Andrew Ng on Life, Creativity, and Failure

HACKERMONTHLY

Issue 67 December 2015

Curator
Lim Cheng Soon

Contributors
Nico Pitney
Adrian Mouat
Peteris Caune
Nate Bartley
Chris Beams
Leonie Watson
Robleh Jama

Proofreader
Shelly Davis

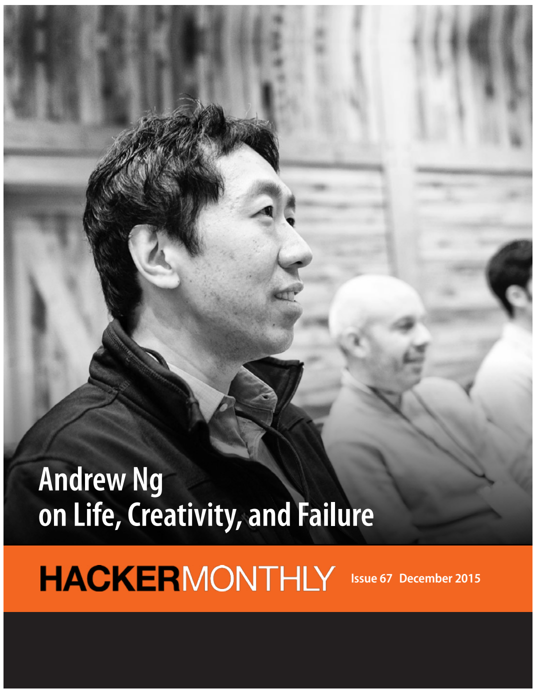
Printer
Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Photo by: Christopher Michel [flickr.com/photos/cmichel67/13181917214]

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-67*

Photo credit: Christopher Michel [flickr.com/photos/cmichel67/13181917214]

Andrew Ng on Life, Creativity, and Failure

By NICO PITNEY

Here's a list of universities with arguably the greatest computer science programs: Carnegie Mellon, MIT, UC Berkeley, and Stanford. These are the same places, respectively, where Andrew Ng received his bachelor's degree, his master's, his Ph.D., and has taught for 12 years.

Ng is an icon of the artificial intelligence world with the pedigree to match, and he is not yet 40 years old. In 2011, he founded Google Brain, a deep-learning research project supercharged by Google's vast stores of computing power and data. Delightfully, one of its most important achievements came when computers analyzing scores of YouTube screenshots were able to recognize a cat. (The New York Times' headline: "How Many Computers to Identify a Cat? 16,000.") As Ng explained, "The remarkable thing was that [the system] had discovered the concept of a cat itself. No one had ever told it what a cat is. That was a milestone in machine learning."

Ng exudes a cheerful but profound calm. He happily discusses the various mistakes and failures of his career, the papers he read but didn't understand. He wears identical blue oxford shirts each and every day. He is blushing but proud when a colleague mentions his adorable robot-themed engagement photo shoot with his now-wife, a surgical roboticist named Carol Reiley.

One-on-one, he speaks with a softer voice than anyone you know, though this has not hindered his popularity as a lecturer. In 2011, when he posted videos from his own Stanford machine learning course on the web, over 100,000 people registered. Within a year, Ng had co-founded Coursera, which is today the largest provider of open online courses. Its partners include Princeton and Yale, top schools in China and across Europe. It is a for-profit venture, though all classes are accessible for free. "Charging for content would be a tragedy," Ng has said.

Then, last spring, a shock. Ng announced he was departing Google and stepping away from day-to-day involvement at Coursera. The Chinese tech giant Baidu was establishing an ambitious $300 million research lab devoted to artificial intelligence just down the road from Google's Silicon Valley headquarters, and Andrew Ng would head it up.

At Baidu, as before, Ng is trying to help computers identify audio and images with incredible accuracy, in realtime. (On Tuesday, Baidu announced it had achieved the world's best results on a key artificial intelligence benchmark related to image identification, besting Google and Microsoft.) Ng believes speech recognition with 99 percent accuracy will spur revolutionary changes to how humans interact with computers, and how operating systems are designed. Simultaneously, he must help Baidu work well for the millions of search users who are brand new to digital life. "You get queries [in China] that you just wouldn't get in the United States," Ng explained. "For example, we get queries like, "Hi Baidu, how are you? I ate noodles at a corner store last week and they were delicious. Do you think they're on sale this weekend?" That's the query." Ng added: "I think we make a good attempt at answering."

Elon Musk and Stephen Hawking have been sounding alarms over the potential threat to humanity from advanced artificial intelligence. Andrew Ng has not. "I don't work on preventing AI from turning evil for the same reason that I don't work on combating overpopulation on the planet Mars," he has said. AI is many decades away (if not longer) from achieving something akin to consciousness, according to Ng. In the meantime, there's a far more urgent problem. Computers enhanced by machine learning are eliminating jobs long done by humans. The trend is only accelerating, and Ng frequently calls on policymakers to prepare for the socioeconomic consequences.

At Baidu's new lab in Sunnyvale, Calif., we spoke to Andrew Ng for Sophia, a HuffPost project to collect life lessons from fascinating people. He explained why he thinks "follow your passion" is terrible career advice and he shared his strategy for teaching creativity; Ng discussed his failures and his helpful habits, the most influential books he's read, and his latest thoughts on the frontiers of AI.

You recently said, "I've seen people learn to be more creative." Can you explain?

The question is, how does one create new ideas? Is it those unpredictable lone acts of genius, people like Steve Jobs, who are special in some way? Or is it something that can be taught and that one can be systematic about?

I believe that the ability to innovate and to be creative are teachable processes. There are ways by which people can systematically

“I don’t know how the human brain works but it’s almost magical: when you read enough or talk to enough experts, when you have enough inputs, new ideas start appearing.”

innovate or systematically become creative. One thing I’ve been doing at Baidu is running a workshop on the strategy of innovation. The idea is that innovation is not these random unpredictable acts of genius, but that instead one can be very systematic in creating things that have never been created before.

In my own life, I found that whenever I wasn’t sure what to do next, I would go and learn a lot, read a lot, talk to experts. I don’t know how the human brain works but it’s almost magical: when you read enough or talk to enough experts, when you have enough inputs, new ideas start appearing. This seems to happen for a lot of people that I know.

When you become sufficiently expert in the state of the art, you stop picking ideas at random. You are thoughtful in how to select ideas, and how to combine ideas. You are thoughtful about when you should be generating many ideas versus pruning down ideas.

Now there is a challenge still — what do you do with the new ideas, how can you be strategic in how to advance the ideas to build useful things? That’s another whole piece.

Can you talk about your information diet, how you approach learning?

I read a lot and I also spend time talking to people a fair amount. I think two of the most efficient ways to learn, to get information, are reading and talking to experts. So I spend quite a bit of time doing both of them. I think I have just shy of a thousand books on my Kindle. And I’ve probably read about two-thirds of them.

At Baidu, we have a reading group where we read about half a book a week. I’m actually part of two reading groups at Baidu, each of which reads about half a book a week. I think I’m the only one who’s in both of those groups [laughter]. And my favorite Saturday afternoon activity is sitting by myself at home reading.

Let me ask about your early influences. Is there something your parents did for you that many parents don’t do that you feel had a lasting impact on your life?

I think when I was about six, my father bought a computer and helped me learn to program. A lot of computer scientists learned to program from an early age, so it’s probably not that unique, but I think I was one of the ones that was fortunate to have had a computer and could learn to start to program from a very young age.

Unlike the stereotypical Asian parents, my parents were very laid back. Whenever I got good grades in school, my parents would make a fuss, and I actually found that slightly embarrassing. So I used to hide them. [Laughter] I didn’t like showing my report card to my parents, not because I was doing badly but because of their reaction.

I was also fortunate to have gotten to live and work in many different places. I was born in the U.K., raised in Hong Kong and Singapore, and came to the U.S. for college. Then for my own studies, I have degrees from Carnegie Mellon, MIT, and Berkeley, and then I was at Stanford.

I was very fortunate to have moved to all these places and gotten to meet some of the top people. I interned at AT&T Bell Labs when it existed, one of the top labs, and then at Microsoft Research. I got to see a huge diversity of points of view.

“The world has an infinite supply of interesting problems. The world also has an infinite supply of important problems. I would love for people to focus on the latter.”

Is there anything about your education or your early career that you would have done differently? Any lessons you’ve learned that people could benefit from?
I wish we as a society gave better career advice to young adults. I think that “follow your passion” is not good career advice. It’s actually one of the most terrible pieces of career advice we give people.

If you are passionate about driving your car, it doesn’t necessarily mean you should aspire to be a race car driver. In real life, “follow your passion” actually gets amended to, “Follow your passion of all the things that happen to be a major at the university you’re attending.”

But often, you first become good at something, and then you become passionate about it. And I think most people can become good at almost anything.

So when I think about what to do with my own life, what I want to work on, I look at two criteria. The first is whether it’s an opportunity to learn. Does the work on this project allow me to learn new and interesting and useful things? The second is the potential impact. The world has an infinite supply of interesting problems. The world also has an infinite supply of important problems. I would love for people to focus on the latter.

I’ve been fortunate to have repeatedly been able to find opportunities that had a lot of potential for impact and also gave me fantastic opportunities to learn. I think young people optimizing for these two things will often have the best careers.

Our team here has a mission of developing hard AI technologies, advanced AI technologies that let us impact hundreds of millions of users. That’s a mission I’m genuinely excited about.

Do you define importance primarily by the number of people who are impacted?
No, I don’t think the number is the only thing that’s important. Changing hundreds of millions of people’s lives in a significant way, I think that’s the level of impact that we can reasonably aspire to. That is one way of making sure we do work that isn’t just interesting, but that also has an impact.

You’ve talked previously about projects of yours that have failed. How do you respond to failure?
Well, it happens all the time, so it’s a long story. [Laughter] A few years ago, I made a list in Evernote and tried to remember all the projects I had started that didn’t work out, for whatever reason. Sometimes I was lucky and it worked out in a totally unexpected direction, through luck rather than skill.

But I made a list of all the projects I had worked on that didn’t go anywhere, or that didn’t succeed, or that had much less to show for it relative to the effort that we put into it. Then I tried to categorize them in terms of what went wrong and tried to do a pretty rigorous post mortem on them.

So, one of these failures was at Stanford. For a while we were trying to get aircraft to fly in formation to realize fuel savings, inspired by geese flying in a V-shaped formation. The aerodynamics are actually pretty solid. So we spent about a year working on making these aircraft fly autonomously. Then we tried to get the airplanes to fly in formation.

But after a year of work, we realized that there is no way that we could control the aircraft with sufficient accuracy to realize fuel savings. Now, if at the start of the project we had thought through the position requirements, we would have realized that with the small aircraft we were using, there is just no way we could do it. Wind gusts will blow you around far more than the precision needed to fly the aircraft in formation.

So one pattern of mistakes I've made in the past, hopefully much less now, is doing projects where you do step one, you do step two, you do step three, and then you realize that step four has been impossible all along. I talk about this specific example in the strategy innovation workshop I talked about. The lesson is to de-risk projects early.

I've become much better at identifying risks and assessing them earlier on. Now when I say things like, "We should de-risk a project early," everyone will nod their head because it's just so obviously true. But the problem is when you're actually in this situation and facing a novel project, it's much harder to apply that to the specific project you are working on.

The reason is these sorts of research projects, they're a strategic skill. In our educational system we're pretty good at teaching facts and procedures, like recipes. How do you cook spaghetti bolognese? You follow the recipe. We're pretty good at teaching facts and recipes.

But innovation or creativity is a strategic skill where every day you wake up and it's a totally unique context that no one's ever been in, and you need to make good decisions in your completely unique environment. So as far as I can tell, the only was we know way to teach strategic skills is by example, by seeing tons of examples. The human brain, when you see enough examples, learns to internalize those rules and guidelines for making good strategic decisions.

Very often, what I find is that for people doing research, it takes years to see enough examples and to learn to internalize those guidelines. So what I've been experimenting with here is to build a flight simulator for innovation strategy. Instead of having everyone spend five years before you see enough examples, to deliver many examples in a much more compressed time frame.

Just as in a flight simulator, if you want to learn to fly a 747, you need to fly for years, maybe decades, before you see any emergencies. But in a flight simulator, we can show you tons of emergencies in a very compressed period of time and allow you to learn much faster. Those are the sorts of things we've been experimenting with.

When this lab first opened, you noted that for much of your career you hadn't seen the importance of team culture, but that you had come to realize its value. Several months in, is there anything you've learned about establishing the right culture?

A lot of organizations have cultural documents like, "We empower each other," or whatever. When you say it, everyone nods their heads, because who wouldn't want to empower your teammates. But when they go back to their desks five minutes later, do they actually do it? It's difficult for people to bridge the abstract and the concrete.

At Baidu, we did one thing for the culture that I think is rare. I don't know of any organization that has done this. We created a quiz that describes to employees specific scenarios — it says, "You're in this situation and this happens. What do you do: A, B, C, or D?"

No one has ever gotten full marks on this quiz the first time out. I think the quiz interactivity, asking team members to apply specifics to hypothetical scenarios, has been our way of trying to connect the abstract culture with the concrete; what do you actually do when a teammate comes to you and does this thing?

What are some books that had a substantial impact on your intellectual development?

Recently I've been thinking about the set of books I'd recommend to someone wanting to do something innovative, to create something new.

The first is "Zero to One" by Peter Thiel, a very good book that gives an overview of entrepreneurship and innovation.

We often break down entrepreneurship into B2B ("business to business," i.e., businesses whose customers are other businesses) and B2C ("business to consumer"). For B2B, I recommend "Crossing the Chasm." For B2C, one of my favorite books is "The Lean Startup," which takes a narrower view but it gives one specific tactic for innovating quickly. It's a little narrow but it's very good in the area that it covers.

Then to break B2C down even further, two of my favorites are "Talking to Humans," which is a very short book that teaches you how to develop empathy for users you want to serve by talking to them. Also, "Rocket Surgery Made Easy." If you want to build products that are important, that users care about, this teaches you different tactics for learning about users, either through user studies or by interviews.

Then finally there is "The Hard Thing about Hard Things." It's a bit dark but it does cover a lot of useful territory on what building an organization is like.

For people who are trying to figure out career decisions, there's a very interesting one: "So Good They Can't Ignore You." That gives a valuable perspective on how to select a path for one's career.

Do you have any helpful habits or routines?

I wear blue shirts every day, I don't know if you know that. [laughter] Yes. One of the biggest levers on your own life is your ability to form useful habits.

When I talk to researchers, when I talk to people wanting to engage in entrepreneurship, I tell them that if you read research papers consistently, if you seriously study half a dozen papers a week and you do that for two years, after those two years you will have learned a lot. This is a fantastic investment in your own long term development.

But that sort of investment, if you spend a whole Saturday studying rather than watching TV, there's no one there to pat you on the back or tell you you did a good job. Chances are what you learned studying all Saturday won't make you that much better at your job the following Monday. There are very few, almost no short-term rewards for these things. But it's a fantastic long-term investment. This is really how you become a great researcher, you have to read a lot.

People that count on willpower to do these things, it almost never works because willpower peters out. Instead I think people that are into creating habits — you know, studying every week, working hard every week — those are the most important. Those are the people most likely to succeed.

For myself, one of the habits I have is working out every morning for seven minutes with an app. I find it much easier to do the same thing every morning because it's one less decision that you have to make. It's the same reason that my closet is full of blue shirts. I used to have two color shirts actually, blue and magenta. I thought that's just too many decisions. [Laughter] So now I only wear blue shirts.

You've urged policymakers to spend time thinking about a future where computing and robotics have eliminated some substantial portion of the jobs people have now. Do you have any ideas about possible solutions?

It's a really tough question. Computers are good at routine repetitive tasks. Thus far, the main things that computers have been good at automating are tasks where you kind of do the same thing day after day.

Now this can be at multiple points on the spectrum. Humans work on an assembly line, making the same motion for months on end, and now robots are doing some of that work. A midrange challenge might be truck-driving. Truck drivers do very similar things day after day, so computers are trying to do that too. It's harder than most people think, but automated driving might happen in the next decade or so, we don't know. Then, even higher-end things, like some radiologists read the same types of x-rays over and over each day. Again, computers may have traction in those areas.

But for the social tasks which are non-routine and non-repetitive, those are the tasks that humans will be better at than computers for quite a period of time, I think. In many of our jobs we do different things every day. We meet different people, we have to arrange different things, solve problems differently. Those things are relatively difficult for computers to do, for now.

The challenge that faces us is that, when the U.S. transformed from an agricultural to a manufacturing and services economy, we had people move from one routine task, such as farming, to a different routine task, such as manufacturing or working call service centers. A large fraction of the population has made that transition, so they've been okay, they've found other jobs. But many of their jobs are still routine and repetitive.

The challenge that faces us is to find a way to scalably teach people to do non-routine non-repetitive work. Our education system, historically, has not been good at doing that at scale. The top universities are good at doing that for a relatively modest fraction of the population. But a lot of our population ends up doing work that is important but also routine and repetitive. That's a challenge that faces our educational system.

I think it can be solved. That's one of the reasons why I've been thinking about teaching innovation strategy, teaching creativity strategy. We need to enable a lot of people to do non-routine, non-repetitive tasks. These tactics for teaching innovation and creativity, these flight simulators for innovation, could be one way to get there. I don't think we've figured out yet how to do it, but I'm optimistic it can be done.

You've said, "Engineers in China work much harder than the average Silicon Valley engineer. Engineers in Silicon Valley at startups work really hard. At mature companies, I don't see the same intensity as you do in startups and at Baidu." Why do you think that is?

I don't know. I think the individual engineers in China are great. The individual engineers in Silicon Valley are great. The difference I think is the company. The teams of engineers at Baidu tend to be incredibly nimble.

There is much less appreciation for the status quo in the Chinese internet economy and I think there's a much bigger sense that all assumptions can be challenged and everything is up for grabs. The Chinese internet ecosystem is very dynamic. Everyone sees huge opportunity, everyone sees massive competition. Stuff changes all the time. New inventions arise, and large companies will one day suddenly jump into a totally new business sector.

To give you an idea, here in the United States, if Facebook were to start a brand new web search engine, that might feel like a slightly strange thing to do. Why would Facebook build a search engine? It's really difficult. But that sort of thing is much more thinkable in China, where there is more of an assumption that there will be new creative business models.

This seems to suggests a different management culture, where you can make important decisions quickly and have them be intelligent and efficient and not chaotic. Is Baidu operating in a unique way that you feel is particularly helpful to its growth?

Gosh, that's a good question. I'm trying to think what to point to. I think decision making is pushed very far down in the organization at Baidu. People have a lot of autonomy, and they are very strategic. One of the things I really appreciate about the company, especially the executives, is there's a very clear-eyed view of the world and of the competition.

When executives meet, and the way we speak with the whole company, there is a refreshing absence of bravado. The statements that are made internally — they say, "We did a great job on that. We're not so happy with those things. This is going well. This is not going well. These are the things we think we should emphasize. And let's do a post-mortem on the mistakes we made." There's just a remarkable lack of bravado, and I think this gives the organization great context on the areas to innovate and focus on.

You're very focused on speech recognition, among other problems. What are the challenges you're facing that, when solved, will lead to a significant jump in the accuracy of speech recognition technology?

We're building machine learning systems for speech recognition. Some of the machine learning technologies we're using now have been around for decades. It was only in the last several years that they've really taken off.

Why is that? I often make an analogy to building a rocket ship. A rocket ship is a giant engine together with a ton of fuel. Both need to be really big. If you have a lot of fuel and a tiny engine, you won't get off the ground. If you have a huge engine and a tiny amount of fuel, you can lift up, but you probably won't make it to orbit. So you need a big engine and a lot of fuel.

The reason that machine learning is really taking off now is that we finally have the tools to build the big rocket engine — that is giant computers, that's our rocket engine. And the fuel is the data. We finally are getting the data that we need.

The digitization of society creates a lot of data and we've been creating data for a long time now. But it was just in the last several years we've been finally able to build big enough rocket engines to absorb the fuel. So part of our approach, not the whole thing, but a lot of our approach to speech recognition is finding ways to build bigger engines and get more rocket fuel.

For example, here is one thing we did, a little technical. Where do you get a lot of data for speech recognition? One of the things we did was we would take audio data. Other groups use maybe a couple thousand hours of data. We use a hundred thousand hours of data. That is much more rocket fuel than what you see in academic literature.

Then one of the things we did was, if we have an audio clip of you saying something, we would take that audio clip of you and add background noise to it, like a clip recorded in a cafe. So we synthesize an audio clip of what you would sound like if you were speaking in a cafe. By synthesizing your voice against lots of backgrounds, we just multiply the amount of data that we have. We use tactics like that to create more data to feed to our machines, to feed to our rocket engines.

One thing about speech recognition: most people don't understand the difference between 95 and 99 percent accurate. Ninety-five percent means you get one-in-20 words wrong. That's just annoying, it's painful to go back and correct it on your cell phone.

Ninety-nine percent is game changing. If there's 99 percent, it becomes reliable. It just works and you use it all the time. So this is not just a four percent incremental improvement, this is the difference between people rarely using it and people using it all the time.

So what is the hurdle to 99 percent at this point?

We need even bigger rocket engines and we still need even more rocket fuel. Both are still constrained and the two have to grow together. We're still working on pushing that boundary.

Sophia

LIFE LESSONS from FASCINATING PEOPLE

Sophia is a project to collect life lessons from fascinating people. Learn more: [hn.my/sophia]

Andrew Ng is an Associate Professor at Stanford; Chief Scientist of Baidu; and Chairman and Co-Founder of Coursera. In 2011 he led the development of Stanford University's main MOOC (Massive Open Online Courses) platform and also taught an online Machine Learning class that was offered to over 100,000 students, leading to the founding of Coursera.

Nico Pitney is an executive editor at the The Huffington Post. Previously he was national editor, managing editor, and VP of product development. He was also HuffPost's first DC bureau chief, overseeing coverage during the 2008 elections. He lives in San Francisco with his wife and daughter.

Swarm v. Fleet v. Kubernetes v. Mesos

Comparing Different Orchestration Tools

By ADRIAN MOUAT

Most software systems evolve over time. New features are added and old ones pruned. Fluctuating user demand means an efficient system must be able to quickly scale resources up and down. Demands for near-zero downtime require automatic fail-over to pre-provisioned back-up systems, normally in a separate data centre or region.

On top of this, organizations often have multiple such systems to run, or need to run occasional tasks such as data-mining that are separate from the main system but require significant resources, or talk to the existing system.

When using multiple resources, it is important to make sure they are efficiently used — not sitting idle — but can still cope with spikes in demand. Balancing cost-effectiveness against the ability to quickly scale is a difficult task that can be approached in a variety of ways.

All of this means that the running of a non-trivial system is full of administrative tasks and challenges, the complexity of which should not be underestimated. It quickly becomes impossible to look after machines on an individual level; rather than patching and updating machines one-by-one, they must be treated identically. When a machine develops a problem, it should be destroyed and replaced, rather than nursed back to health.

Various software tools and solutions exist to help with these challenges. Let's focus on orchestration tools, which help make all the pieces work together, working with the cluster to start containers on appropriate hosts and connect them together. Along the way, we'll consider scaling and automatic failover, which are important features.

Swarm

Swarm [hn.my/swarm] is the native clustering tool for Docker. Swarm uses the standard Docker API, meaning containers can be launched using normal Docker run commands, and Swarm will take care of selecting an appropriate host to run the container on. This also means that other tools which use the Docker API — such as Compose and bespoke scripts — can use Swarm without any changes and take advantage of running on a cluster rather than a single host.

The basic architecture of Swarm is fairly straightforward; each host runs a Swarm agent and one host runs a Swarm manager (on small test clusters this host may also run an agent). The manager is responsible for the orchestration and scheduling of containers on the hosts. Swarm can be run in a high-availability mode where one of etcd, Consul, or ZooKeeper is used to handle fail-over to a back-up manager. There are several different methods for how hosts are found and added to a cluster, which is known as discovery in Swarm. By default, token-based discovery is used, where the addresses of hosts are kept in a list stored on the Docker Hub.

Fleet

Fleet [hn.my/fleet] is the cluster management tool from CoreOS. It bills itself as a "low-level cluster engine", meaning that it is expected to form a "foundation layer" for higher-level solutions such as Kubernetes.

The most distinguishing feature of fleet is that it builds on top of systemd. Whereas systemd provides system and service initialization for a single machine, Fleet extends this to a cluster of machines. Fleet reads systemd unit files, which are then scheduled on a machine or machines in the cluster.

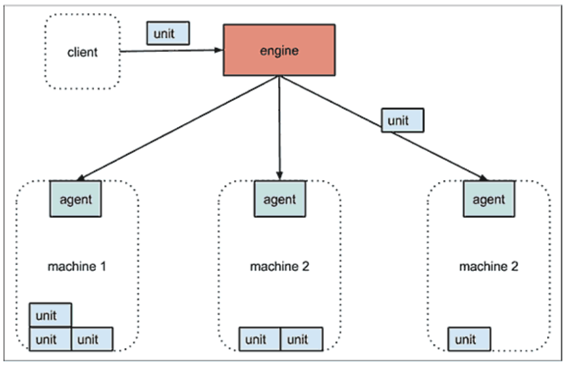


Figure 12-2. Fleet Architecture

The technical architecture of fleet is shown in Figure 12-2. Each machine runs an engine and an agent. Only one engine is active in the cluster at any time, but all agents are constantly running (for the sake of the diagram, the active engine is shown separately to the machines, but it will be running on one of them). Systemd unit files (henceforth units) are submitted to the engine, which will schedule the job on the "least-loaded" machine. The unit file will normally simply run a container. The agent takes care of starting the unit and reporting state. Etcd is used to enable communication between machines and store the status of the cluster and units.

The architecture is designed to be fault-tolerant; if a machine dies, any units scheduled on that machine will be restarted on new hosts.

Fleet supports various scheduling hints and constraints. At the most basic level, units can be scheduled as global, meaning an instance will run on all machines, or as a single unit which will run on a single machine. Global scheduling is very useful for utility containers for tasks such as logging and monitoring. Various affinity type constraints are supported, so for example, a container that runs a health check can be scheduled to always run next to the application server. Metadata can also be attached to hosts and used for scheduling, so you could, for example, ask for your containers to run on machines belonging to a given region or with certain hardware.

As Fleet is based on systemd, it also supports the concept of socket activation; a container can be spun up in response to a connection on a given port. The primary advantage of this is that processes can be created just-in-time, rather than sitting around idle waiting for something to happen. There are potentially other benefits related to management of sockets, such as not losing messages between container restarts.

Kubernetes

Kubernetes [kubernetes.io] is a container orchestration tool built by Google, based on their experiences using containers in production over the last decade. Kubernetes is somewhat opinionated and enforces several concepts around how containers are organized and networked. The primary concepts you need to understand are:

- **Pods** — Pods are groups of containers that are deployed and scheduled together. Pods form the atomic unit of scheduling in Kubernetes, as opposed to single containers in other systems. A pod will typically include one to five containers which work together to provide a service. In addition to these user containers, Kubernetes will run other containers to provide logging and monitoring services. Pods are treated as ephemeral in Kubernetes; you should expect them to be created and destroyed continually as the system evolves.
- **Flat Networking Space** — Networking is very different in Kubernetes to the default Docker networking. In the default Docker networking, containers live on a private subnet and can't communicate directly with containers on other hosts without forwarding ports on the host or using proxies. In Kubernetes, containers within a pod share an IP address, but the address space is "flat" across all pods, meaning all pods can talk to each other without any Network Address Translation (NAT). This makes multi-host clusters much easier to manage, at the cost of not supporting links and making single-host (or, more accurately, single-pod) networking a little more tricky. As containers in the same pod share an IP, they can communicate by using ports on the local host address (which does mean you need to coordinate port usage within a pod).

- **Labels** — Labels are key-value pairs attached to objects in Kubernetes, primarily pods, used to describe identifying characteristics of the object, e.g. version: dev and tier: front-end. Labels are not normally unique; they are expected to identify groups of containers. Label selectors can then be used to identify objects or groups of objects, for example, all the pods in the front-end tier with environment set to production. Through using labels, it is easy to do grouping tasks such as assigning pods to load-balanced groups or moving pods between groups.
- **Services** — Services are stable endpoints that can be addressed by name. Services can be connected to pods by using label selectors; for example, my "cache" service may connect to several "redis" pods identified by the label selector "type": "redis". The service will automatically round-robin requests between the pods. In this way, services can be used to connect parts of a system to each other. Using services provides a layer of abstraction that means applications do not need to know internal details of the service they are calling; for example, application code running inside a pod only needs to know the name and port of the database service to call. It does not care how many pods make up the database, or which pod it talked to last time. Kubernetes will set up a DNS server for the cluster that watches for new services and allows them to be addressed by name in application code and configuration files.

  It is also possible to set up services which do not point to pods but to other preexisting services such as external APIs or databases.
- **Replication Controllers** — Replication controllers are the normal way to instantiate pods in Kubernetes (typically, you don't use the Docker CLI in Kubernetes). They control and monitor the number of running pods (called replicas) for a service. For example, a replication controller may be responsible for keeping five Redis pods running. Should one fail, it will immediately launch a new one. If the number of replicas is reduced, it will stop any excess pods. Although using replication controllers to instantiate all pods adds an extra layer of configuration, it also significantly improves fault tolerance and reliability.

Mesos and Marathon

Apache Mesos [mesos.apache.org] is an open-source cluster manager. It's designed to scale to very large clusters involving hundreds or thousands of hosts. Mesos supports diverse workloads from multiple tenants; one user's Docker containers may be running next to another user's Hadoop tasks.

Apache Mesos was started as a project at the University of California, Berkeley before becoming the underlying infrastructure used to power Twitter and an important tool at many major companies such as eBay and Airbnb. A lot of continuing development in Mesos and supporting tools (such as Marathon) is undertaken by Mesosphere, a company co-founded by Ben Hindman, one of the original developers of Mesos.

The architecture of Mesos is designed around high-availability and resilience. The major components in a Mesos cluster are:

- **Mesos Agent Nodes** — Responsible for actually running tasks. All agents submit a list of their available resources to the master. There will typically be tens to thousands of agent nodes.
- **Mesos Master** — The master is responsible for sending tasks to the agents. It maintains a list of available resources and makes "offers" of them to frameworks. The master decides how many resources to offer based on an allocation strategy. There will typically be two or four standby masters ready to take over in case of a failure.
- **ZooKeeper** — Used in elections and for looking up address of current master. Typically three or five ZooKeeper instances will be running to ensure availability and handle failures.
- **Frameworks** — Frameworks coordinate with the master to schedule tasks onto agent nodes. Frameworks are composed of two parts: the executor process which runs on the agents and takes care of running the tasks, and the scheduler which registers with the master and selects which resources to use based on offers from the master. There may be multiple frameworks running on a Mesos cluster for different kinds of tasks. Users wishing to submit jobs interact with frameworks rather than directly with Mesos.

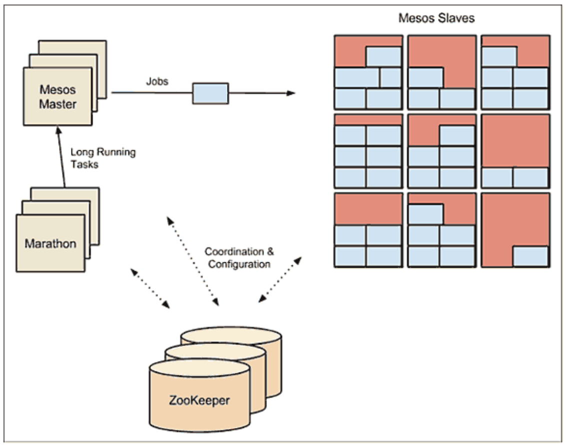


Figure 12-4. Mesos Cluster

In Figure 12-4 we see a Mesos cluster which uses the Marathon framework as the scheduler. The Marathon scheduler uses ZooKeeper to locate the current Mesos master which it will submit tasks to. Both the Marathon scheduler and the Mesos master have standbys ready to start work should the current master become unavailable.

Typically, ZooKeeper will run on the same hosts as the Mesos master and its standbys. In a small cluster, these hosts may also run agents, but larger clusters require communication with the master, making this less feasible. Marathon may run on the same hosts as well, or may instead run on separate hosts which live on the network boundary and form the access point for clients, thus keeping clients separated from the Mesos cluster itself.

Marathon [hn.my/marathon] (from Mesosphere) is designed to start, monitor, and scale long-running applications. Marathon is designed to be flexible about the applications it launches, and can even be used to start other complementary frameworks such Chronos ("cron" for the datacenter). It makes a good choice of framework for running Docker containers, which are directly supported in Marathon.

Like the other orchestration frameworks we've looked at, Marathon supports various affinity and constraint rules. Clients interact with Marathon through a REST API. Other features include support for health checks and an event stream that can be used to integrate with load-balancers or for analyzing metrics.

Conclusion

There are clearly a lot of choices for orchestrating, clustering, and managing containers. That being said, the choices are generally well-differentiated. In terms of orchestration, we can say the following:

- Swarm has the advantage (and disadvantage) of using the standard Docker interface. Whilst this makes it very simple to use Swarm and to integrate it into existing workflows, it may also make it more difficult to support the more complex scheduling that may be defined in custom interfaces.

- Fleet is a low-level and fairly simple orchestration layer that can be used as a base for running higher-level orchestration tools, such as Kubernetes or custom systems.

- Kubernetes is an opinionated orchestration tool that comes with service discovery and replication baked-in. It may require some redesigning of existing applications, but used correctly will result in a fault-tolerant and scalable system.

- Mesos is a low-level, battle-hardened scheduler that supports several frameworks for container orchestration, including Marathon, Kubernetes, and Swarm. At the time of writing, Kubernetes and Mesos are more developed and stable than Swarm. In terms of scale, only Mesos has been proven to support large-scale systems of hundreds or thousands of nodes. However, when looking at small clusters of, say, less than a dozen nodes, Mesos may be an overly complex solution. ■

Adrian Mouat is Chief Scientist at Container Solutions, a pan-European services company that specialises in Docker and Mesos. He is the author of the O'Reilly book Using Docker: Developing and Deploying Software with Containers. In the past, he has worked on a wide range of software projects, from small webapps to large-scale data-analysis software.

Deploying a Django App with No Downtime

By PETERIS CAUNE

WHEN HEALTHCHECKS.IO STARTED TO RECEIVE more than one request per second, it became clear I could not just go on carelessly restarting web servers after code deploys. For a monitoring service, it would be bad form to miss even a few HTTP requests. And, going forward, if the server gets busier, the problem only becomes bigger.

To give a quick overview of what I'm working with, the app is a relatively straightforward Django app, served by gunicorn behind nginx. Data lives in a PostgreSQL database. The gunicorn process and an additional background job are both managed by supervisor [supervisord.org]. It's hosted on a single $20 DigitalOcean droplet.

Aside: With regard to technology choices, the guiding principle I've been following is to keep the stack as simple as is feasible for as long as possible. Adding things, like load balancers, database replication, key value store, message queue, and so on, would each have certain benefits. Then on the other hand, there would also be more stuff to be managed, monitored, and kept backed up. Also, for someone new to the project, it would take more time to figure out the "ins and outs" of the system and set up everything from scratch. I see it as a nifty challenge to stay with the simple, no-frills setup, while also not compromising performance or features.

The deployment mechanism I've used thus far is a Fabric script plus configuration templates for supervisor and nginx. Each time I run "fab deploy" from my workstation, Fabric script does the following on the remote host:

- sets up a new directory for the new deployment. Let's refer to this directory as $TARGET.
- sets up a Python3 virtualenv in $TARGET/venv
- fetches the latest snapshot of code from GitHub into $TARGET. It is convenient to use GitHub's Subversion interface for this and run an "svn export" command. It produces just the source files without any version control metadata — exactly what's needed.
- installs dependencies listed in requirements file. These get installed into the new virtualenv and don't affect the live application. Downloading and building the dependencies take up to a minute.
- runs Django management commands to collect static files, run database migrations, etc.
- rewrites the supervisor configuration file to run gunicorn from the new virtual environment
- updates nginx configuration, in case I've changed anything in the nginx configuration template
- runs "supervisorctl reload" and "/etc/init.d/nginx restart". At this point the web application becomes unavailable and remains unavailable until supervisor starts back up, launches gunicorn process, and the Django code initializes. This usually takes five to ten seconds, and nginx would typically return "502 Bad Gateway" responses during this time.
- All done!

Here's how the relevant part of Fabric script looks. The virtualenv context manager seen below is from the excellent fabtools library. [hn.my/fabtools]

```
def deploy():
    """ Checks out code, prepares venv, runs
    management commands, updates supervisor and
    nginx configuration. """

    now = datetime.datetime.today()
    now_string = now.strftime("%Y%m%d-%H%M%S")
    project_dir = "/home/hc/webapps/hc-%s" %
now_string
    venv_dir = os.path.join(project_dir, "venv")

    svn_url = "https://github.com/healthchecks/
healthchecks/trunk"
    run("svn export %s %s" % (svn_url, proj-
ect_dir))

    with cd(project_dir):
        run("virtualenv --python=python3 --sys-
tem-site-packages venv")
        # local_settings.py is where things like
        # access keys go
        put("local_settings.py", ".")
        put("newrelic.ini", ".")

        with virtualenv(venv_dir):
            run("pip install -U gunicorn raven
newrelic")
            run("pip install -r requirements.
txt")
            run("python manage.py collectstatic
--noinput")
            run("python manage.py compress")

            with settings(user="hc"):
                run("python manage.py migrate")
                run("python manage.py ensure-
triggers")
                run("python manage.py clearses-
sions")

    switch(project_dir)
```

```
def switch(project_dir):
    # Supervisor
    upload_template("supervisor/hc.conf.tmpl",
                    "/etc/supervisor/conf.d/
hc.conf",
                    context=locals(),
                    backup=False,
                    use_sudo=True)

    upload_template("supervisor/hc_sendalerts.
conf.tmpl",
                    "/etc/supervisor/conf.d/
hc_sendalerts.conf",
                    context=locals(),
                    backup=False,
                    use_sudo=True)

    # Nginx
    upload_template("nginx/hc.conf.tmpl",
                    "/etc/nginx/sites-enabled/
hc.conf",
                    context=locals(),
                    backup=False,
                    use_sudo=True)

    sudo("supervisorctl reload")
    sudo("/etc/init.d/nginx reload")
```

Now, how to eliminate the downtime during the last steps of each deploy? Let's set some constraints: no load balancer (for now anyway). Everything runs off a single box, and even a single non-200 response is undesirable. And, baby steps: I will consider the simple (and common) case when there are no database migrations to be applied or they are backwards-compatible; the old version of the app keeps working acceptably after the migrations are applied.

The first idea I looked into was based on the observation that availability is more important for some parts of the app than others. Specifically, the API part of the app listens for pings from the monitored client systems, and the front-end part serves pages to normal website visitors. While it would be embarrassing to show error pages to human visitors, not missing any pings is actually more important. A missed ping can lead to a false alert being sent sometime later. That's even more embarrassing!

I considered and prototyped listening to pings using Amazon API Gateway. It would put ping messages in Amazon SQS queue, which the Django app could consume at its leisure. This would be a relatively simple way to improve availability and scalability by quite a lot at the cost of somewhat increased complexity and a new external dependency. I might look into this again in the future.

Next idea: Separate the "listen to pings" functionality from the rest of the Django app. The ping listener logic is very simple and, ultimately, amounts to two SQL operations: one update and one insert. It could be easy enough to rewrite this part, perhaps using one of the Python microframeworks, or maybe using a language other than Python, or maybe even handle it from nginx itself, using ngx_postgres module [hn.my/ngx]. For a little amusement, here's the nginx configuration fragment which, basically, works as is:

```
location ~ ^/(\w\w\w\w\w\w\w\w-\w\w\w\w-\w\w\w\
w-\w\w\w\w-\w\w\w\w\w\w\w\w\w\w\w\w)/?$ {
    add_header Content-Type text/plain;

    postgres_pass   database;
    postgres_output value;

    postgres_escape $ip $remote_addr;
    postgres_escape $agent =$http_user_agent;
    postgres_escape $body =$request_body;

    postgres_query "
        WITH t AS (
            UPDATE api_check
            SET last_ping=now()
            WHERE code='$1'
            RETURNING id, last_ping
        )
        INSERT INTO api_ping
            (created, remote_addr, method, ua,
body, owner_id, scheme)
        SELECT
            last_ping, $ip, '$request_method',
$agent, $body, id, '$scheme'
        FROM t
        RETURNING 'OK'
    ";

    postgres_rewrite no_changes 400;
}
```

Here's what's going on: When the client requests the URL of a certain format, the server runs a PostgreSQL query and returns either HTTP code 200 or HTTP code 400. This is also a performance win because the request doesn't have to travel through the hoops of gunicorn, Django, and psycopg2. As long as the database is available, nginx can handle the ping requests, even if the Django application is not running for any reason.

The not-so-great thing with this approach is that it's "tricky" and adds to the number of things that the developer and systems administrator need to know. For example, when the database schema changes, the SQL query above might need to be updated and tested as well. Getting the ngx_postgres extension set up isn't a simple matter of "apt-get install" either.

Thinking more about it, the main goal of zero downtime can also be achieved by just carefully orchestrating process restarts and reloads.

My deployment script was using "/etc/init.d/nginx restart" because I didn't know any better. As I learned, it can be replaced it with "/etc/init.d/nginx reload" which handles things gracefully:

> *Run service nginx reload or /etc/init.d/nginx reload.*
>
> *It will do a hot reload of the configuration without downtime. If you have pending requests, then there will be lingering nginx processes that will handle those connections before it dies, so it's an extremely graceful way to reload configs. – "Nginx config reload without downtime" on ServerFault [hn.my/configreload]*

Similarly, my deployment script was using "supervisorctl reload" which stops all managed services, rereads configuration, and starts all services. Instead "supervisorctl update" can be used to start, stop, and restart the changed tasks as necessary.

Now, here's what "fab deploy" can do:

- set up a new virtual environment as before
- create a supervisor task with unique name ("hc_timestamp")
- start the new gunicorn process alongside the running one. nginx talks to gunicorn processes using UNIX sockets, and each process uses a separate, again timestamped, socket file
- wait a little — then verify that the new gunicorn process has started up and is serving responses

- update nginx configuration to point to the new socket file and reload nginx
- stop the old gunicorn process

Here's the improved part of Fabric script which juggles supervisor jobs:

```
def switch(tag, project_dir):
    # Supervisor
    supervisor_conf_path = "/etc/supervisor/
conf.d/hc_%s.conf" % tag
    upload_template("supervisor/hc.conf.tmpl",
supervisor_conf_path, context=locals(),
backup=False, use_sudo=True)

    upload_template("supervisor/hc_sendalerts.
conf.tmpl", "/etc/supervisor/conf.d/hc_sen-
dalerts.conf", context=locals(), backup=False,
use_sudo=True)

    # Starts up gunicorn from the new virtualenv
    sudo("supervisorctl update")

    # Give it some time to start up
    time.sleep(5)

    # Let's check the new server is nominally
    # working gunicorn listens on UNIX socket so
    # this is a bit contrived:
    l = ("GET /about/ HTTP/1.0\\r\\n"
         "Host: healthchecks.io\\r\\n"
         "\\r\\n")

    cmd = 'echo -e "%s" | nc -U /tmp/hc-%s.sock'
% (l, tag)
    # Look for known string in response. If it's
    # not found, something is wrong with the new
    # deployment and we abort
    assert "Monkey See Monkey Do" in run(cmd,
quiet=True)

    # nginx
    upload_template("nginx/hc.conf.tmpl",
"/etc/nginx/sites-enabled/hc.conf",
context=locals(), backup=False, use_sudo=True)

    sudo("/etc/init.d/nginx reload")

# should be live now - remove supervisor conf
# for previous versions
    s = sudo("for i in /etc/supervisor/conf.d/*.
conf; do echo $i; done")
    for line in s.split("\n"):
        line = line.strip()
        if line == supervisor_conf_path:
            continue
        if line.startswith("/etc/supervisor/
conf.d/hc_2"):
            sudo("rm %s" % line)

    # This stops gunicorn processes
    sudo("supervisorctl update")
```

With this, nginx is always serving requests and is talking to a live gunicorn process at all times. To verify this in practice, I wrote a quick script that requests a particular URL again and again in an infinite loop. As soon as it hits a non-200 response, it would print out a hard-to-miss error message. With this banging against my test VM, I did a couple deploys and saw no missed requests. Success!

Summary

There are many ways to achieve zero downtime during code deploys, and each has its own trade-offs. For example, a reasonable strategy is to extract the critical parts out of the bigger application. Each part can then be updated independently. Later, the parts can also be scaled independently. The downside to this is more code and configuration to maintain.

What I ultimately ended up doing:

- hot-reload supervisor and nginx configurations instead of just restarting them. Obvious thing to do in retrospect.
- make sure the new gunicorn process is alive and being used by nginx before stopping the old gunicorn process.
- and keep the whole setup relatively simple. As the project gets more usage, I will need to look at performance hotspots and figure out how to scale horizontally, but this should do for now! ■

Pēteris Caune is the creator of *healthchecks.io*, a tool for monitoring cron jobs. Pēteris doesn't get very much done because on sunny days he is out cycling, and on rainy days he is on his indoor bike trainer.

Erlang Beauty

By NATE BARTLEY

There is an adage in the Erlang community that we should make our code work, then make it beautiful, then, and if truly necessary, make it all run fast. This insight is rendered from decades of coding of many brilliant hackers. In fact, the last part is probably a nod to Donald Knuth's famous caution against early optimization.

It may look familiar to some, or maybe it's new. Certainly, there are variations of this in other languages for sure. But all the same, one may coil up and ask, why is beautiful on the list?

Making code beautiful is essential, and there's a reason it is parked in the number-two spot in the pecking order. If our code isn't beautiful, it's unlikely that it will convey just what the hell is going on.

Since I'm no level-ten Erlang poet, I consult a refactoring style guide that ensures my code will be straightforward to read. A lot of this guideline is lifted from Garrett Smith's [hn.my/gar1t] fantastic insight on the same matter — with some light repurposing here and there. If you need some tips on getting your Erlang code past the mere working stage and into the beauty pageant, then please, read on.

I call this module-writing heuristic "fucrs" (pronounced any way you like).

Give me an "F": function-ize

After a version of an Erlang module is working, I go back and turn all my "case" statements into plain ol' functions. For example:

```
foo(X) ->
    case X of
        bar   -> void;
        _Else -> undefined
    end.
```

would become:

```
foo(X) -> foo1(X).

foo1(bar)    -> void;
foo1(_Else) -> undefined.
```

This practice is a gold-standard for me, as I immediately noticed my code to be vastly more lucid. "F" also serves as a reminder to turn any haphazard "funs" into normal functions, too. Sometimes you need "funs" — of course — but ordinarily, writing a plain function is a much cleaner way to present things. Once done, I move on to the next letter.

Gimme a "U": un-nest

This is an easy one to do, but sadly, it is not popular. It's not alarming to see code in the wild with functions entrenched six or seven times. And incredibly, the Elixir language has introduced a language device, the pipe operator, to better mollify this malpractice. Obviously, nesting runs rampant. The "fucrs" guideline takes an extreme conservative stance on this; nesting should never happen, ever. The values passed into functions must only be variables, or basic data constructs like lists, tuples, etc.; never a function. For example:

```
foo(X) ->
final_function(maybe_function(X)).
```

would be rewritten:

```
foo(X) ->
    Maybe = maybe_function(X),
    final_function(Maybe).
```

Spelling things out so pedantically makes code dead-simple and clear. Yes, there is a tad more code, but you will also note that nothing is hiding. Un-nesting simply dumbs things down. Now, who wouldn't want that after hours of squinting at a screen?

Gimme a "C": canonicalize

It's nice to open a module and see some familiar faces. Although there is no standard naming convention for functions, "fucrs" makes an attempt to include the ones from Garrett's talk whenever possible. Functions named as "new", "start", "init", etc. are pleasantly unsurprising and can help your modules have an air of familiarity.

Gimme an "R": rename

Quite related to canonicalization is to do a pass on the code in order to rename variables/atoms as necessary. I find that in the heat of the moment, a naming will occur out of some emotional response to something. For example, I once caught an atom I had named "stupid_foo" which at the time meant quite a bit, but means little thereafter. Naming things can be hard, but I find when I am tasked with renaming things in bulk, it's easier than coming up with great names while also tasked with getting code to work. Expressive names are helpful to your future self and others. Use simple, deliberate words as much as possible.

Finally; "S": seven-ize

Someone smart, somewhere, at some time, did a study which concluded that the human brain can easily hold six or seven items in short-term memory with little trouble, but beyond that, it becomes taxing. This applies nicely to lines of code in a function. Amazingly, after all the above guidelines have been followed, bringing a function's lines-of-code count down to a maximum of seven is surprisingly easy to do. It may seem crazy, but I challenge you to try it, first as an exercise, then come to your own conclusions. For me, I tend to max-out around five lines of code per function with no effort, though at first, seven was an easier goal. The reason should be clear; a reader can keep a handful of statements in mind when reading such a terse function, but reasoning a behemoth, it goes against what our brains are capable of.

So, there you have it: a one-word style guide to consult after your Erlang code is ticking along without error, but could benefit from a face-lift. Give it a try, and be sure to thank Garrett if "fucrs" works for you. ■

Nate Bartley is co-founder of ikura, an Oakland-based Erlang startup bringing CRON into the 21st century. When he's not hacking, he's playing violin & climbing rocks around California.

How to Write a Git Commit Message

By CHRIS BEAMS

Introduction: Why good commit messages matter

If you browse the log of any random git repository, you will probably find its commit messages are more or less a mess. For example, take a look at these gems from my early days committing to Spring:

```
$ git log --oneline -5 --author
cbeams --before "Fri Mar 26
2009"

e5f4b49 Re-adding Configuration-
PostProcessorTests after its
brief removal in r814. @Ignore-
ing the testCglibClassesAre-
LoadedJustInTimeForEnhance-
ment() method as it turns out
this was one of the culprits
in the recent build breakage.
[...] The test method is still
useful, but should only be run
on a manual basis to ensure
CGLIB is not prematurely class-
loaded, and should not be run
as part of the automated build.
2db0f12 fixed two build-
breaking issues: + reverted
ClassMetadataReadingVisitor
to revision 794 + eliminated
ConfigurationPostProcessorT-
ests until further investiga-
tion determines why it causes
downstream tests to fail (such
as the seemingly unrelated
ClassPathXmlApplicationContext-
Tests)
147709f Tweaks to package-info.
java files
22b25e0 Consolidated Util and
MutableAnnotationUtils classes
into existing AsmUtils
7f96f57 polishing
```

Yikes. Compare that with these more recent commits from the same repository:

```
$ git log --oneline -5 --author
pwebb --before "Sat Aug 30
2014"

5ba3db6 Fix failing Compos-
itePropertySourceTests
84564a0 Rework @PropertySource
early parsing logic
e142fd1 Add tests for ImportSe-
lector meta-data
887815f Update docbook depen-
dency and generate epub
ac8326d Polish mockito usage
```

Which would you rather read?

The former varies wildly in length and form; the latter is concise and consistent. The former is what happens by default; the latter never happens by accident.

While many repositories' logs look like the former, there are exceptions. The Linux kernel [hn.my/linuxcommit] and git [hn.my/gitcommit] itself are great examples. Look at Spring Boot, [hn.my/spring] or any repository managed by Tim Pope.

The contributors to these repositories know that a well-crafted git commit message is the best way to communicate context about a change to fellow developers (and indeed to their future selves). A diff will tell you what changed, but only the commit message can properly tell you why. Peter Hutterer makes this point well:

> *Re-establishing the context of a piece of code is wasteful. We can't avoid it completely, so our efforts should go to reducing it [as much] as possible. Commit messages can do exactly that, and as a result, a commit message shows whether a developer is a good collaborator.*

If you haven't given much thought to what makes a great git commit message, it may be that you haven't spent much time using `git log` and related tools. There is a vicious cycle here; because the commit history is unstructured and inconsistent, one doesn't spend much time using or taking care of it. And because it doesn't get used or taken care of, it remains unstructured and inconsistent.

But a well-cared-for log is a beautiful and useful thing. `git blame`, `revert`, `rebase`, `log`, `shortlog`, and other subcommands come to life. Reviewing others' commits and pull requests becomes something worth doing and suddenly can be done independently. Understanding why something happened months or years ago becomes not only possible but efficient.

A project's long-term success rests (among other things) on its maintainability, and a maintainer has few tools more powerful than his project's log. It's worth taking the time to learn how to care for one properly. What may be a hassle at first soon becomes habit and, eventually, a source of pride and productivity for all involved.

Most programming languages have well-established conventions as to what constitutes idiomatic style, i.e. naming and formatting and so on. There are variations on these conventions, of course, but most developers agree that picking one and sticking to it is far better than the chaos that ensues when everybody does their own thing.

A team's approach to its commit log should be no different. In order to create a useful revision history, teams should first agree on a commit message convention that defines at least the following three things:

Style. Markup syntax, wrap margins, grammar, capitalization, punctuation. Spell these things out, remove the guesswork, and make it all as simple as possible. The end result will be a remarkably consistent log that's not only a pleasure to read but that actually does get read on a regular basis.

Content. What kind of information should the body of the commit message (if any) contain? What should it not contain?

Metadata. How should issue-tracking IDs, pull request numbers, etc. be referenced?

Fortunately, there are well-established conventions as to what makes an idiomatic git commit message. Indeed, many of them are assumed in the way certain git commands function. There's nothing you need to reinvent. Just follow the seven rules below and you're on your way to committing like a pro.

The seven rules of a great git commit message

1. Separate subject from body with a blank line
2. Limit the subject line to fifty characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at seventy-two characters
7. Use the body to explain what and why vs. how

For example:

```
Summarize changes in around
fifty characters or less

More detailed explanatory
text, if necessary. Wrap it to
about seventy-two characters
or so. In some contexts, the
first line is treated as the
subject of the commit and the
rest of the text as the body.
The blank line separating the
summary from the body is criti-
cal (unless you omit the body
entirely); various tools like
`log`, `shortlog`, and `rebase`
can get confused if you run the
two together.

Explain the problem that this
commit is solving. Focus on
why you are making this change
as opposed to how (the code
explains that). Are there side
effects or other unintuitive
consequences of this change?
Here's the place to explain
them.

Further paragraphs come after
blank lines.

- Bullet points are okay, too.

- Typically a hyphen or aster-
isk is used for the bullet,
preceded by a single space,
with blank lines in between,
but conventions vary here.

If you use an issue tracker,
put references to them at the
bottom, like this:

Resolves: #123
See also: #456, #789
```

1 Separate subject from body with a blank line

From the `git commit` manpage:

> *Though not required, it's a good idea to begin the commit message with a single short (less than fifty-character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout git. For example, git-format-patch (1) turns a commit into email, and it uses the title on the subject line and the rest of the commit in the body.*

Firstly, not every commit requires both a subject and a body. Sometimes a single line is fine, especially when the change is so simple that no further context is necessary. For example:

```
Fix typo in introduction to
user guide
```

Nothing more need be said; if the reader wonders what the typo was, she can simply take a look at the change itself, i.e. use `git show` or `git diff` or `git log -p`.

If you're committing something like this at the command line, it's easy to use the `-m` switch to `git commit`:

```
$ git commit -m "Fix typo in
introduction to user guide"
```

However, when a commit merits a bit of explanation and context, you need to write a body. For example:

```
Derezz the master control
program

MCP turned out to be evil and
had become intent on world
domination. This commit throws
Tron's disc into MCP (causing
its deresolution) and turns it
back into a chess game.
```

This is not so easy to commit this with the `-m` switch. You really need a proper editor. If you do not already have an editor set up for use with git at the command line, read this section of Pro Git. [hn.my/progitconf]

In any case, the separation of subject from body pays off when browsing the log. Here's the full log entry:

```
$ git log
commit 42e769bdf4894310333942f
Author: Kevin Flynn
        <kevin@flynnsarcade.com>
Date:   Fri Jan 01 00:00:00
        1982 -0200

 Derezz the master control program

 MCP turned out to be evil and
had become intent on world
domination. This commit throws
Tron's disc into MCP (causing
its deresolution) and turns it
back into a chess game.
```

And now `git log --oneline`, which prints out just the subject line:

```
$ git log --oneline 42e769
Derezz the master control
program
```

Or, `git shortlog`, which groups commits by user, again showing just the subject line for concision:

```
$ git shortlog
Kevin Flynn (1):
Derezz the master control program

Alan Bradley (1):
Introduce security program "Tron"

Ed Dillinger (3):
Rename chess program to "MCP"
Modify chess program
Upgrade chess program

Walter Gibbs (1):
Introduce protoype chess program
```

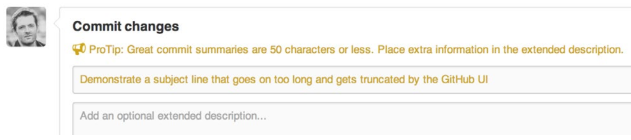
There are a number of other contexts in git where the distinction between subject line and body kicks in — but none of them work properly without the blank line in between.

2 Limit the subject line to fifty characters

Fifty characters is not a hard limit, just a rule of thumb. Keeping subject lines at this length ensures that they are readable and forces the author to think for a moment about the most concise way to explain what's going on.

> *Tip: If you're having a hard time summarizing, you might be committing too many changes at once. Strive for atomic commits (a topic for a separate post).*

GitHub's UI is fully aware of these conventions. It will warn you if you go past the fifty-character limit:



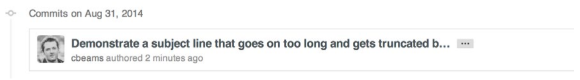
Commit changes

ProTip: Great commit summaries are 50 characters or less. Place extra information in the extended description.

Demonstrate a subject line that goes on too long and gets truncated by the GitHub UI

Add an optional extended description...

And will truncate any subject line longer than sixty-nine characters with an ellipsis:



Commits on Aug 31, 2014

Demonstrate a subject line that goes on too long and gets truncated b... ···
cbeams authored 2 minutes ago

So shoot for fifty characters, but consider sixty-nine the hard limit.

3 Capitalize the subject line

This is as simple as it sounds. Begin all subject lines with a capital letter.

For example:

- Accelerate to eighty-eight miles per hour

Instead of:

- accelerate to eighty-eight miles per hour

4 Do not end the subject line with a period

Trailing punctuation is unnecessary in subject lines. Besides, space is precious when you're trying to keep them to 50 chars or less.

Example:

- Open the pod bay doors

Instead of:

- Open the pod bay doors.

5 Use the imperative mood in the subject line

Imperative mood just means "spoken or written as if giving a command or instruction". A few examples:

- Clean your room
- Close the door
- Take out the trash

Each of the seven rules you're reading about right now are written in the imperative ("Wrap the body at seventy-two characters", etc).

The imperative can sound a little rude; that's why we don't often use it. But it's perfect for git commit subject lines. One reason for this is that **git itself uses the imperative whenever it creates a commit on your behalf.**

For example, the default message created when using `git merge` reads:

```
Merge branch 'myfeature'
```

And when using `git revert`:

```
Revert "Add the thing with the
stuff"
This reverts commit
cc87791524aedd593cff5a74532be-
fe7ab69ce9d.
```

Or when clicking the "Merge" button on a GitHub pull request:

```
Merge pull request #123 from
someuser/somebranch
```

So when you write your commit messages in the imperative, you're following git's own built-in conventions. For example:

- Refactor subsystem X for readability
- Update getting started documentation
- Remove deprecated methods
- Release version 1.0.0

Writing this way can be a little awkward at first. We're more used to speaking in the indicative mood, which is all about reporting facts. That's why commit messages often end up reading like this:

- Fixed bug with Y
- Changing behavior of X

And sometimes commit messages get written as a description of their contents:

- More fixes for broken stuff
- Sweet new API methods

To remove any confusion, here's a simple rule to get it right every time.

A properly formed git commit subject line should always be able to complete the following sentence:

- If applied, this commit will *your subject line here*

For example:

- If applied, this commit will *refactor subsystem X for readability*
- If applied, this commit will *update getting started documentation*
- If applied, this commit will *remove deprecated methods*
- If applied, this commit will *release version 1.0.0*
- If applied, this commit will *merge pull request #123 from user/branch*

Notice how this doesn't work for the other non-imperative forms:

- If applied, this commit will *fixed bug with Y*
- If applied, this commit will *changing behavior of X*
- If applied, this commit will *more fixes for broken stuff*
- If applied, this commit will *sweet new API methods*

Remember: Use of the imperative is important only in the subject line. You can relax this restriction when you're writing the body.

6 Wrap the body at seventy-two characters

Git never wraps text automatically. When you write the body of a commit message, you must mind its right margin and wrap text manually.

The recommendation is to do this at seventy-two characters, so that git has plenty of room to indent text while still keeping everything under eighty characters overall.

A good text editor can help here. It's easy to configure Vim, for example, to wrap text at seventy-two characters when you're writing a git commit. Traditionally, however, IDEs have been terrible at providing smart support for text wrapping in commit messages (although in recent versions, IntelliJ IDEA has finally gotten better about this).

7 Use the body to explain what and why vs. how

This commit from Bitcoin Core is a great example of explaining what changed and why:

```
commit eb0b56b19017ab5c16c745e-
6da39c53126924ed6
Author: Pieter Wuille <pieter.
wuille@gmail.com>
Date:   Fri Aug 1 22:57:55 2014
+0200

Simplify serialize.h's excep-
tion handling

Remove the 'state' and 'except-
mask' from serialize.h's stream
implementations, as well as
related methods.

As exceptmask always included
'failbit', and setstate was
always called with bits =
failbit, call it did was
immediately raise an excep-
tion. Get rid of those vari-
ables, and replace the setstate
with direct exception throwing
(which also removes some dead
code).

As a result, good() is never
reached after a failure (there
are only 2 calls, one of which
is in tests), and can just be
replaced by !eof().

fail(), clear(n) and excep-
tions() are just never called.
Delete them.
```

Take a look at the full diff [hn.my/eb0b5] and just think how much time the author is saving fellow and future committers by taking the time to provide this context here and now. If he didn't, it would probably be lost forever.

In most cases, you can leave out details about how a change has been made. Code is generally self-explanatory in this regard (and if the code is so complex that it needs to be explained in prose, that's what source comments are for). Just focus on making clear the reasons you made the change in the first place — the way things worked before the change (and what was wrong with that), the way they work now, and why you decided to solve it the way you did.

The future maintainer that thanks you may be yourself!

Tips

Learn to love the command line. Leave the IDE behind.

For as many reasons as there are git subcommands, it's wise to embrace the command line. Git is insanely powerful; IDEs are too, but each in different ways. I use an IDE every day (IntelliJ IDEA) and have used others extensively (Eclipse), but I have never seen IDE integration for git that could begin to match the ease and power of the command line (once you know it).

Certain git-related IDE functions are invaluable, like calling `git rm` when you delete a file, and doing the right stuff with `git` when you rename one. Where everything falls apart is when you start trying to commit, merge, rebase, or do sophisticated history analysis through the IDE.

When it comes to wielding the full power of git, it's command-line all the way.

Remember that whether you use Bash or Z shell, there are tab completion scripts that take much of the pain out of remembering the subcommands and switches.

Read Pro Git

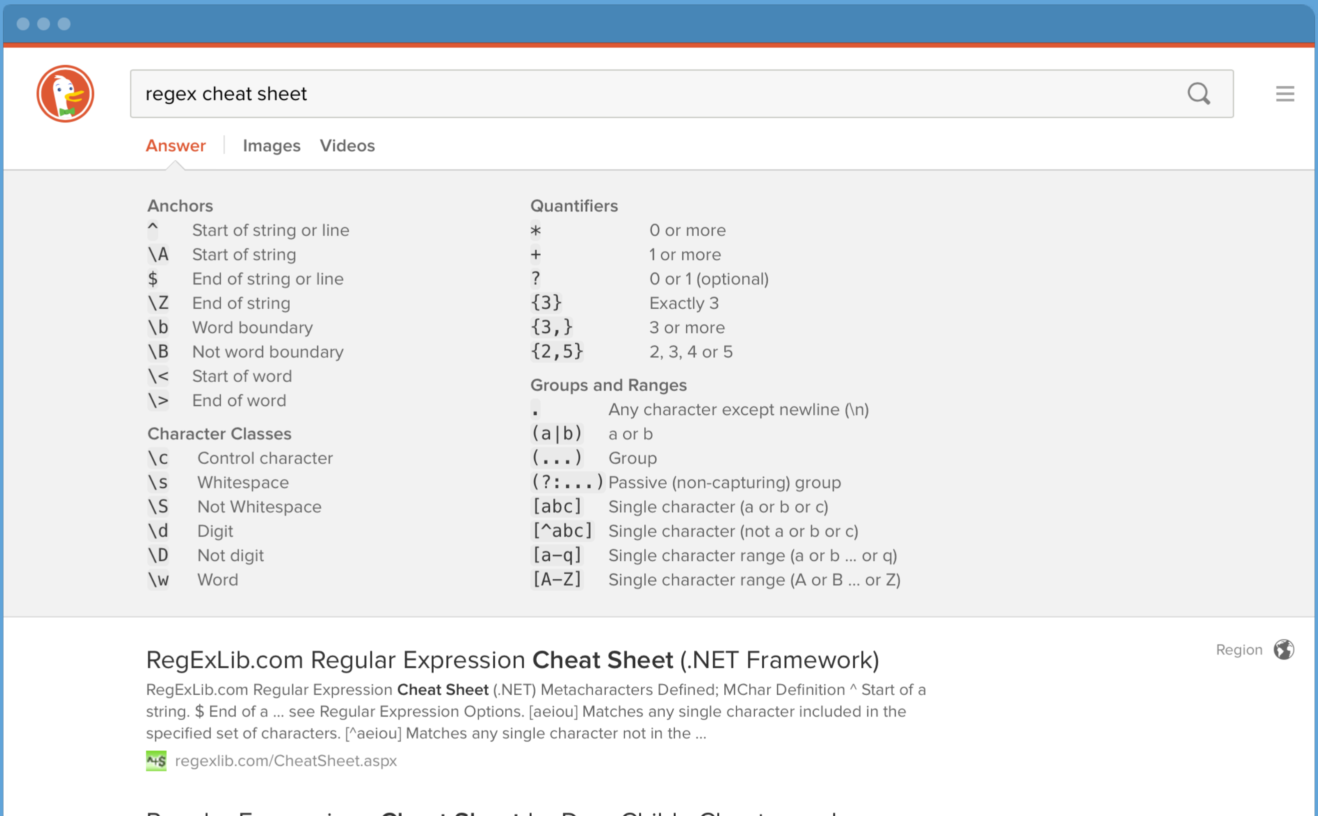
The Pro Git [hn.my/progit] book is available online for free, and it's fantastic. Take advantage! ■

Chris Beams is an open source software developer currently working on the Gradle build system. Follow him on Twitter at *@cbeams*

Join the DuckDuckGo Open Source Community.

Create Instant Answers or share ideas and help change the future of search.

Featured IA: Regex Contributor: mintsoft
Get started at duckduckhack.com



regex cheat sheet

Answer | Images Videos

Anchors

`^`	Start of string or line
`\A`	Start of string
`$`	End of string or line
`\Z`	End of string
`\b`	Word boundary
`\B`	Not word boundary
`\<`	Start of word
`\>`	End of word

Character Classes

`\c`	Control character
`\s`	Whitespace
`\S`	Not Whitespace
`\d`	Digit
`\D`	Not digit
`\w`	Word

Quantifiers

`*`	0 or more
`+`	1 or more
`?`	0 or 1 (optional)
`{3}`	Exactly 3
`{3,}`	3 or more
`{2,5}`	2, 3, 4 or 5

Groups and Ranges

`.`	Any character except newline (\n)
`(a|b)`	a or b
`(...)`	Group
`(?:...)`	Passive (non-capturing) group
`[abc]`	Single character (a or b or c)
`[^abc]`	Single character (not a or b or c)
`[a-q]`	Single character range (a or b ... or q)
`[A-Z]`	Single character range (A or B ... or Z)

Region

RegExLib.com Regular Expression **Cheat Sheet** (.NET Framework)
RegExLib.com Regular Expression **Cheat Sheet** (.NET) Metacharacters Defined; MChar Definition ^ Start of a string. $ End of a ... see Regular Expression Options. [aeiou] Matches any single character included in the specified set of characters. [^aeiou] Matches any single character not in the ...
regexlib.com/CheatSheet.aspx

Losing Sight

By LÉONIE WATSON

I DON'T KNOW WHO will read this. I don't even know why it has suddenly become important to write it, but for whatever it's worth, this is an account of an event in my life that changed everything.

I lost my sight over the course of twelve months, from late 1999 to late 2000. It was mostly my fault that it happened. I was diagnosed with Type I diabetes when I was a little girl. At the time they explained that I would have to eat a precise amount of food each day, and that I would need to inject a precise amount of insulin to handle it. These measurements were reviewed and revised on an annual basis.

Note: Type I diabetes means your body stops producing insulin and you have to inject it instead. Type II diabetes (the kind you hear about on the news) means that your body still produces insulin but is unable to absorb it properly, which is why it can often be controlled through diet and tablet-based medication instead of insulin injections.

When I was a little older, I asked my pediatrician why it had to be this way. I wanted to know why I couldn't work out how much food (carbohydrates) I was about to eat, measure my blood glucose, and then calculate my insulin dose based on those and other factors. After all, I reasoned, this is what my body would have done naturally, so why couldn't I emulate that behaviour? To this day I don't know whether he actually patted me on the head, or whether my subconscious has superimposed that memory based on his reply ("don't be so ridiculous"), but it doesn't really matter in the scheme of things.

Looking back, I understand that was the moment that everything changed. It would take another fifteen years for the impact of that moment to be felt, but that was where it all began. That was where the rebellion started.

At some point during my teens, I discovered that I could skip an injection without anything terrible happening. It wasn't something I did intentionally, at least not then, but it made me think I could get away with it from time to time. From then on, I did just enough injections to stop anyone from figuring out what I was up to. I stopped monitoring my blood glucose levels almost entirely, and as soon as I was old enough, I stopped going to see my doctor for annual checkups.

Throughout my student days, I had a riot. I went to drama school where I worked hard and played harder. I smoked, danced, partied, fell in and out of love, discussed Stanislavsky and Brecht until the wee small hours, drunkenly declaimed Shakespeare, and kept on ignoring the fact I was diabetic.

By the time the century was drawing to a close, I was working in the tech industry. Quite by accident I had gone to work for one of the UK's first ISPs in early 1997. Somewhere along the way I taught myself to code and by 1999 was working as a web developer. This was the era of the dot-com boom, and everyone was having fun. There were pool tables in the office, Nerf guns on every desk, insane parties that the company wrote off against taxes; Paul Okenfold was the soundtrack to our lives, and we'd fall out of clubs at six a.m. and drive to Glastonbury Tor to watch the sunrise just for the hell of it.

One morning in October 1999, I woke up with a hangover. As I looked at my reflection in the mirror, I realised I could see a ribbon of blood in my line of sight. As I looked left then right, the ribbon moved sluggishly as though floating in dense liquid. Assuming it was a temporary result of the previous night's antics, I left it a couple of days before visiting an optician to get it checked out. When I did, the optician took one look at the backs of my eyes and referred me to the nearest eye hospital for further investigation.

When diabetes is uncontrolled for a time, it causes a lot of unseen damage. It works something like this: When you eat something, your blood sugar levels rise and your body produces insulin to convert that glucose into usable energy. If enough insulin isn't available, then the cells in your body are starved of energy and begin to die, and the excess glucose remains trapped in your bloodstream. If that wasn't enough, the excess glucose smothers your red blood cells and prevents them from transporting oxygen efficiently around your body.

One of the ways this damage eventually manifests itself is diabetic retinopathy. In an attempt to get enough oxygen to the retina at the back of your eye, your body creates new blood vessels to try and compensate. The new blood vessels are created as an emergency measure and so they're weaker than they need to be. This means they're prone to bursting and hemorrhaging blood into the eye — creating visible ribbons of blood like the one I could see. The lack of oxygen to your retina and the accumulation of blood in your eye have the inevitable effect of damaging your sight.

The people at the eye hospital told me I would need laser treatment. This might halt the breakdown of the blood vessels at the back of my eye and enable the remaining vessels to strengthen enough to get the oxygen where it needed to go. It seemed like a reasonable option since my sight was still reasonably good at this point.

Laser treatment isn't pleasant. It requires an anesthetic injection into the eye before a laser is fired repeatedly at the blood vessels at the back. One side effect of this is that it plays havoc with your retina. If you think of your retina like a piece of cling film pulled taut over the open end of a jar, then imagine how it distorts when pressure is applied to it, you won't be far off the effect laser treatment can have. I remember one round of treatment skewing the sight in one eye around ninety degrees and making everything slightly pink for several hours. Trust me when I tell you that it's impossible to remain upright with one eye seeing straight and the other ninety degrees out of whack!

Looking back now, I realise my consultant knew that laser treatment was almost certainly a futile gesture. Despite this, no one ever came straight out and told me the consequences of having advanced diabetic retinopathy. It was always spoken of in terms of progressive deterioration without ever mentioning the logical conclusion of that progression.

I do remember the day I admitted it to myself though. My sight had been steadily worsening over the months, and it was a day in the spring of 2000 that it happened. I have no idea what prompted it, but I was walking down the stairs at home when it hit me like the proverbial sledgehammer — I would be blind. Until that moment I had never believed people when they said an emotional reaction could be like a physical blow. I don't doubt that anymore. With absolute certainty, I knew I would lose my sight and that I only had myself to blame. I sat on a step a couple up from the bottom of the stairs and fell apart. I cried like a child. I cried for my lost sight, for all my broken dreams, for my stupidity, for all the books I would never read, for the faces I would forget, and for all the things I would never accomplish.

Having come unravelled, I couldn't pull myself together, and after a few weeks, I reluctantly realised I needed help. My doctor prescribed anti-depressants that effectively put me to sleep for about twenty-three hours out of every twenty-four. After six weeks I decided enough was enough and took myself off the meds for good.

A curious thing had happened in the intervening weeks though. Whilst I was asleep, my mind appeared to have wrapped itself around the enormity of what was happening. This wasn't any kind of revelation, but it was a recognition of what I was up against, and that was enough for the time being.

Over the ensuing months I gave up work as my sight continued to deteriorate, and I stayed at home and tried to keep busy. Most people look baffled when I tell them that going stir crazy was one of the hardest things to deal with about losing my sight. People who know me understand that to me, boredom is a fate worse than death (or blindness as it turns out)!

There were days when I raged out of control. Days when I screamed and threw things at the people I loved just because they were there. There was the day I stumbled in the kitchen and upended a draining rack full of crockery that smashed into a thousand pieces around me, the days when I demolished a keyboard I could no longer use, or kicked the hell out of the hoover because the cable was so caught up around the furniture I couldn't untangle it, days too numerous to count when I bruised, cut, scratched, or burned myself in pursuit of everyday tasks, and the rage and the tears would overwhelm me all over again.

That amount of fury isn't a good thing. It took my relationship with the most important person in my life to the brink of collapse, but fortunately, patience is one of his abiding qualities even though I tested it to the limit during those times. My friends and family went through this every bit as much as I did, only they managed it with a degree of grace, humour, and affection I was incapable of recognising at the time. Now I know with absolute certainty that if it hadn't been for their collective love and support, things would have turned out very differently for me.

Towards the end of that year, not long before Christmas, the last of my sight vanished. To think of it now, it seems that I went to bed one night aware of a slight red smudge at the farthest reaches of my vision (the standby light on the television), then woke the next morning to nothing at all. I don't suppose it happened exactly like that, but it's close enough.

I do remember being surprised to learn that only 3% of blind people are completely blind. Most have some degree of light perception or even a little usable vision, but I'm one of the few who can see nothing at all, and nothing is the best way to describe it. People assume it must be like closing your eyes or being in a dark room, but it's not like that at all. It's a complete absence of light, so it isn't black or any other colour I can describe.

To offset what would otherwise be an incredibly boring view, my mind obligingly gives me things to look at instead. It shows me a shadowy representation of what it thinks I would see if I could — like my hands holding a cup of tea in front of me. Since my mind is constrained only by my imagination, it rather charmingly overlays everything with millions of tiny sparkles of light, that vary in brightness and intensity depending on my emotional state.

My retinas are long since gone, so no actual light makes its way to the backs of my eyes. This is what gives my eyes their peculiar look — each pupil is permanently open to its fullest extent in an effort to take in light. Oddly this is the one thing that still makes me feel a little uncertain about being blind, but given that I no longer really remember what I look like, perhaps there will come a time when that uncertainty will fade.

With the last of my sight gone, I discovered something I hadn't expected. Now that I couldn't see anything, things started to get a lot easier to deal with. Looking back now, I realise that was because I stopped trying to look at what I was doing and started to use my other senses.

I'll pause at this point to clear up a common misconception — I do not have extraordinary hearing, sense of smell, or any other sense. I do pay more attention to those other senses, though, so although I'll often hear a phone ringing when other people don't, it's only because I'm devoting more of my concentration to listening than they are.

The next few months were a time of discovery, sometimes painful, often frustrating, but also littered with good memories. I learned how to do chores, how to cook, where to find audio books, how to cross the road, what it feels like to drink too much when you can't see straight to begin with, and many more things I'd learned once in my life before. The one thing I didn't do was learn Braille, at least not with any dedication. I simply didn't have the patience to go back to reading baby books, at a time when so much else was new and strange. I did learn something far more important though.

I discovered something called a screen reader, a piece of software that could be installed on my computer and which would speak on-screen content to me in a synthetic voice. It would even echo my keystrokes back to me as I typed, which was just as well because I suddenly realised I couldn't touch-type despite having used computers on/off since the early 1980s!

I then decided to enrol in an online course with the Open University. The course was called, "You, your computer, and the Internet", a subject that was child's play for someone who had been using the Internet for nearly a decade. For someone who could barely use a computer with a screen reader, it was something of a practical challenge, though. It took me an entire day to figure out how to log into the course environment, and for a while, every new task seemed to take as long. Day by day things got easier, though, and by the time I finished the course, I was well on my way to regaining an important part of my life. Somewhere along the way, I'd also rediscovered my love of learning and promptly enrolled in another course. One thing led to another, and I eventually graduated with a degree in computer science in 2010.

It's been fifteen years since all this happened. Somewhere along the way I went back to work, and I now have the uncommonly good fortune to be working and collaborating with lots of smart and interesting people, many of whom I'm delighted to call friends.

I still find technology challenging sometimes because we have yet to reach a time when things are engineered to be accessible as standard. That, too, is changing.

So life moved on, as life has a habit of doing, and as I celebrated my fortieth birthday last year, perhaps it gave me cause to reflect. ■

Léonie is a Senior Accessibility Engineer with The Paciello Group (TPG) and owner of LJWatson Consulting. Amongst other things she is co-chair of the W3C Web Platform Working Group, and a member of the ARIA and SVG working groups. In her spare time Léonie blogs on tink.uk, writes for Smashing magazine, SitePoint.com and Net magazine. She also loves cooking, dancing and drinking tequila (although not necessarily in that order).

How I Quit My Job and Built My First App

And Turned My App Into a Sustainable Business

By ROBLEH JAMA

This story starts back in the summer of 2009. I had just gotten my first iPhone 3GS, and I was loving it. Think way back, back to the time when Angry Birds wasn't a hit on the App Store yet. Apple had launched the App Store in 2008, the same year I had sold my previous startup. I was working full-time at a large software company, but I wasn't very Intuit (into it). I was getting extremely excited about the mobile space and looked into starting to make apps.

On a more personal note, in 2009, I had just gotten married and was expecting my first daughter. I wanted to make an app that she would love and use.

I planned to publish my first app under a company name. My wife and I were coming up with names on our way back from the midwives, where we'd just listened to our unborn baby's heartbeat. It was a big moment. I thought out loud, "Why don't we call the studio Tiny Heartbeats?" My wife said that was good, but Tiny Hearts sounded better. She was right.

Tiny Hearts

The old Tiny Hearts logo

With the name settled, I started exploring the App Store and using a lot of different apps. I grew obsessed deciphering the ingredients that made some apps climb to the top of the charts while I watched others disappear. The App Store became my school. One of the apps I really loved in 2009 was Live Cams Pro, an app that allowed users to watch live streams of their security cameras on the go. More interestingly, users could view various streams of public live camera feeds at traffic lights, cities, and airports. Live Cams Pro was briefly the number-one paid app before Angry Birds.

Live Cams Pro

It wasn't the best-designed app, but I loved the live cameras of animals at the zoos the most. At the time, I lived close to the Toronto Zoo and went there regularly with my wife (we had yearly passes). My passion for animals made me curious enough to see if I could make the best animal app in the App Store.

When I was thinking of this app concept, I looked at what jobs people might use it for. The main goal of the app would be to educate children about animals in an entertaining and engaging way. As a secondary, but not insignificant benefit, the app would buy parents five minutes of peace and quiet. Children will love it, but parents will pay for it, so it was great that the app could complete jobs for both audiences. The goal was to create an app that kids would love to use, and parents would love to buy.

I was a soon-to-be parent at this point, so I had a good sense of what my target market was thinking and what their plans and interests were. If you're not your own target market, then you should go talk to members of your market and validate your ideas from there. (You could consider co-creating your product with members of that community. It'll be challenging, but well worth it. We'll save that story for another day).

I can't overstate the importance of understanding your target market. It's your job to know your target market's problems, fears, aspirations, and what they want in an app or digital product. It's not the customer's job to know what they want. In a way, I was building the app for myself and my daughter — which is why I was in charge of the feature set, the designs, and various other product decisions.

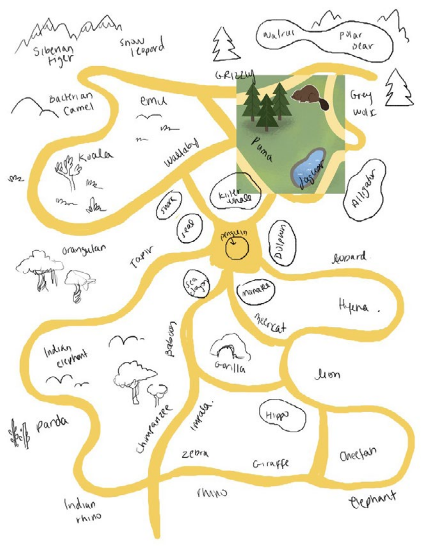
I decided to create a virtual zoo that would fit in people's pockets — hence the name of my animal app, Pocket Zoo. It was going to educate and entertain children and nature lovers (like myself). I resolved to create the best app to solve that specific problem. We created a beautiful virtual zoo with over fifty animal illustrations. We also curated some of the best animal content online from photos to videos and live cams. We made it educational by creating easy-to-read facts for each animal, which was where we spent most of our time.

How We Built Pocket Zoo

Pocket Zoo app icon starring Pingwin. We spent a lot of time trying to get this just right.

Pocket Zoo app screens post iOS 7 update

After this idea came about, I just focused on executing. I got in touch with the co-founders of my previous startup, Rob Chia and Mohamed Hashi, and hired them on contract. I also collaborated with two illustrators that I found after doing some digging around. I found a designer friend to help with our branding and video. My wife helped out with content. I even got my teacher friends involved to approve it and make sure it'd be good for children.

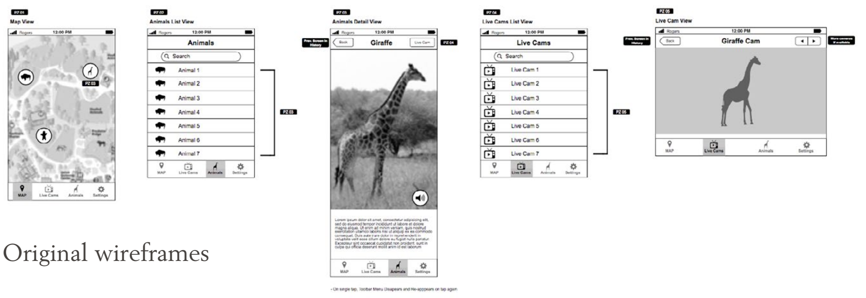


Early Pocket Zoo map divided into zoogeographic locations inspired by the Toronto Zoo

Tiger
Snow Leopard
Grizzly Bear
Walrus
Polar Bear
Beaver
Emu
Camel
Grey Wolf
Puma
Alligator
Wallaby
Killer Whale
Koala
Jaguar
Tapir
Seal
Manatee
Penguin
Leopard
Shark
Dolphin
Orangutan
Sea Dragon
Meerkat
Hyena
Baboon
Gorilla
Lion
Indian Elephant
Impala
Hippo
Chimp
Panda
Cheetah
Zebra
Lemur
ZOO
Indian Rhino
Giraffe
Elephant

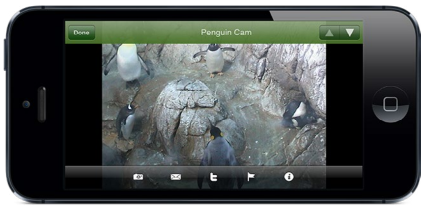
Final Pocket Zoo map

In the App Store, design is your marketing. How your app looks will get you in the door, and usefulness will keep you in the user's phone.



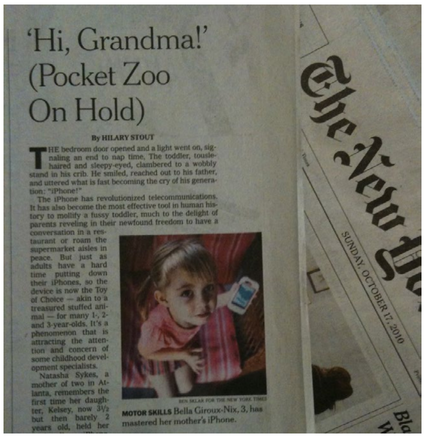
Original wireframes

When you understand mobile, you realize what you choose not to do often is just as important as what you choose to do. You can't cram everything into this little screen. At Tiny Hearts, [tinyhearts.com] we try to focus each of our apps to do just one thing really, really well. It's little coincidence that the most successful apps not only look great but are focused and intuitive.



This is what the live cams feature looked like in 2010

Back to the story — I wanted to have this app ready for the App Star Awards, and submissions were due either March or April of 2010. In order to qualify for this award, we needed to have a forty-five-second demo. The App Star awards were started by Oriel Ohayon from AppsFire, and it's where I was first introduced to the work of Mills and the crew at Ustwo and Jeremy Olson from Tapity — who had both submitted apps to the competition. If we won, Pocket Zoo would be seen by folks like Robert Scoble and get mentions on blogs like Techcrunch, TUAW, and Read Write Web, which meant great exposure down the line. We ended up being the runner-up and received a bunch of press because of it. As a side note, creating external deadlines is also a great way to rally a team.



'Hi, Grandma!' (Pocket Zoo On Hold)

By HILARY STOUT

THE bedroom door opened and a light went on, signaling an end to nap time. The toddler, tousle-haired and sleepy-eyed, clambered to a wobbly stand in his crib. He smiled, reached out to his father, and uttered what is fast becoming the cry of his generation: "iPhone!"

The iPhone has revolutionized telecommunications. It has also become the most effective tool in human history to mollify a fussy toddler, much to the delight of parents reveling in their newfound freedom to have a conversation in a restaurant or roam the supermarket aisles in peace. But just as adults have a hard time putting down their iPhones, so the device is now the Toy of Choice — akin to a treasured stuffed animal — for many 1-, 2- and 3-year-olds. It's a phenomenon that is attracting the attention and concern of some childhood development specialists.

Natasha Sykes, a mother of two in Atlanta, remembers the first time her daughter, Kelsey, now 3½ but then barely 2 years old, held her

BEN SKLAR FOR THE NEW YORK TIMES

MOTOR SKILLS Bella Giroux-Nix, 3, has mastered her mother's iPhone.

SUNDAY, OCTOBER 17, 2010

Pocket Zoo featured in the New York Times

May came around, and my daughter was born (May 10, 2010 to be exact), the same week that my app was submitted to the App Store (May 5th). I launched Pocket Zoo in the store and did a bit of outreach later that month. I just had a baby and was still focused on the product, so I didn't have a chance to do marketing and outreach earlier before the launch. Pocket Zoo debuted at the end of May and hit a bit of a lull. Fortunately, Apple featured it in June, and the app gained a ton of traction. It made the top fifty paid apps and was the number-one education app for a short time. Pocket Zoo also subsequently got featured in The New York Times, which lifted its traction again. Wired magazine also called it a "must-have app."

More importantly, I got several emails from people of all ages, all over the world — parents, grandparents, children, and teachers. They loved the app and spent hours watching animals on Pocket Zoo. They left awesome reviews, which inherently is a huge reward. If I had to choose, I would rather hear from real people using my apps than any award, accolade, or press coverage. (No disrespect, obviously.)

Taking the Leap

After my daughter was born, I started my parental leave. As that was wrapping up, I had a choice to make — either to take the leap and keep doing apps full-time or to go back to my day job. I made enough money for my family to survive, so I was in a good position to quit my job.

This decision is different for people in different situations with different risk tolerances and aversion. When the Pocket Zoo iPad app came out, I knew there was no turning back. I quit the big company.

That's not to say it was an easy decision. A few months after deciding to do apps full-time, Pocket Zoo wasn't generating consistent revenue, and my bank account started feeling it. (I'll talk about how I solved that problem in the section below, where Pocket Zoo evolves into Tiny Hearts studio.)

Here's a major reason why I left: I knew that in order to succeed, I'd have to take the app business seriously. Even though apps might seem like fun and games, you can't win if you come to it lightly. Building an app can be a hobby. Making a living from apps is not. Think business, not apps. You can't be a Sunday developer and be in business. Apple says in its review guidelines:

> *If your app looks like it was cobbled together in a few days, or you're trying to get your first practice app into the store to impress your friends, please brace yourself for rejection. We have lots of serious developers who don't want their quality apps to be surrounded by amateur hour.*

Pocket Zoo's initial momentum continued after I quit my job, and I had more time to make the most of it. Although I did marketing late, Pocket Zoo was inherently compelling for Apple, and the press loved it. These influencers wanted to share it with people. The decisions I'd made with product and design came to make order-of-magnitude differences in word-of-mouth marketing.

At the time, Pocket Zoo was also very unique; new things get news. Even though Pocket Zoo was very niche — for children and animal lovers — it was still unique and new. Today, there are millions of apps out there, so it's more difficult to be unique and new. Don't get discouraged. There will always be room for more good apps. There are also lots of new ideas out there, and there are lots of new ways of doing old things.

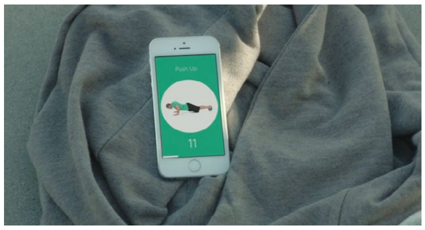
From Pocket Zoo to Tiny Hearts studio

tiny hearts

New Tiny Hearts logo. The four quadrants represent the types of products we strive to create: beautiful, playful, useful, and meaningful

I'd mentioned my original intention to launch under a company name earlier in this piece. From the get-go, I knew that I would need a studio in order to make apps for a living. I couldn't rely on just Pocket Zoo or any other one app; I needed to diversify.

The studio model fits my DNA well because my brain doesn't like to do one thing. I wanted to make different things. I knew I wanted to create something else after I launched Pocket Zoo. I would go on to launch a game, an alarm clock app (that hit number one in its category in the App Store), a fitness app (that was featured in an Apple commercial), and an iOS keyboard (that recently peaked at number two on the paid apps chart). I followed my curiosity as I hopped into these different categories, but it was also a great way to learn the art and science of creating apps through experimentation.



Quick Fit as it appeared in the international Apple "Strength" TV commercial

Through the years, I figured out other ways to make money with Tiny Hearts. Some collaborators had great ideas but lacked mobile expertise, so I would work with them to bring their vision to life. Pocket Zoo organically led to us working on educational games with edtech and media companies. Quick Fit opened the door to work on software for wearables with clients. The cash flow I got by providing services work for big corporations, start-ups, and non-profits made bootstrapping a bit less stressful. It empowered us to be less reliant on investors and fluctuating app revenues. Most importantly, we got to learn a lot from projects that fulfill and excite us.

Today, Tiny Hearts is made up of over a dozen full-time team members and contractors working on our own products and products for clients. Our mission at Tiny Hearts is to make people's lives better in small, meaningful, ways.

We make apps people love, by making apps we love.

You can find a lot of Tiny Hearts' DNA, in terms of processes and values, in Pocket Zoo. We eventually had to put Pocket Zoo to sleep (another story for another day), but it will forever remain the foundation that we built the rest of our apps and services business on. ■

The Tiny Hearts team — team work makes the dream work

Robleh Jama is the founder of Tiny Hearts, [tinyhearts.com] an award-winning product studio. They make their own products like Next Keyboard, Wake Alarm, and Quick Fit — as well as products for select clients. You can follow him on Twitter *@robjama*

Metrics and monitoring for people who know what they want

We know from experience that monitoring your servers and applications can be painful, so we built the sort of service that we would want to use. Simple to set up, responsive support from people who know what they're talking about, and reliably fast metric collection and dashboards.

Dashboards **StatsD** **Happiness**

Now with Grafana!

Why Hosted Graphite?

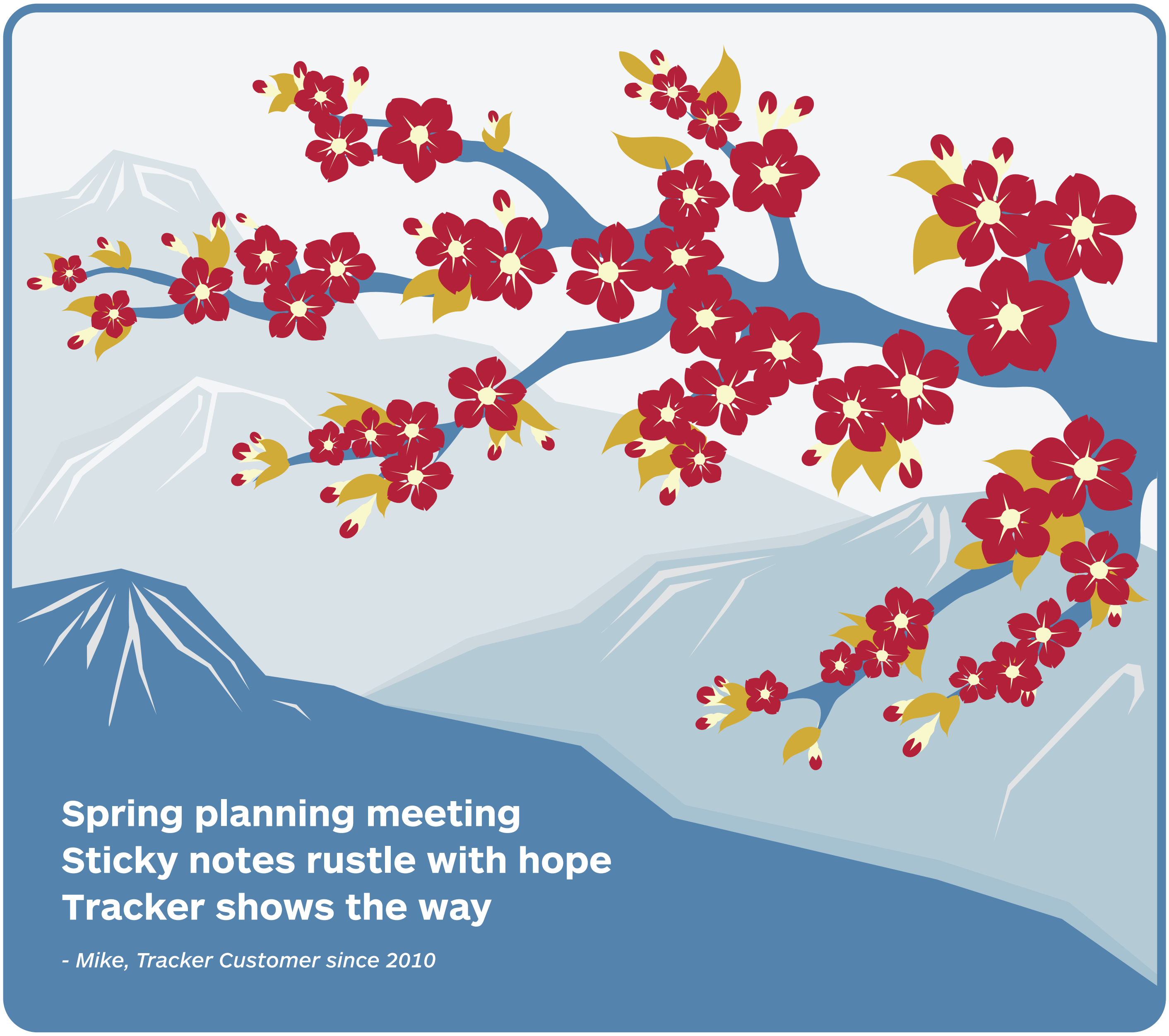
- **Hosted metrics and StatsD:** Metric aggregation without the setup headaches
- **High-resolution data:** See everything like some glorious mantis shrimp / eagle hybrid*
- **Flexible:** Lots of sample code, available on Heroku
- **Transparent pricing:** Pay for metrics, not data or servers
- **World-class support:** We want you to be happy!

Promo code: **HACKER**

Grab a free trial at http://www.hostedgraphite.com

*Hosted Graphite's mantis shrimp / eagle breeding program has been unsuccessful thus far

HOSTEDGRAPHITE


Spring planning meeting
Sticky notes rustle with hope
Tracker shows the way
- Mike, Tracker Customer since 2010

Discover the newly redesigned Pivotal Tracker

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused, and reflect the true status of all your software projects.

With a new UI, cross-project funcionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at pivotaltracker.com.

PivotalTracker

Build better software, faster.