evolving. But it's very much like any Single Page App at this point. If you want your content indexed it's probably best to pre-render as much of it on the server as possible. That said, GoogleBot is pretty great at running JS. But in reality, most people building these types of experiences are

This is about building completely reliable experiences with web tech...

building things for logged in users with personalized content anyway.

All this said, I would not be in the least bit surprised if at some point PWAs get preferential treatment by Google's search algorithms. I hope this happens because in my opinion, this would all but ensure wide adoption.

What else do you think our readers should know about PWAs?

The "offline" word is a distraction, in my opinion. Frankly, so is the word "progressive." This is about building completely reliable experiences with web tech and teaching users they can rely on web apps for things they normally think they'd want native apps for.

What is a parting piece of advice to our readers looking to get started learning and building PWAs?

1. Life will be easier if you build your PWA as a set of completely static assets. It makes it easier to know what you should cache, etc.
2. Try writing a ServiceWorker from scratch to get your head around it, but then check out all the goodies in: <https://github.com/GoogleChrome/sw-tool-box>.
3. Right now all IoT stuff has a related native app. This simply will not scale when there are hundreds of connected devices you want to interact with. The web will win here, eventually, and PWAs are the logical way to interact with a world full of connected devices.

Where can people find out more about you, and what is the best way for people to connect with you?

I use Twitter as a professional tool and tweet almost exclusively about tech stuff. So connect with me there: [@HenrikJoreteg](https://twitter.com/HenrikJoreteg).

Since going independent last year, I've brought [my blog](#) back to life and will post my best thoughts there.

If you're interested in my professional help, you can [get in touch about consulting](#). ■

19 tips for everyday git use

By ALEX KRAS

I've been using git full time for the past 4 years, and I wanted to share the most practical tips that I've learned along the way. Hopefully, it will be useful to somebody out there.

If you are completely new to git, I suggest reading [Git Cheat Sheet](#) first. This article is aimed at somebody who has been using git for three months or more.

Table of contents

1. Parameters for better logging

```
git log --oneline --graph
```

2. Log actual changes in a file

```
git log -p filename
```

3. Only Log changes for some specific lines in file

```
git log -L 1,1:some-file.txt
```

4. Log changes not yet merged to the parent branch

```
git log --no-merges master..
```

5. Extract a file from another branch

```
git show some-branch:some-file.js
```

6. Some notes on rebasing

```
git pull --rebase
```

7. Remember the branch structure after a local merge

```
git merge --no-ff
```

8. Fix your previous commit, instead of making a new commit

```
git commit --amend
```

9. Three stages in git, and how to move between them

```
git reset --hard HEAD and git status -s
```

10. Revert a commit, softly

```
git revert -n
```

11. See difference for the entire project (not just one file at a time) in a 3rd party diff tool

```
git difftool -d
```

12. Ignore the white space

```
git diff -w
```

13. Only "add" some changes from a file

```
git add -p
```

14. Discover and zap those old branches

```
git branch -a
```

15. Stash only some files

```
git stash -p
```

16. Good commit messages

17. Git Auto-completion

18. Create aliases for your most frequently used commands

19. Quickly find a commit that broke your feature (EXTRA AWESOME)

`git bisect`

1. Parameters for better logging

Sample Command: `git log --oneline --graph`

Chances are, by now you've used `git log`. It supports a number of command line parameters, which are very powerful, especially when used in combination. Here are the ones that I use the most:

- `--author="Alex Kras"`
Only show commits made by a certain author
- `--name-only`
Only show names of files that changed
- `--oneline`
Show commit data compressed to one line
- `--graph`
Show dependency tree for all commits
- `--reverse`
Show commits in reverse order (Oldest commit first)
- `--after`
Show all commits that happened after certain date
- `--before`
Show all commits that happened before certain data

For example, I once had a manager who required weekly reports submitted each Friday. So every Friday I would just run: `git log --author="Alex Kras" --after="1 week ago" --oneline`, edit it a little and send it in to the manager for review.

Git has a lot more command line parameters that are handy. Just run `man git-log` and see what it can do for you.

If everything else fails, git has a `--pretty` parameter that lets you create a highly customizable output.

```
g/r/git-rebase-todo
pick afece96 Commit 1
pick a6671aa Commit 2
pick 2261d76 Commit 3
pick 8f462e7 Commit 4
pick 7c9d689 First bad commit
pick e549232 Commit 5
pick e5efc0e Commit 6

# Rebase 11a77a3..e5efc0e onto 11a77a3
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
# Note that empty commits are commented out
~
<rebase-merge/git-rebase-todo CWD: /Users/alexk/Desktop/express Line: 1
```

2. Log actual changes in a file

Sample Command: `git log -p filename`

`git log -p` OR `git log -p filename` lets you view not only the commit message, author, and date, but actual changes that took place in each commit.

Then you can use the regular `less` search command of "slash" followed by your search term/ `{{your-search-here}}` to look for changes to a particular keyword over time. (Use lower case n to go to the next result, and upper case N to go to the previous result).

3. Only log changes for some specific lines in a file

Sample Command: `git log -L 1,1:some-file.txt`

You can use `git blame filename` to find the person responsible for every line of the file.

```
express - less - 72x21
Author: Douglas Christopher Wilson <doug@somethingdoug.com>
Date: Thu Oct 23 02:20:51 2014 -0400

docs: visionmedia is now tj on Github

commit 5f7a37ee517e172c0762fc3debaf94c066e531f9
Author: Fishrock123 <fishrock123@rocketmail.com>
Date: Sat Oct 11 14:12:03 2014 -0400

docs: misc. tweaks

closes #2394

commit ff3a368b2f9a7e640fb3fd18f116de5352e73d58
Author: Douglas Christopher Wilson <doug@somethingdoug.com>
Date: Thu Oct 23 02:08:34 2014 -0400

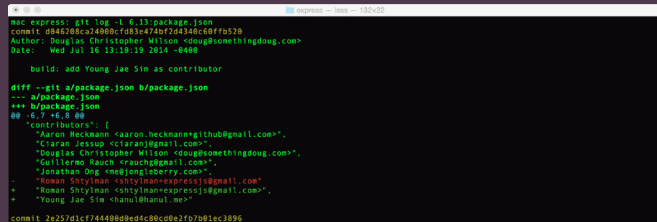
deps: update example dependencies

commit ccc45a74f89b45a6551bc8ff305581dbc8175bca
/visionmedia
```

`git blame` is a great tool, but sometimes it does not provide enough information.

An alternative is provided by `git log` with a `-L` flag. This flag allows you to specify particular lines in a file that you are interested in. Then Git would only log changes relevant to those lines. It's kind of like `git log -p` with focus.

```
git log -L 1,1:some-file.txt
```



```
mac express: git log -L 1,1:package.json
commit db4628c2400cfd3e474b72d34dc6d4f7520
Author: Douglas Christopher Wilson <doug@somethingdoug.com>
Date:   Wed Jul 16 13:10:19 2014 -0800

    build: add Young Jae Sim as contributor

diff --git a/package.json b/package.json
--- a/package.json
+++ b/package.json
@@ -6,7 +6,8 @@
   "contributors": [
     "Aaron Heckmann <aaron.heckmann@github.com>",
     "Ciaran Jessup <ciarasj@gmail.com>",
     "Douglas Christopher Wilson <doug@somethingdoug.com>",
     "Guillermo Rauch <rauchg@gmail.com>",
     "Jonathan Ong <me@jongleberry.com>",
     "Roman Shityn <shityan@expressjs.com>",
     "Roman Shityn <shityan@expressjs.com>",
     "Young Jae Sim <chauni@naver.com>"
   ],
   "commit 2a257d1c7444080ed4c80c0e27b7081ec3896
```

4. Log changes not yet merged to the parent branch

Sample: `git log --no-merges master..`

If you ever worked on a long-lived branches, with multiple people working on it, chances are you've experienced numerous merges of the parent branch (i.e. master) into your feature branch. This makes it hard to see the changes that took place on the master branch vs. the changes that have been committed on the feature branch and which have yet to be merged.

`git log --no-merges master..` will solve the issue. Note the `--no-merges` flag indicate to only show changes that have not been merged yet to ANY branch, and the `master..` option, indicates to only show changes that have not been merged to master branch. (You must include the `..` after master).

You can also do `git show --no-merges master..` or `git log -p --no-merges master..` (output is identical) to see actual file changes that have yet to be merged.

5. Extract a file from another branch

Sample: `git show some-branch:some-file.js`

Sometimes it is nice to take a pick at an entire file on a different branch, without switching to this branch.

You can do so via `git show some-branch-name:some-file-name.js`, which would show the file in your terminal.

You can also redirect the output to a temporary file, so you can perhaps open it up in a side by side view in your editor of choice.

```
git show some-branch-name:some-file-name.js > delete-me.js
```

Note: If all you want to see is a diff between two files, you can simple run:

```
git diff some-branch some-filename.js
```

6. Some notes on rebasing

Sample: `git pull --rebase`

We've talked about a lot of merge commits when working on a remote branch. Some of those commits can be avoided by using `git rebase`.

Generally I consider rebasing to be an advanced feature, and it's probably best left for another post.

Even the git book has the following to say about rebasing.

"Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family."

<https://git-scm.com/book/en/v2/Git-Branching-Rebasing#The-Perils-of-Rebasing>

That being said, rebasing is not something to be afraid of either, rather something that you should do with care.

Probably the best way to rebase is using *interactive rebasing*, invoked via `git rebase -i {{some commit hash}}`. It will open up an editor, with self-explanatory instruction. Since rebasing is outside of the scope of this article, I'll leave it at that.


```
g/r/git-rebase-todo
pick afece96 Commit 1
pick a6671aa Commit 2
pick 2261d76 Commit 3
pick 8f462e7 Commit 4
pick 7c9d689 First bad commit
pick e549232 Commit 5
pick e5efc0e Commit 6

# Rebase 11a77a3..e5efc0e onto 11a77a3
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~
<rebase-merge/git-rebase-todo CWD: /Users/alexk/Desktop/express Line: 1
```

One particular rebase that is very helpful is `git pull --rebase`.

For example, imagine you are working on a local version of a master branch, and you made one small commit. At the same time, somebody else checked in a week worth of work onto the master branch. When you try to push your change, git tells you to do a `git pull` first, to resolve the conflict. Being a good citizen that you are, you do a `git pull` to end up with the following commit message auto generated by git.

*Merge remote-tracking branch
'origin/master'*

While this is not a big deal and is completely safe, it does clutter log history a bit.

In this case, a valid alternative is to do a `git pull --rebase` instead.

This will force git to first pull the changes, and then re-apply (rebase) your un-pushed commits on top of the latest version of the remote branch, as if they just took place. This will remove the need for merge and the ugly merge message.

7. Remember the branch structure after a local merge

Sample: `git merge --no-ff`

I like to create a new branch for every new bug or feature. Among other benefits, it helps me get

great clarity on how a series of commits may relate to a particular task. If you ever merged a pull request on github or a similar tool, you will in fact be able to nicely see the merged branch history in `git log --oneline --graph` view.

If you ever try to merge a local branch, into another local branch, you may notice git has flattened out the branch, making it show up as a straight line in git history.

If you want to force git to keep branches history, similarly to what you would see during a pull request, you can add a `--no-ff` flag, resulting in a nice commit history tree.

`git merge --no-ff some-branch-name`

```
mac express: git merge --no-ff no-ff-demo
Merge made by the 'recursive' strategy.
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
mac express:
mac express: git log --oneline --graph
* 397215d Merge branch 'no-ff-demo'
| \
| * b4ae5f8 Another commit
| * 8abe7f9 Some commit
| /
* 11a77a3 Fix inner numeric indices incorrectly altering parent req.params
* ee90042 Fix infinite loop condition using mergeParams: true
* 97b2d70 4.13.2
* a4fcd91 deps: update example dependencies
* a559ca2 build: istanbul@0.3.17
* c398a99 deps: type-is@1.6.6
* 1cea9ce deps: accepts@1.2.12
* e33c503 deps: path-to-regexp@0.1.7
* 9848645 Merge tag '3.21.2'
```

8. Fix your previous commit, instead of making a new commit

Sample: `git commit --amend`

This one is pretty straightforward. Let's say you made a commit and then realized you made a typo. You could make a new commit with a "descriptive" message *typo*. But there is a better way.

If you haven't pushed to the remote branch yet, you can simply do the following:

1. Fix your typo
2. Stage the newly fixed file via `git add some-fixed-file.js`
3. Run `git commit --amend` which would add the most recent changes to your latest commit. It will also give you a chance to edit the commit message.
4. Push the clean branch to remote, when ready

If you are working on your own branch, you can fix commits even after you have pushed, you

would just have to do a `git push -f` (-f stands for force), which would over-ride the history. But you *WOULD NOT want to do this* on a branch that is being *used by other people* (as discussed in re-base section above). At that point, a “typo” commit, might be your best bet.

9. Three stages in git, and how to move between them

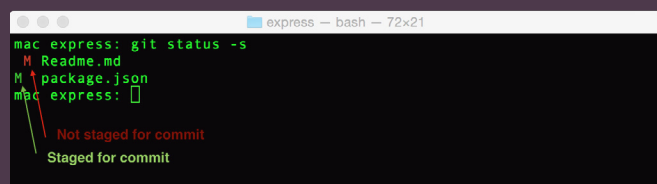
Sample: `git reset --hard HEAD` and `git status -s`

As you may already know by now, a file in git can be in 3 stages:

1. Not staged for commit
2. Staged for commit
3. Committed

You can see a long description of the files and state they are in by running `git status`. You move a file from “not staged for commit” stage to “staged for commit” stage, by running `git add file-name.js` or `git add .` to add all files at once.

Another view that makes it much easier to visualize the stages is invoked via `git status -s` where `-s` stand for short (I think), and would result in an output that looks like that:



Obviously, `git status` will not show files that have already been committed, you can use `git log` to see those instead. :)

There are a couple of options available to you to move the files to a different stage.

Resetting the files

There are 3 types of reset available in git. A reset allows you to return to a particular version in git history.

1. `git reset --hard {{some-commit-hash}}`
Return to a particular point in history. *All changes made after this commit are discarded.*

2. `git reset {{some-commit-hash}}`
Return to a particular point in history. *All changes made after this commit are moved to “not yet staged for commit” stage.* Meaning you would have to run `git add .` and `git commit` to add them back in.
3. `git reset --soft {{some-commit-hash}}`
Return to a particular point in history. *All changes made after this commit are moved to “staged for commit” stage.* Meaning you only need to run `git commit` to add them back in.

This may appear as useless information at first, but it is actually very handy when you are trying to move through different versions of the file.

Common use cases that I find myself using the reset are below:

1. I want to forget all the changes I’ve made, clean start
`git reset --hard HEAD` (most common)
2. I want to edit, re-stage and re-commit files in some different order
`git reset {{some-start-point-hash}}`
3. I just want to re-commit past 3 commits as one big commit
`git reset --soft {{some-start-point-hash}}`

Check out some files

If you simply want to forget some local changes for some files, but at the same time want to keep changes made in other files, it is much easier to check out committed versions of the files that you want to forget, via:

```
git checkout forget-my-changes.js
```

It’s like running `git reset --hard` but only on some of the files.

As mentioned before you can also check out a different version of a file from another branch or commit.

```
git checkout some-branch-name file-name.js and  
git checkout {{some-commit-hash}} file-name.js
```

You’ll notice that the checked out files will be in a “staged for commit” stage. To move them back to “un-staged for commit” stage, you would

have to do a `git reset HEAD file-name.js`. You can run `git checkout file-name.js` again, to return the file to its original state.

Note that running `git reset --hard HEAD file-name.js` does not work. In general, moving through various stages in git is a bit confusing and the pattern is not always clear, which I hoped to remedy a bit with this section.

10. Revert a commit, softly

Sample: `git revert -n`

This one is handy if you want to undo a previous commit or two, look at the changes, and see which ones might have caused a problem.

Regular `git revert` will automatically re-commit reverted files, prompting you to write a new commit message. The `-n` flag tells git to take it easy on committing for now, since all we want to do is look.

11. See diff-erence for the entire project (not just one file at a time) in a 3rd party diff tool

Sample: `git difftool -d`

My favorite diff-ing program is [Meld](#). I fell in love with it during my Linux times, and I carry it with me.

I am not trying to sell you on Meld, though. Chances are you have a diff-ing tool of choice already, and git can work with it too, both as a merge and as a diff tool. Simply run the following commands, making sure to replace Meld with your favorite diff tools of choice:

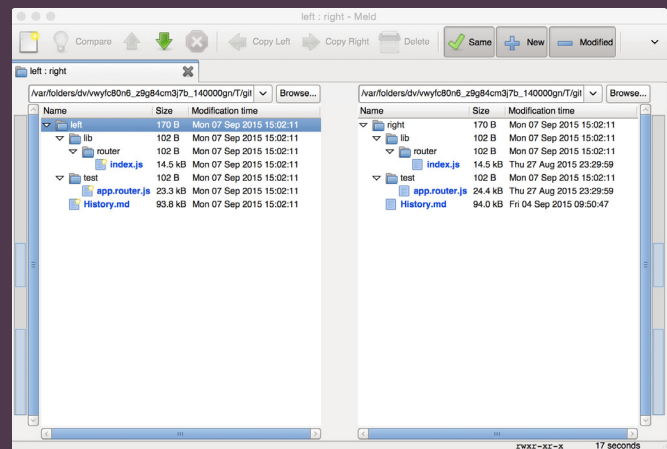
```
1 git config --global diff.tool meld
2 git config --global merge.tool meld
3
```

After that all you have to do is run `git difftool some-file.js` to see the changes in that program instead of the console.

But some of the diff-ing tools (such as Meld) support full directory diffs.

If you invoke `git difftool` with a `-d` flag, it will try to diff the entire folder, which could be really handy at times.

`git difftool -d`



12. Ignore the white space

Sample: `git diff -w` or `git blame -w`

Have you ever re-indented or re-formatted a file, only to realize that now `git blame` shows that you are responsible for everything in that file?

Turns out, git is smart enough to know the difference. You can invoke a lot of the commands (i.e. `git diff`, `git blame`) with a `-w` flag, and git will ignore the white space changes.

13. Only “add” some changes from a file

Sample: `git add -p`

Somebody at git must really like the `-p` flag, because it always comes with some handy functionality.

In case of `git add`, it allows you to interactive select exactly what you want to be committed. That way you can logically organize your commits in an easy to read manner.

```

mac express: git add -p
diff --git a/package.json b/package.json
index 3b4389e..5112b4d 100644
--- a/package.json
+++ b/package.json
@@ -1,5 +1,5 @@
 {
-  "name": "express",
+  "name": "express-custom",
   "description": "Fast, unopinionated, minimalist web framework",
   "version": "4.13.2",
   "author": "TJ Holowaychuk <tj@vision-media.ca>",
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit: do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
@@ -1,5 +1,5 @@
 {
-  "name": "express",
+  "name": "express-custom",
   "description": "Fast, unopinionated, minimalist web framework",
   "version": "4.13.2",
   "author": "TJ Holowaychuk <tj@vision-media.ca>",
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? 

```

14. Discover and zap those old branches

Sample: `git branch -a`

It is common for a large number of remote branches to just hang around, some even after they have been merged into the master branch. If you are a neat freak (at least when it comes to code) like me, chances are they will irritate you a little.

You can see all of the remote branches by running `git branch` with the `-a` flag (show all branches) and the `--merged` flag would only show branches that are fully merged into the master branch.

You might want to run `git fetch -p` (fetch and purge old data) first, to make sure your data is up to date.

```

mac express: git branch -a
* master
  remotes/origin/1.x
  remotes/origin/2.x
  remotes/origin/3.x
  remotes/origin/4.x
  remotes/origin/5.0
  remotes/origin/5.x
  remotes/origin/HEAD -> origin/master
  remotes/origin/benchmark
  remotes/origin/external-isAbsolute
  remotes/origin/master
  remotes/origin/mscdex-router-optimize
  remotes/origin/slim-benchmark
  remotes/origin/streaming-render
mac express: git branch -a --merged
* master
  remotes/origin/3.x
  remotes/origin/4.x
mac express: 

```

If you want to get really fancy, you can get a list of all the remote branches, and the list of last commits made on those branches by running:

```

git for-each-ref --sort=committerdate --format='%(%refname:short) * %(%authorname) * %(%committerdate:relative)' refs/remotes/ | column -t -s '*'

```

```

mac express: git for-each-ref --sort=committerdate --format='%(%refname:short) * %(%authorname) * %(%committerdate:relative)' refs/remotes/ | column -t -s '*'
origin/1.x      TJ Holowaychuk      4 years, 6 months ago
origin/2.x      TJ Holowaychuk      3 years ago
origin/streaming-render  Roman Shitylman    1 year, 10 months ago
origin/benchmark Douglas Christopher Wilson 1 year, 2 months ago
origin/mscdex-router-optimize Douglas Christopher Wilson 1 year, 1 month ago
origin/slim-benchmark Douglas Christopher Wilson 1 year, 1 month ago
origin/5.x      Douglas Christopher Wilson 10 months ago
origin/external-isAbsolute Jeremiah Senkpiel     5 months ago
origin/4.x      Douglas Christopher Wilson 9 weeks ago
origin/5.0      Douglas Christopher Wilson 9 weeks ago
origin/3.x      Douglas Christopher Wilson 5 weeks ago
origin/master   Brendan Ashworth    5 weeks ago
origin/HEAD     Brendan Ashworth    5 weeks ago
mac express: 

```

Unfortunately, there is no easy way (that I know of) to only show merged branches. So you might have to just compare the two outputs or write a script to do it for you.

15. Stash only some files

Sample: `git stash -keep-index` or `git stash -p`

If you don't yet know what `git stash` does, it simply puts all your unsaved changes on a "git stack" of sorts. Then at a later time you can do `git stash pop` and your changes will be re-applied. You can also do `git stash list` to see all your stashed changes. Take a look at `man git-stash` for more options.

One limitation of regular `git stash` is that it will stash all of the files at once. And sometimes it is handy to only stash some of the files, and keep the rest in your working tree.

Remember the magic `-p` command? Well it's really handy with `git stash` as well. As you may have probably guessed by now, it will ask you to see which chunks of changes you want to be stashed.

Make sure to hit `?` while you are at it to see all available options.

```
mac express: git stash -p
diff --git a/Readme.md b/Readme.md
index 8da83a5..4735255 100644
--- a/Readme.md
+++ b/Readme.md
@@ -1,6 +1,6 @@
[[Express Logo](https://i.cloudup.com/zfY6lL7eFa-3000x3000.png)](http://expressjs.com/)

- Fast, unopinionated, minimalist web framework for [node](http://nodejs.org).
+ Fast, unopinionated, minimalist web framework for [node](http://nodejs.org)..

[[NPM Version]][npm-image]][npm-url]
[[NPM Downloads]][downloads-image]][downloads-url]
Stash this hunk [y,n,q,a,d,/,e,?]? ?
y - stash this hunk
n - do not stash this hunk
q - quit; do not stash this hunk or any of the remaining ones
a - stash this hunk and all later hunks in the file
d - do not stash this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
@@ -1,6 +1,6 @@
[[Express Logo](https://i.cloudup.com/zfY6lL7eFa-3000x3000.png)](http://expressjs.com/)

- Fast, unopinionated, minimalist web framework for [node](http://nodejs.org).
+ Fast, unopinionated, minimalist web framework for [node](http://nodejs.org)..

[[NPM Version]][npm-image]][npm-url]
[[NPM Downloads]][downloads-image]][downloads-url]
Stash this hunk [y,n,q,a,d,/,e,?]? [
```

Another handy trick, for stashing only some of the files, is to:

1. add the files that you DO NOT want to get stashed (i.e. `git add file1.js, file2.js`)
2. Call `git stash --keep-index`. It will only stash files that have not been added.
3. Call `git reset` to un-stage the added files and continue your work.

16. Good commit messages

A little while ago I came across a great article on how to write a good commit message. Check it out here: [How to Write a Git Commit Message](#).

One rule that really stood out for me is, “every good commit should be able to complete the following sentence”:

When applied, this commit will:
{{ YOUR COMMIT MESSAGE }}

For example:

- When applied this commit will *Update README file*
- When applied this commit will *Add validation for GET /user/:id API call*
- When applied this commit will *Revert commit 12345*

17. Git auto-completion

Git packages for some operating systems (i.e. Ubuntu) come with git auto completion enabled by default. If your operating system did not come with one (Mac doesn't), you can easily enable it by following these guidelines:

<https://git-scm.com/book/en/v1/Git-Basics-Tips-and-Tricks#Auto-Completion>

18. Create aliases for your most frequently used commands

TLDR; Use git or bash aliases for most commonly used long git commands

Best way to use Git is via command line, and the best way to learn the command line is by doing everything the hard way first (typing everything out).

After a while, however, it might be a good idea to track down your most used commands, and create an easier aliases for them.

Git comes with built in aliases, for example you can run the following command once:

```
git config --global alias.l "log --oneline --graph"
```

Which would create a new git alias named `l`, that would allow you to run:

```
git l instead of git log --oneline --graph
```

Note that you can also append other parameters after the alias (i.e. `git l --author="Alex"`).

Another alternative, is good old Bash alias.

For example, I have the following entry in my `.bashrc` file.

```
alias gil="git log --oneline --graph", allowing me to use gil instead of the long command, which is even 2 character shorter than having to type git l. :).
```

19. Quickly find a commit that broke your feature (EXTRA AWESOME)

Sample: `git bisect`

`git bisect` uses divide and conquer algorithm to find a broken commit among a large number of commits.

Imagine yourself coming back to work after a week-long vacation. You pull the latest version of the project only to find out that a feature that you worked on right before you left is now broken.

You check the last commit that you've made before you left, and the feature appear to work there. However, there has been over a hundred of other commits made after you left for your trip, and you have no idea which of those commits broke your feature.

At this point you would probably try to find the bug that broke your feature and use `git blame` on the breaking change to find the person to go yell at.

If the bug is hard to find, however, you could try to navigate your way through the commit history, in attempt to pin point where things went bad.

The second approach is exactly why `git bisect` is so handy. It will allow you to find the breaking change in the fastest time possible.

So what does git bisect do?

After you specify any known bad commit and any known good commit, `git bisect` will split the in-between commits in half, and check out a new (nameless) branch in the middle commit to let you check if your feature is broken at that point in time.

Let's say the middle commit still works. You would then let git know that via `git bisect good` command. Then you only have half of the commits left to test.

Git would then split the remaining commits in half and into a new branch (again), letting you test the feature again.

`git bisect` will continue to narrow down your commits in a similar manner, until the first bad commit is found.

Since you divide the number of commits by half on every iteration, you are able to find your bad commits in $\log(n)$ time (which is simply a "[big O](#)" speak for very fast).

The actual commands you need to run to execute the full git bisect flow are:

1. `git bisect start`
let git know to start bisecting.
2. `git bisect good {{some-commit-hash}}`
let git know about a known good commit (i.e. last commit that you made before the vacation).
3. `git bisect bad {{some-commit-hash}}`
let git know about a known bad commit (i.e. the HEAD of the master branch). `git bisect bad HEAD` (HEAD just means the last commit).
4. At this point git would check out a middle commit, and let you know to run your tests.
5. `git bisect bad`
let git know that the feature does not work in currently checked out commit.
6. `git bisect good`
let git know that the feature does work in currently checked out commit.
7. When the first bad commit is found, git would let you know. At this point `git bisect` is done.
8. `git bisect reset`
returns you to the initial starting point of `git bisect` process, (i.e. the HEAD of the master branch).
9. `git bisect log`
log the last git bisect that completed successfully.

You can also automate the process by providing `git bisect` with a script. You can read more here: http://git-scm.com/docs/git-bisect#_bisect_run. ■

Spotlight



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.

The scale of the web and ability to reach billions of people continues to amaze me.

What technology has you excited today?

I am at the point that I am beginning to question all technology and realize that none of it is perfect. That being said, I am still very excited about the web. The scale of the web and ability to reach billions of people continues to amaze me.

What are 1-2 blogs, podcasts, newsletters, etc. that you use to stay on top of the fast-changing and ever evolving industry?

PluralSight and Coursera for longer videos and courses. For podcasts it's JavaScript Jabber, The Changelog and Data Skeptic. I also subscribe to JavaScript Weekly and Hacker Newsletter. I also visit Hacker News fairly often. Last but not least, I am a

member of a Slack room where a few of my former co-workers like to share all kinds of web-related links.

Do you have an Internet resource that you recommend, such as Google Docs? Why do you recommend it?

I've been pretty happy with WorkFlowy lately for outlining my blog ideas and what I want to write about. I also still use Evernote for regular notes. I've been a fan of Gmail for a very long time. Even though I don't love PHP, I think Wordpress is an amazing project with a great community.

What is a personal habit that contributes to your success?

Continuous education and curiosity. I've only been a full-time

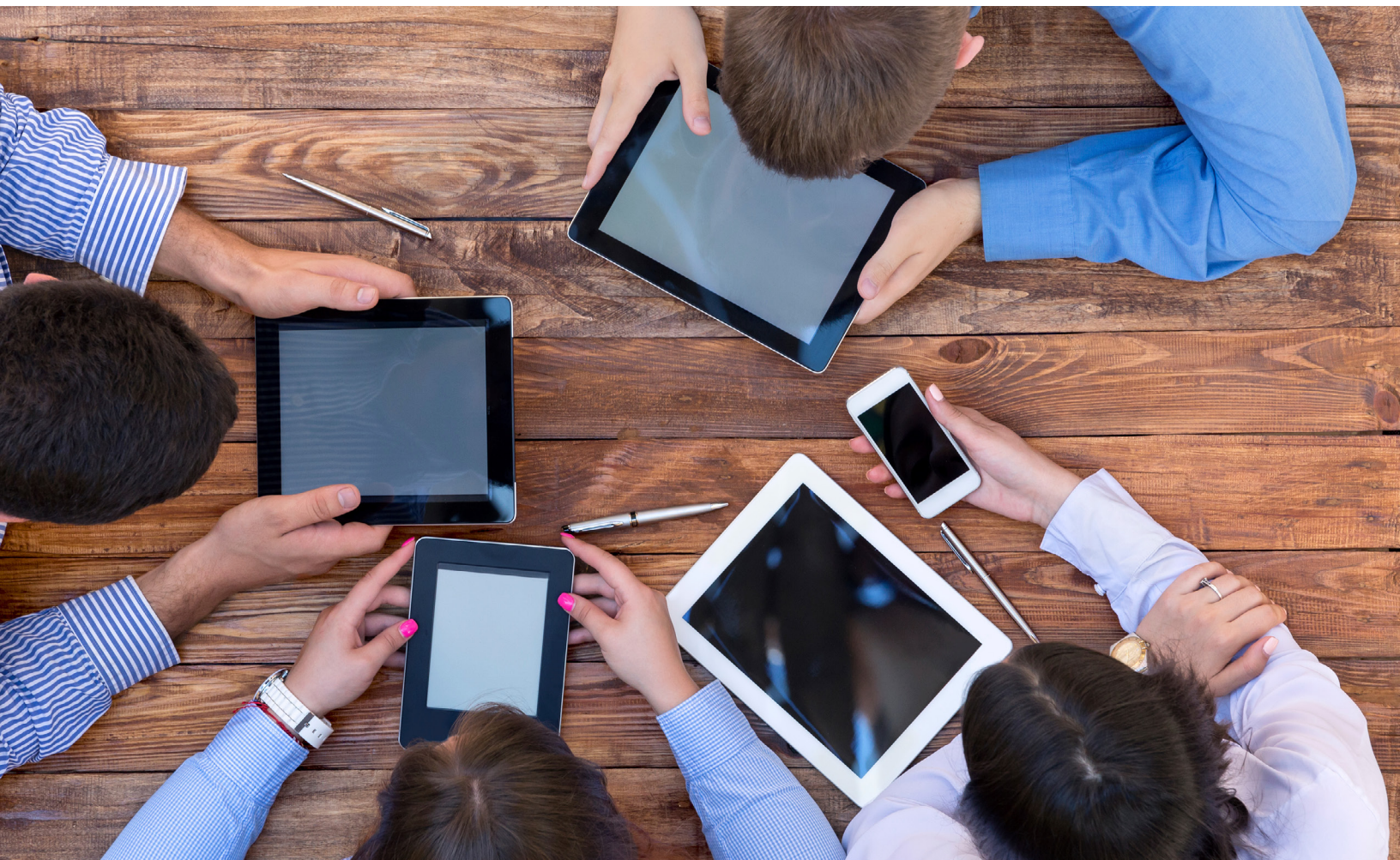
developer for 4 years, so I still very much enjoy what I do, and I'm always eager to learn more. I am a father of two, so I don't have as much time to invest in reading and researching various technologies as I would like. But I try to take full advantage of the free time that I have. I listen to podcasts, watch PluralSight courses, and read books at night on my Kindle.

If there's one book you'd recommend, what is it and why?

For web developers: Effective JavaScript by David Herman.

Where can people find out more about you?

My blog at: alexkras.com. You can also find me on Twitter at [@akras14](https://twitter.com/akras14). ■



It takes all kinds

By JUSTIN ETHEREDGE

After reading a blog post provocatively called [“Today I accept that Rails is yesterday’s software.”](#) I felt the need to reply. I’m not sure why, I’m not going to convince anyone of anything, definitely not the author of that post.

I want to reply because Rails is my current platform of choice, and let’s be honest, seeing a blog post with a title like that getting passed around makes you squirm a little. I think it made me squirm in a good way, because it caused me to step back and evaluate what I’m doing, and why. Enough with the back story, on with the show...

But at the end of the day, they are tools. The only value they have is in what you can create with them. Your tool can be safe, efficient, shiny, but if no one uses it, it is just a dead lump of code.

These tools we have allow us to create amazing things. Many of these tools are quite complex. They try to hide a lot of that from us, but at the end of the day, modern web applications are complex beasts.

Anyone involved in the creation of a web application knows that there are so many moving parts and pieces involved that it is mind boggling. There is no

What are you optimizing for?

Everyone is optimizing for something different. Is Rails a good choice for every shop? No way. Is it a good choice for your shop? I don’t know. What are you optimizing for? Are you a big team that is looking for safe tools that allow you to reliably refactor and give you a lot of compile time safety? Then Rails would be a terrible choice.

Are you a small/medium development shop that needs to be able to stand up and maintain an application easily while leveraging a huge amount of commu-

The framework is there to provide us with a doctrine and the ecosystem that builds up around it is what makes it powerful.

It always comes back to the tools. Everyone blames the tools. And there is good reason for that. Tools empower us. Tools hold us back. Tools excite us. Sharp tools allow us to stab ourselves. Dull tools don’t allow us to get much done. Some tools are optimized for safety. Some are optimized for speed. Some tools are optimized for flexibility, others push you down a happy path.

way you could fit everything you need to build a website into a single framework, even a framework as large as Rails or Django.

And you would never want it that way, everyone needs to do something different. You need a framework that is optimized for what you’re trying to do. The framework is there to provide us with a [doctrine](#) and the ecosystem that builds up around it is what makes it powerful.

nity code/knowledge to get that done? Then a framework like Rails/Django/Laravel might be just the thing you need.

Alternatively, maybe all of your developers know Python, then go with Django! The whole point is that you should pick tools/techniques that fit your team; don’t just grab the newest hippest tool off the shelf unless it solves some very concrete problems you currently have, or

...if you're running into silly problems with your tools then you should be looking for solutions, not throw everything out and reboot.

you are going to feel some pain. Maybe a lot of pain.

And I don't mean the kind of hand-wavy "we can do better" type problems; I'm talking about solid technical problems that you can put your finger on.

Looking for something better, or just different?

In the post I mentioned above it sounds like the author has a lot of frustration with his tooling. I'm sure that is something that everyone has experienced. I can't speak to his exact issues, we don't seem to have many of the same issues, but that doesn't mean they don't exist.

Just as an anecdote though, I have often found that developers working in a framework for years get a 'boiling the frog' moment where they just accept poor ergonomics in their environment for years until someone new comes along and points

them out, or they just lose their mind and flip out.

Once you look for a solution, you'll often find that it was a problem in your workflow all along, because more often than not, broken tools don't stick around in the open source world. Can't say the same thing for other ecosystems though.

The whole point is, don't throw out the baby with the bathwater. These frameworks are complex. Software is complex. Sometimes they don't play well together, but if you're running into silly problems with your tools then you should be looking for solutions, not throw everything out and reboot.

If you're a consultant, then those types of reboots can more easily occur, and are often very lucrative, but they are rarely good for your clients.

My problems aren't your problems

I constantly find myself waxing to other developers about how we, as a group, seem to be stuck in the mindset that all developers have the same problems. The tools and frameworks that Facebook, Twitter, Google and etc. use must be the best, and because I want to be the best, I must use them. Well guess what, you don't have the same problems they do. They have a virtually unlimited amount of developer time; you probably don't.

Would I ever tell you to not use Elixir/Phoenix, Node.js, Revel, Iron, etc...? No, of course not; I don't know what your problems are. But what I would tell you to do is to thoroughly evaluate each one based on your needs. What libraries do you need? What are you willing to write yourself? What is the longevity of the tools? What tools are available to you for deploy-

I tend to be harsh sometimes on developers who always jump on the new shiny tool, but the reality is that we need those people.

ment/hosting/management/troubleshooting? What is the skill-set of your team?

These are all critical questions to ask when evaluating a platform, and if you're not asking them then you probably don't know what you're getting yourself into.

Yesterday? Today? Tomorrow?

Is Rails yesterday's software? Sure. So is PHP. So is C#. So is Python. So is every web framework that has come before. It is mature. It doesn't mean that it isn't today's software, or even tomorrow's. It just means that it has been around for a while.

Are there better platforms out there? Depends on what you're doing. Are there better frameworks for what we do? Probably not. But I don't know you and your problems; you have to make these decisions on your own. Taking ownership of that is always scarier than listen-

ing to a bunch of loud consultants and bloggers proclaiming that they have the future in their pocket.

It takes all kinds

I tend to be harsh sometimes on developers who always jump on the new shiny tool, but the reality is that we need those people (even if I don't want to have to maintain their projects). We need the trailblazers, because if we didn't, there wouldn't be a trail for the rest of us to follow.

If Rails didn't have those people 10 years ago then it wouldn't be anywhere near where it is now. It never would have been able to push through those tough early years where running, deploying, hosting, and maintaining a Rails app was a really painful process.

This is where a lot of these frameworks are now, and that is exciting. I really hope to see many of them mature into stable/reliable platforms and

ecosystems. And one day, for what I do, another framework will pop up that *will* be a better choice than Rails. And I'll probably move on to build amazing things with that framework.

But guess what, when that time comes, there will be somebody writing a blog post telling me my new platform is old news and I'll quietly close my browser, fight the urge to write a blog post, and get back to work. Just like I should have done today. ■

When to rewrite from scratch: autopsy of a failed software

By UMER MANSOOR



It was winter of 2012. I was working as a software developer in a small team at a startup. We had just released the first version of our software to a real corporate customer. The development finished right on schedule. When we launched, I was over the moon and very proud.

It was extremely satisfying to watch the system process couple of millions of unique users a day and send out tens of millions of SMS messages. By summer, the company had real revenue. I got promoted to software manager. We hired new guys. The compa-

ny was poised for growth. Life was great.

And then we made a huge blunder and decided to rewrite the software. From scratch.

Why we felt that rewrite from scratch was needed?

We had written the original system with a gun to our heads. We had to race to the finish line. We weren't having long design discussions or review meetings – we didn't have time for such

things. We would finish up a feature, get it tested quickly and move on to the next. We had a [shared office](#) and I remember software developers at other companies getting into lengthy design and architecture debates and arguing for weeks over design patterns.

Despite agile-on-steroids design, the original system wasn't badly written and generally was well structured. There were some spaghetti code that carried over from the company's previous proof of concept attempts that we left untouched because

it was working and we had no time. But instead of thinking about incremental improvements, we *convinced* ourselves that we needed to rewrite from scratch because:

- The old code was bad and hard to maintain.
- The “monolith java architecture” was inadequate for our future needs of supporting a very large operator with 60 million mobile users and multi-site deployments.
- I *wanted* to try out new, shiny technologies like Apache Cassandra, Virtualization, Binary Protocols, Service Oriented Architecture, etc.

We convinced the entire organization and the board and sadly, we got our wish.

The rewrite journey

The development officially began in spring of 2012 and we set end of January, 2013 as the release date. Because the vision was so grand, we needed even more people. I hired consultants and a couple of remote developers in India.

However, we didn’t fully anticipate the need to maintain the original system in parallel with new development and underestimated customer demands. Remember I said in the beginning we had a real customer?

The customer was one of the biggest mobile operators in South America and once our system had adoption from its users, they started making demands for changes and new features.

So we had to continue updating the original system, albeit

half-heartedly because we were digging its grave. We dodged new feature requests from the customer as much as we can because we were going to throw the old one away anyway. This contributed to delays and we missed our January deadline. In fact, we missed it by 8 whole months!

But let’s skip to the end. When the project was finally finished, it looked great and met all the requirements. Load tests showed that it can easily support over 100 million users. The configuration was centralized and it had a beautiful UI tool to look at charts and graphs.

It was time to go and kill the old system and replace it with the new one... *until the customer said “no” to the upgrade.* It turned out that the original system had gained wide adoption and their users had started relying on it. They wanted absolutely no risks. Long story short, after months of back and forth, we got nowhere. The project was officially doomed.

Lessons learnt

- You should almost never, ever rewrite from scratch. We rewrote for all the wrong reasons. While parts of the code were bad, we could have easily fixed them with refactoring if we had taken time to read and understand the source code that was written by other people. We had genuine concerns about the scalability and performance of the architecture to support more sophisticated business logic, but we could have introduced these changes incrementally.

- Systems rewritten from scratch offer no new value to the user. To the engineering team, new technology and buzzwords may sound cool but they are *meaningless to customers* if they don’t offer new features that the customers need.

- We *missed real opportunities* while we were focused on the rewrite. We had a very basic ‘Web Tool’ that the customer used to look at charts and reports. As they became more involved, they started asking for additional features such as real-time charts, access-levels, etc. Because we weren’t interested in the old code and had no time anyway, we either rejected new requests or did a bad job. As a result, the customer stopped using the tool and insisted on reports by email. Another lost opportunity was an opportunity to build a robust Analytics platform that was *badly* needed.

- I underestimated the effort of maintaining the old system while the new one is in development. We estimated 3-5 requests a month and got 3 times as many.
- We thought our code was harder to read and maintain since we didn’t use proper design patterns and practices that other developers spent days discussing. It turned out that most professional code I have seen in larger organizations is 2x worse than what we had. So we were dead wrong about that.

When is rewrite the answer?

[Joel Spolsky made strong arguments against rewrite](#) and suggests that one should never do it. I'm not so sure about it. Sometimes incremental improvements and refactoring are very difficult and the only way to *understand* the code is to rewrite it. Plus software developers love to write code and create new things – it's boring to read someone else's code and try to understand their code and their 'mental abstractions'. But good programmers are also good maintainers.

If you want to rewrite, do it for the right reasons and plan properly for the following:

- The old code will still need to be maintained, in some cases, long after you release the new version. Maintaining two versions of code will require huge efforts and you need to ask yourself if you have enough time and resources to justify that based on the size of the project.
- Think about losing other opportunities and prioritize.
- Rewriting a big system is riskier than smaller ones. Ask yourself if you can incrementally rewrite. We switched to a new database, became a 'Service Oriented Architecture' and changed our protocols to binary, all at the same time. We could have introduced each of these changes incrementally.
- Consider the developers' bias. When developers want

to learn a new technology or language, they want to write some code in it. While I'm not against it and it's a sign of a good environment and culture, you should take this into consideration and weigh it against risks and opportunities.

Michael Meadows made [excellent observations](#) on when the "BIG" rewrite becomes necessary:

Technical

- The coupling of components is so high that changes to a single component cannot be isolated from other components. A redesign of a single component results in a cascade of changes not only to adjacent components, but indirectly to all components.
- The technology stack is so complicated that future state design necessitates multiple infrastructure changes. This would be necessary in a complete rewrite as well, but if it's required in an incremental redesign, then you lose that advantage.
- Redesigning a component results in a complete rewrite of that component anyway, because the existing design is so fubar that there's nothing worth saving. Again, you lose the advantage if this is the case.

Political

- The sponsors cannot be made to understand that an incremental redesign requires a long-term commitment to the project. Inevitably, most organiza-

tions lose the appetite for the continuing budget drain that an incremental redesign creates. This loss of appetite is inevitable for a rewrite as well, but the sponsors will be more inclined to continue, because they don't want to be split between a partially complete new system and a partially obsolete old system.

- The users of the system are too attached to their "current screens." If this is the case, you won't have the license to improve a vital part of the system (the front-end). A redesign lets you circumvent this problem, since they're starting with something new. They'll still insist on getting "the same screens," but you have a little more ammunition to push back. Keep in mind that the total cost of redesigning incrementally is always higher than doing a complete rewrite, but the impact to the organization is usually smaller. In my opinion, if you can justify a rewrite, and you have superstar developers, then do it.

Abandoning working projects is dangerous and we wasted an enormous amount of money and time duplicating working functionality we already had, rejected new features, irritated the customer and delayed ourselves by years. If you are embarking on a rewrite journey, all the power to you, but make sure you do it for the right reasons, understand the risks and plan for it. ■

Spotlight



Umer Mansoor

Umer Mansoor is a software developer, living in San Francisco, CA. He currently works for [Glu Mobile](#) as Platform Manager, building a cloud gaming backend. He previously served as the *Head of Software* for Starscriber where he built high performance telecommunications software. He blogs at [CodeAhoy.com](#).

Machine learning hands down. Specifically predictive analytics to forecast the efficacy of mobile campaigns.

What technology has you excited today?

Machine learning hands down. Specifically predictive analytics to forecast the efficacy of mobile campaigns. Google recently open sourced their machine learning library, [TensorFlow](#), and it looks very promising.

What are 1-2 blogs, podcasts, newsletters, etc. that you use to stay on top of the fast-changing and ever evolving industry?

I enjoy [Scott Hanselman's blog](#), even though it is heavy on Microsoft technologies. I like his writing style and he covers interesting topics. I highly encourage everyone to read [Joel's blog](#) which isn't updat-

ed regularly these days but is full of extremely useful insight into software development and management. To stay up to date on recent events, I follow [Hacker News](#) and TechCrunch.

Do you have an Internet resource that you recommend, such as Google Docs? Why do you recommend it?

I couldn't recommend [GitHub](#) more. I've been using it for the last 6 years and it's such a cool site and a great community. Recently, I have been pasting lines of source code into GitHub's search bar to find answers to my questions in the form of code.

What is a personal habit that contributes to your success?

Don't really consider myself

successful :), but I would say patience, staying hungry for knowledge and taking full responsibility for my actions have helped me grow. I also try to read at least a couple of books a month on technical and leadership subjects.

If there's one book you'd recommend, what is it and why?

It's tough. But if I had to pick one, I'd go for [Peopleware: Productive Projects and Teams](#). It's an absolute gem and focuses on building and growing productive software teams.

Where can people find out more about you?

I blog on [CodeAhoy.com](#). I'm also on Twitter [@codeahoy](#). ■

Clojure, the good parts

By ALLEN ROHNER



I kid, somewhat. This title is of course based on Douglas Crockford's seminal work, *JavaScript, The Good Parts*, which demonstrated that JavaScript isn't a terrible language, if only you avoid the icky parts.

This is my heavily opinionated guide to what a "good" production Clojure app looks like in 2016.

I love Clojure, and I've been using it pretty much exclusively since 2009. In the 7 years I've been using it professionally, a consensus has started to form

around what 'the good parts' of Clojure looks like. This article is my take on what a good app looks like.

In several places I'll recommend specific libraries. While I recommend that specific library, there are often competitor libraries with similar functionality, and most of the time getting the functionality is more important than that exact library, so feel free to substitute, as long as you're getting the same benefits.

To make my biases explicit, I mostly write webapps and data analysis on servers in the cloud.

If you use Clojure in significantly different applications, these recommendations might not apply to you.

Recommendations are split into several categories: core language, libraries and deployment.

I'll also try to avoid uncontroversial advice that's been covered elsewhere, like 'avoid flatten' and 'don't (def) anywhere but the top-level', etc.

Core

Avoid binding

`clojure.core/binding` is a code-smell. Avoid it. In almost all cases, `binding` is used to sneak extra arguments into a function, reducing its referential transparency. Instead, pass all arguments explicitly into a function, which improves purity and readability, and avoid issues with multi-threaded behavior. If you're dealing with optional arguments, consider an extra arity of the function with an optional `map`.

Avoid agents

It's rare to see a problem that can be precisely modeled by agents. Most of the instances I've used them have been as hacks to get the specific multi-threading properties I want. These days, `core.async` can be used to more explicitly model the desired behavior.

Avoid STM

Like `agent`, it's rare to see a problem that can be properly modeled by STM, in production. Most production apps will need to persist data in a database. It's rare that you can do 'real' work in an STM commit that shouldn't also be stored in the DB. Coordinating commits to both STM and

the DB is error prone (The Two Generals Problem).

Failing to commit to the DB is a Major Error, while failing to commit to the STM is typically less so, because the Clojure process can be restarted and loaded from the DB as the source of truth. Therefore, the easiest way to avoid coordination problems is to just have one source of truth, the database.

Use atoms, sparingly

By process of elimination, because I've just recommended avoiding binding, agent and STM, that leaves only one core mutable-state construct, the atom. Atoms are good, and should be used, but treat them like macros: only use them when they're the only tool available.

Avoid global mutable state

In 2009, I would not have believed how little global mutable state is used in my applications. The vast majority of your state should be in the DB, or a queue, or Redis. I'm now at the point where

```
(def foo (atom ...))
```

is a code smell. Most of the time when using atoms, they should not be `def`-ed at the top level.

They should be returned from constructor functions, or stored in Component. This means that you should end up with only one piece of global state, the system.

Avoid pmap

`pmap` has been subtly broken since chunked seqs were introduced to the language, and it's parallelism is not as high as promised. Use reducers + `fork/join`, or `core.async`'s `pipeline`, or raw Java concurrency instead.

Avoid metadata

It's not always obvious which functions will preserve metadata, and which won't. As a Clojure user since pre-1.0, I've long stopped caring about "oh, `assoc` in 1.x didn't preserve metadata, but it did in 1.(inc x)". Metadata is nice to have, for introspection and working at the repl. As a bright line though, metadata should never be used to control program behavior.

Exercise caution with futures

Futures are great, but they're a potential footgun. There are a few things to watch out for.

First, always always always use `java.lang.Thread/setDefaultUncaughtExceptionHandler`. It looks something the code below.

```
(Thread/setDefaultUncaughtExceptionHandler
  (reify
    Thread$UncaughtExceptionHandler
    (uncaughtException [this thread throwable]
      (errorf throwable "Uncaught exception %s on thread %s" throwable thread))))
(Thread/setDefaultUncaughtExceptionHandler
  (reify
    Thread$UncaughtExceptionHandler
    (uncaughtException [this thread throwable]
      (errorf throwable "Uncaught exception %s on thread %s" throwable thread))))
```

▲ `java.lang.Thread/setDefaultUncaughtExceptionHandler`

This guarantees that if your future throws an exception (and it will, eventually), that will be logged and recorded somewhere.

Second, always consider what you're doing inside the future, and what would happen to the system if power was lost while the future was running. Imagine you're running an e-commerce shop, and a customer buys something, and then we send them a confirmation email in a future. The pseudo-code would look like:

```
(charge-credit-card! user
transaction)
(future (send-confirmation-email
transaction))
```

If power dies while the future is running, the customer might not get their email. Not ideal. In general, futures are a place where transactional guarantees are likely to be lost. In almost all cases, if you're using a future for side effects (sending email, calling 3rd party APIs, etc.), consider whether that action should go into a job queue. Use futures for querying multiple data-sources in parallel, and use durable queues for performing side-effects asynchronously.

Libraries

Some libraries I like, in no particular order. Note that these recommendations are about libraries to use in apps you write. I fully agree with [Stuart Sierra's position on library dependencies](#).

Use Component

Seriously, [Component](#) is the single biggest improvement you can make to a large Clojure codebase. It fixes testing. It

fixes app startup. It fixes configuration. It fixes staging vs. production woes. CircleCI's unit tests were a mess because of a huge amount of `with-redefs` and `binding` to get testing behavior right. If the component under test used Component, there would be no need for redefining at all. Literally thousands of lines of test fixtures would get dramatically simpler.

Use Schema

[Schema All The Things](#). As Emacs tells me, "Code never lies, comments [and docstrings] sometimes do". Schema can reduce the amount of doc strings necessary on a function, and schema-as-doc-strings are more likely to be correct, because they're executable (and therefore, verified). They completely eliminate that annoyance in doc strings where the doc states 'this argument takes a Foo', without every specifying what a Foo is. It can be used to handle input validation from 3rd parties. It can be used to prove your tests are valid (i.e. passing valid data to the function under test).

Use core.async

I've mentioned it several times in this post already, but `core.async` should be your default choice for most complex multi-threading tasks.

Use Timbre

Clojure programmers love to make fun of the horrors that are `j.u.logging`, `logback` and `SLF4J`. Just dump it all, and use Timbre. [Timbre](#) is Clojure[script]-only logging, so it has no Java vestigial tails, no XML, and no class-path weirdness.

Timbre plays well with Component, so when you have separate component systems, one for development, one for

test, etc., they can use different logging policies, *while both are running in the same process at the same time*. Production systems log to Splunk or Loggly or what have you, while tests only log to stdout, etc.

Use clj-time

`clj-time` is essential for readable code that interacts with dates. Let's say you have a DB query that takes a start and end date:

```
(query (-> 7 time/days time/ago)
(time/now))
```

`clj-time` wraps Joda, which is excellent. Libraries for wrapping Java 8 Instant are probably good too, I haven't used them though.

Use Clojure.test

[Your tests don't need a cutesy DSL. Your tests especially do not need eight different cutesy DSLs.](#)

Yes, `clojure.test` has warts. But it has well-known, immutable warts. Not having new releases is a *feature* of testing libraries.

New testing libraries are fun and exciting, until you find a bug in your test library. `clojure.test` works, and is battle tested in a way that no other clojure testing library is. I've shipped bugs to production on code that I *thought* was tested, because I didn't understand the way the testing DSL works. I've seen infinite loops in macros shipped by the testing library, because the DSL was too cutesy and complex.

Never again, use the simplest thing that solves the problem.

Don't wrap clj-http

This is kind of a meta point. Don't use libraries for 3rd party APIs that provide no value on top of your HTTP client. For ex-

```
(def stripe-api-endpoint "https://api.stripe.com")

(defn stripe-api [auth path args]
  (http/request (str stripe-api-endpoint path)
    (merge {:basic-auth auth} args)))
```

▲ Using `clj-http` directly

ample, interacting with Stripe's API is easy. Just write one helper function that `merges` in whatever authentication the service needs, and then just use `clj-http` directly (see above).

That is literally the entirety of what you need from a stripe API library. A library must provide significant value over just writing your own, and most libraries that wrap HTTP rest APIs don't. A few provide nice features, like HTTP pagination, but that code isn't that difficult to write. (Actually, it'd be interesting to see if there are REST patterns that can be abstracted into a single `clj-http` higher-order-function library).

Deployment

Build a single artifact

Whatever you're building, releases should consist of a single artifact. A `.jar` or `.war` or docker container or AMI or whatever it is, and it should be self-contained, and reproducible. Your process should not resolve or download dependencies at run-time. Starting a production server should be simple, reliable and repeatable. Reliable, meaning "very little chance of failing" and repeatable, meaning "starting a

server on one day and starting on another day should have bit-for-bit identical code".

You *will* need to rollback, because you will deploy bad code at some point. You need to know that the prior version still works, because you're already rolling back production because of an error, and you really don't want your problems to get worse.

We improve repeatability by avoiding downloading or resolving anything mutable. Any scheme involving `git` or `lein` or `apt-get` when turning on a server is immediately suspect, and `npm` is right out! Downloading previously-resolved deps is better, but still not as good as baking the deps directly into your artifact. That guarantees that even if there is an `npm`-style disaster, your old build still works.

Avoid writing lein plugins

During the `lein 1.x` days, plugins were the standard way of adding functionality to your build. The combination of `lein run` and `:aliases` has changed that. Whenever possible, write a standard clojure function, then add it to `:aliases` in your project.clj:

```
{:aliases "build-foo" ["run" "-m"
  "rasterize.build.foo/build-foo"]}
```

Typically, the only reason you'll need a plugin is to control the classpath.

Standard functions are much easier to write, test, run, and chain together. Chaining clojure functions is just `do`. Chaining lein plugins is `lein do`, which is slower and awkward, and can't (easily) be done from the repl or other functions.

Prefer Clojure over build tools

In the last section, I said 'prefer Clojure functions over lein plugins'. Now I'm also saying 'prefer Clojure functions over bash and most CLI tools'. Obviously some allowance needs to be made for tools that are very hard to replace, but your asset fingerprinting probably isn't one of them. Clojure is an incredibly powerful language. Most of the time, you'll get more power, flexibility and insight into your build process if you use clojure code over command line tools.

For example, Rasterize's asset compilation, fingerprinting and deploy to S3 are all standard Clojure functions, using `less4j` `java.io` and AWS libraries. The Rasterize static site generation (`.md` to `.html`, etc) is all clojure. These are functions that I can run from the repl, and debug using standard clojure tools. I haven't used `boot` in anger yet, but I'm supportive of its philosophy.

Conclusion

And there we have it. A haphazard collection of poorly justified opinions. Let me know if you agree or disagree on [Hacker News](#). ■



food bit *

Credit: By Matyáš Havel ([Own work](#)) [[CC BY-SA 3.0](#)], via Wikimedia Commons

Akutaq

Also known as Eskimo ice cream, akutaq, is a traditional food of the indigenous people of Alaska. Made from animal fat (whale, reindeer or seal), berries and ground fish, akutaq is a highly nutritious survival food of the Arctic. It is made by whipping the fat until it is light and airy, and mixing in the berries and fish. The concoction is left to freeze in the cold until it resembles ice cream. This sweet and tart treat is usually served at celebrations and gatherings.

* 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.