Tom Booth

Painting in Clojure

HACKERMONTHLY Issue 52 September 2014

Cover image generated with Clojure: *github.com/tombooth/painting-in-clojure*

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Carlos Bueno
Tom Booth
Tyler Langlouis
Aaron Bull Schaefer
Justin Palmer
Thomas Burette
Isabelle Park
Paul Buchheit

Proofreaders
Emily Griffin
Sigmarie Soto

Ebook Conversion
Ashish Kumar Jha

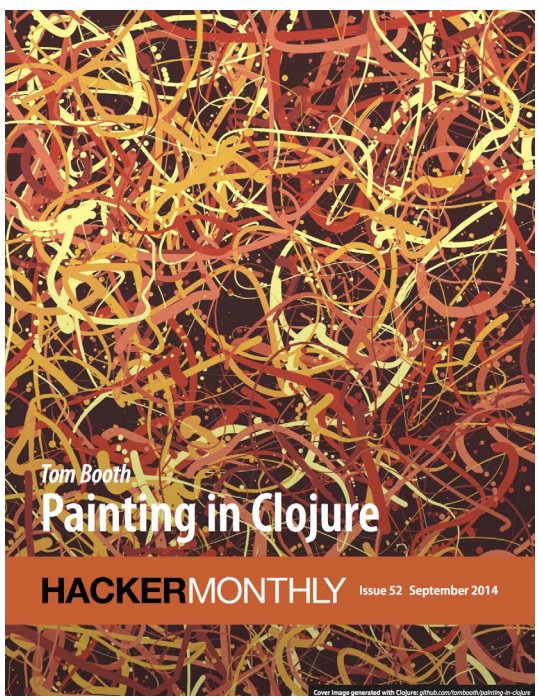
Printer
MagCloud

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 image generated with Clojure: *github.com/tombooth/painting-in-clojure* with modifications on the pallette and dimension. Read more in "Paiting in Clojure".

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-52*

Inside the Mirrortocracy

By CARLOS BUENO

"It is difficult to get a man to understand something when his salary depends upon his not understanding it." — Upton Sinclair

There's a problem with Silicon Valley and the subcultures that imitate it. It's a design bug woven into people's identities and sense of self-worth. Fixing it will be painful. Influential and otherwise very smart people will deny till their last breath that it even exists. But I believe it should be fixed before it gets any worse.

Since credentials are so important these days, here are mine. I'm a programmer, and a good one. I've worked at several companies that went on to be acquired and one that went IPO. I've founded companies and conducted hundreds of interviews. I've written well-respected books, am regularly invited to speak, and have been honored by the White House. I've devised novel ways to optimize billion-dollar computer clusters. You've almost certainly run code that I wrote.

I wouldn't make it past the résumé screen if I were starting my career today.

About twenty years ago I enrolled in a dropout-prevention program at my high school. It allowed me to attend class only half the day. In the afternoons I worked at a startup. The early '90s were a wild time. Any idiot who could spell "HTML" could get a job, and I was one of them. It didn't matter that I had half a high school diploma and no driver's license. They gave me a shot and I ran with it.

The general quality and professionalism of programming has gone up since then. That is a good thing. That's not the problem.

The problem is that Silicon Valley has gone completely to the other extreme. We've created a make-believe cult of objective meritocracy, a pseudo-scientific mythos to obscure and reinforce the belief that only people who look and talk like us are worth noticing. After making such a show of burning down the bad old rules of business, the new ones we've created seem pretty similar.

"The notion that diversity in an early team is important or good is completely wrong. You should try to make the early team as non-diverse as possible." — Max Levchin

Max Levchin, a founder of PayPal, preaches that mythos to young hopefuls who want to follow his success. His thinking is actually more subtle than that quote, but subtlety and introspection are not common traits among young people out to make a lot of money in a short period of time. Encouragement from billionaire heroes leads to even more insularity.

Because the talent market is tight, that insularity presents a problem. It's hard to find good people to hire. All the Stanford graduates have offers from multiple companies and there's no time to develop talent. On the other hand, so many nice-seeming candidates seem to fail the interview process for trivial mistakes that fall under the catch-all category of "culture fit."

Nerdsplaining

The solution, of course, is not self-reflection or asking hard questions about the values and assumptions that form the process. The solution is to write "explainer" blog posts to initiate candidates into the Culture. As the hiring crunch gets more desperate, examples of this genre are more frequent. They are fascinating documents of just how disconnected insiders have become from the very people they are trying to hire. Here's an excerpt from the blog of a San Francisco startup:

> *I asked her how she was doing in the interview process and she said, "I'm actually still trying to get an interview."*
>
> *"That's weird." I told her. "I thought you had already met with them a few times."*
>
> *"Well, I grabbed coffee with the founder, and I had dinner with the team last night, and then we went to a bar together."*
>
> *I chuckled. She was clearly confused with the whole matter. I told her, "Look, you just made it to the third round."*

Clearly, the confusion is *her* fault, right? Let's review the bidding. A capable professional expressed interest in working for a company. Instead of talking with her about that in plain English, she was held at arm's length for days while The Culture examined her for defects: coffee dates in the afternoon, conversations over dinner. When she gets the invisible nod, her reward is a "spontaneous" invitation to a night of drinking with the team. You have to wonder why intelligent people would devise an interview process so strange & oblique that the candidate doesn't even know it's happening.

On the surface there's nothing wrong with getting to know a job candidate in a relaxed setting. But think about who might flunk this kind of pre-interview acculturation. Say, people who don't drink. Or people with long commutes, or who don't have the luxury of time to stay out late with a bunch of twenty-somethings on a whim. Or, perhaps, people who don't like the passive-aggressive contempt shown to those who don't get with The Culture.

Ignorance of The Culture is a serious handicap if you want to land a job out here. Another story from same post is very tense. (The emphasis is mine.)

> *We had a gentleman over to interview for one of our account executive positions... **great resume, great cover letter, did well in our initial phone screen.***
>
> ***He was dressed impeccably in a suit...** I stole a glance to a few of the people from my team who had looked up when he walked in. **I could sense the disappointment.***
>
> *It's not that we're so petty or strict about the dress code that we are going to disqualify him for not following an **unwritten rule**, but **we know empirically** that people who come in dressed in suits rarely work out well for our team.*
>
> ***He was failing the go-out-for-a-beer test and he didn't even know it...***
>
> *I told him he could take off his tie and jacket and loosen up a little bit, and he acknowledged that he felt a little out of place but said that, "you can never overdress for an interview."*
>
> *Well, dude, no, actually you can overdress for an interview and you just did. **Of course I didn't say it...***

The cognitive dissonance on display is painful to see. Clothing is totally not a big deal! Because we're cool like that! But it's plain that it biased the interviewers. The team's disappointment upon seeing the suit was immediate and unanimous. If you truly believe that suit = loser, you can't help it. Nevertheless, the fiction of objectivity has to be maintained, so he denies it to the candidate's face, to us, and himself.

Remember that the *entire point of his article* is to convince candidates to look and act differently: "it's your responsibility to learn [our] cultural norms." Presumably that same account exec is supposed to take the hint, dress in mufti, and do better at his next startup interview. But of course, how you dress is totally not a factor in the scientific decision process.

Even if you take his statements at face value, they make no sense. Suppose that it's a scientific fact that wearing a suit signals that a candidate is unfit for duty. Assuming that's true, then what does teaching the poor bastard how to camouflage himself actually accomplish? Does clothing indicate a person's inner qualities or not? What, exactly, is the moral we're supposed to learn from this grubby little drama?

The theme is familiar to anyone who's tried to join a country club or high-school clique. It's not supposed to make sense. The Culture can't really be written about; it has to be experienced. You are expected to conform to the rules of The Culture before you are allowed to demonstrate your actual worth. What wearing a suit really indicates is — I am not making this up — *non-conformity*, one of the gravest of sins. For extra excitement, the rules are unwritten and ever-changing, and you will never be told how you screwed up.

> *"PayPal once rejected a candidate who aced all the engineering tests because for fun, the guy said that he liked to play hoops. That single sentence lost him the job."*
> *— Max Levchin*

Clothing is the least of it. Your entire lifestyle and outside interests are under examination, as is your "commitment." Say you're asked out for coffee on short notice, which you decline because you're busy. Is that a "ding"? Did that lose you the job? Who knows? Maybe it did. You're still trying to figure out what they mean by "wowing" them. Should you ask? Maybe you'll seem desperate if you ask. Oh, shit!

The obscurity and arbitrariness are very much by design, and is why explainer posts are supposed to be so valuable. Having engineered an unfair situation, insiders then offer secret guides to winning it.

How to make it in the Valley

As far as I can tell, these are the seven rules to follow if you're going to have a chance at being snubbed by a Valley Culture startup. The initial gauntlet is not as harsh if you possess trendy technical skills but that is by no means a free ticket.

1. **Live in the Valley.** If you don't, move. The pioneers who are connecting the global human family and removing barriers of time and space won't take you seriously unless you brunch at the same restaurants they do. Ideally you should live in "The City," which is on a peninsula, and not on "The Peninsula," [en.wikipedia.org/wiki/San_Francisco_Peninsula] which is in a valley.

2. **We expect you to click with us "organically," which means on our schedule.** Be flexible with your time. It's best to behave as though you have nothing better to do all day but wait for us to call you in for coffee or some skateboarding.

3. **Don't overdress, but don't underdress.** You should mirror as precisely as possible our socioeconomic level, social cues, and idiom. Remember unlucky Mr. Hoops. But no pressure, you know? Laid back.

4. **To distinguish yourself from the throngs, find a way to surprise us that has nothing to do with your ability to perform your job.** Maybe you could bring some appropriately quirky luxury foods as tribute.

5. **You are expected to read everything we blog about and work it into the conversation.** This shows commitment.

6. **We don't actually want to talk to you.** You need to locate someone else in our social circle and convince them to send us a "warm intro." This is a wonderfully recursive time-waster, as those people will want a warm intro from someone they know before talking to you, and so on.

7. **We're objective meritocratic folks and will violently reject any suggestion that we are not.** We totally won't "ding" you for not doing steps 1-6, we swear. But they help. Totally.

Watch yourself

The problem with gathering a bunch of logically-oriented young males together and encouraging them to construct a Culture gauntlet has nothing to do with their logic, youth, or maleness. The problem is that all cliques are self-reinforcing. There is no way to re-calibrate once the insiders have convinced themselves of their greatness.

It's astonishing how many of the people conducting interviews and passing judgment on the careers of candidates have had no training at all on how to do it well. Aside from their own interviews, they may not have ever seen one. I'm all for learning on your own but at least when you write a program wrong it breaks. Without a natural feedback loop, interviewing mostly runs on myth and survivor bias. "Empirically," people who wear suits don't do well; therefore anyone in a suit is judged before they open their mouths. On my interview

I remember we did thus and so, therefore I will always do thus and so. I'm awesome and I know X; therefore anyone who doesn't know X is an idiot. Exceptions, also known as opportunities for learning, are not allowed to occur. This completes the circle.

You can protest your logic and impartiality all day long, but the only honest statement is that we're all biased. The decisions of parole judges, professionals who spend their entire careers making decisions about the fate of others, are measurably affected by whether they had just eaten lunch [hn.my/lunch]. And that's with a much more rigorous and formal process whose rules are in the open. But you're sure your process is totally solid, right?

If spam filters sorted messages the way Silicon Valley sorts people, you'd only get email from your college roommate. And you'd never suspect you were missing a thing.

> *"I want to stress the importance of being young and technical. Young people are just smarter."*
> *— Mark Zuckerberg*

I was in the audience when a 22-year-old Zuck led with that drop of wisdom during his first Startup School talk. It wasn't a slip of the tongue, it was the thesis of his entire 30 minutes on stage. It would have been forgettable startup blah-blah except that his talk followed Mitch Kapor's. The contrast could not have been more raw. Ironically, Zuck had arrived late and didn't hear Kapor speak. He's since evolved his views, thanks to Sheryl Sandberg's influence and (ahem) getting older himself.

Kapor is the legendary founder of Lotus, which more or less kicked off the personal computer revolution by making desktop computers relevant to business. He spoke about the dangers of what he called the "mirror-tocracy": confirmation bias, insularity, and cliquish modes of thinking. He described the work of his institute [kaporcenter.org] to combat bias, countering the anecdotes and fantasies that pass for truth with actual research about diversity in the workplace.

The first step toward dissolving these petty Cultures is writing down their unwritten rules for all to see. The word "privilege" literally means "private law." It's the secrecy, deniable and immune to analysis, that makes the balance of power so lopsided in favor of insiders.

Calling it out and making fun of it is not enough. Whatever else one can say about the Mirrortocracy, it has the virtue of actually working, in the sense that the lucky few who break in have a decent rate of success. Compared to what? Well, that is carefully left unasked. The collateral damage of "false negatives" is as large as it is invisible. But it is difficult to argue with success. It takes a humility and generosity that must come from within. It can't be forced on others, only encouraged to develop.

Lest you get the wrong idea, I'm not making a moral case but a fairly amoral one. It's hard to argue against the fact that the Valley is unfairly exclusionary. This implies that there is a large untapped talent pool to be developed. Since the tech war boils down to a talent war, the company that figures out how to get over itself and tap that pool wins.

So the second step is on you. Instead of demanding that others reflect your views, reflect on yourself. Try to remember the last time someone successfully changed your mind. Try, just for a moment, to suppose that it's probably unnatural for an industry to be so heavily dominated by white/Asian middle-class males under 30 who keep telling each other to only hire their friends. Having supposed that, think about what a just future should look like, and how to get there.

You want a juicy industry to disrupt? How about your own? ■

Carlos Bueno is an engineer at Facebook. He writes occasionally about programming, performance, internationalization, and why everyone should learn computer science.

Reprinted with permission of the original author. First appeared in *hn.my/mirror* (bueno.org)

Painting in Clojure

By TOM BOOTH

Learning Clojure by building a digital Jackson Pollock. This article and the source code backing it can be found on GitHub. [github.com/tombooth/painting-in-clojure]

Jackson Pollock

He was an abstract artist who lived through the first half of the 20th century and is most famous for his drip paintings. This style of painting involves him using brushes, sticks, and cans to apply paint to the canvas with the motion of his gestures causing the artworks to come alive. You can get a good idea of how this comes together from this YouTube video. [hn.my/pollock]

Setting the scene

We want to define some facts about the space that our digital Pollock will work in. These facts will not change over the execution of our model and fit Clojure's preference for immutability perfectly. For those who have not come across the idea of mutability before, it is simply whether something can be changed in place. In most languages if you set the label `some_number` to equal `5`, further on you can increment the value of `some_number` to 6 or even 7. In Clojure if you tried to increment `some_number` you would get a new value rather than changing `some_number`.

Clojure will let us define facts using one of the following value types:

- A number. This could be 5 (an integer), 3/2 (a ratio/fraction), or 3.14 (a floating point number).
- A string, represented as a sequence of characters, for example "Hello world!"
- A keyword, which consists of very similar to strings preceded by a colon, e.g. `:an-identifier`. As alluded to in the example, they are usually used for identifiers or labels and do not allow spaces.
- A list `(...)`. This is a way of grouping values into a collection with an explicit order. You may notice all of the code written takes the form of lists. By default if you have written `(...)`, Clojure will assume the first item is a function and the rest are arguments to be passed in. In order for the list not to be executed, you should prefix it with a `'`.
- A vector `[...]`. This is a lot like a list except that it is optimized for appending to the end of the sequence rather than to the front.
- A set `#{...}`. If you are not particularly bothered by the order of the values stored in your collection then you can use a set.
- Lastly there are maps `{...}`. These store pairs of values where the first is a key and the second is a value.

The most important fact about the space is its size. We will use meters to measure the size, only converting to pixels when we need to draw to the screen. We are going to define size as a vector containing its width, height, and depth.

```
(def space [8   ;; width
            5   ;; height
            6]) ;; depth
```

We need to know the gravity of the space so it can influence the flow of paint as it leaves the brush. This will be defined as a vector that represents acceleration influenced by gravity.

```
(def gravity [0 -9.8 0])
```

Lastly, we need to know the normal of the surface of the canvas that the paint will impact with. This will be used to dictate how paint acts when it spatters from the impact with the canvas.

```
(def canvas-normal [0 1 0])
```

Starting points and projection

Our digital Pollock is going to start a stroke of the brush by picking a random point in space. This point will then be projected to find where it impacts with the canvas.

In order to generate a random point inside of the space, we need to define a function that will emit a vector containing the position of the point each time it is called. Function values can be created by calling `(fn [...] ...)` with the first vector being the arguments the function should receive and any follow items in the list are the body of the function and are executed with the arguments bound. Rather than calling `(def name (fn ...))`, Clojure has provided a shortcut function `(defn name [...] ...)`. An example of a defined function is `(defn say-hello [name] (str "Hello " name))`, this creates a function called say-hello that when called `(say-hello "James")` will return the string "Hello James".

We are going to cover a common functional idiom when dealing with lists to change the dimensions of the space above into a random point inside that space. To do this, we want to iterate over each dimension of the size of space, generate a random number between 0 and the magnitude of each dimension and then return the resultant list of random numbers as a list. To generate a random number in Clojure, we can use the `(rand)` function, which will return a random number between 0 (inclusive) and 1 (exclusive). The rand function can take an optional parameter `(rand 100)`; this will define the bounds of the number generated to 0 and 100.

The function map `(map [fn] [sequence])` will iterate through the sequence executing the function with the current value of the sequence as its first parameter, the values returned from the function will be stored in a list the same length as the sequence and returned by the function.

We can now define a random point inside of space as follows:

```
(defn starting-point [] (map rand space))
```

Now that we can generate a random point in space, we want to project this onto the canvas. We are going to use Newtonian equations of motion. We know the position, velocity, and acceleration of the point, and we want to know what the position and velocity are when y is 0. In order to work out final positions, we need to know the total time the point spent falling. We can do this using the y position, since we know that the final position should be 0.

To work out the time it takes for the point to reach the canvas, we will solve the following equation for t:

- r = final displacement
- r_0 = initial displacement
- v_0 = initial velocity
- a = acceleration
- t = time

$$r = r_0 + v_0 * t + at^2/2$$

This rearranges to:

$$at^2 + 2v_0t + 2r_0 - 2r = 0$$

We can solve this using the Quadratic Equation, but this will yield us two results. In general we can say that we are interested in the result with the maximum value.

In the next block of code you can see an example of call out to Java(Script). Clojure doesn't have an in-built square root function, so we are calling out to the Java(Script) version. A function named in the form `foo/bar` means it will call the function `bar` in the namespace `foo`. You might be wondering, what is a namespace?

All good languages need a way to bundle up code that is related so that it can be reused and accessed only when needed. Clojure's take on this is to provide namespaces. Every Clojure source file will declare its namespace at the top of the file so that other files can reference it, extract values, and use functions. Given that Clojure is a hosted language, its namespace will be related to packages in Java and Google Closure Library namespaces in JavaScript.

When hosted on Java, all of `java.util.*` is automatically imported. When on JavaScript, assorted core and Google Closure Library modules are imported. Both of these languages provide us with a Math namespace which contains a `sqrt` function.

```
 (defn time-to-canvas [position velocity accel-
eration]
  (let [a acceleration
        b (* 2 velocity)
        c (* 2 position)
        discriminant (- (* b b) (* 4 a c))
        minus-b (- 0 b)
        add-sqrt (/ (+ minus-b (Math/sqrt
discriminant)) (* 2 a))
        minus-sqrt (/ (- minus-b (Math/sqrt
discriminant)) (* 2 a))]
    (max add-sqrt minus-sqrt)))
```

We can now calculate the time to impact, but we want the final position and velocity. For position we can use the same function that we rearranged above to derive the time.

```
(defn position-at [time initial-position ini-
tial-velocity acceleration]
  (+ initial-position
     (* initial-velocity time)
     (/ (* acceleration time time) 2)))
```

For velocity we can use another equation of motion:

$v = at + v_0$

```
(defn velocity-at
[time initial-velocity acceleration]
  (+ (* acceleration time) initial-velocity))
```

These functions we just implemented can be joined up so that, given an initial position and velocity, we can return the final position and velocity. This function doesn't explicitly ask for the acceleration acting on the paint. It assumes only gravity is acting using the constant defined earlier on.

```
(defn project-point [position velocity]
  (let [[i j k]         position
        [vi vj vk]      velocity
        [ai aj ak]      gravity

        time            (time-to-canvas j vj aj)

projected-position [(position-at time i vi ai)
                    0  ;; we don't need to
                       ;; calculate as it should
                       ;; be 0, on the canvas
                    (position-at time k vk ak)]

projected-velocity [(velocity-at time vi ai)
                    (velocity-at time vj aj)
                    (velocity-at time vk ak)]]
[projected-position
projected-velocity]))
```

Paint splatter

An important aspect of Pollock's painting is the splatter of the paint hitting the canvas and what this adds to the images. We are going to add a simple splatter model based of the velocity at impact we calculated in the last part.

Not all paint that hits the canvas will splatter, so we need to work out the impact force of the paint and use this as a cutoff for whether the paint should splatter.

We will work out the impact force of the paint by taking the velocity at impact and calculating the force required to reduce that velocity to 0 over a set impact distance.

```
(def impact-distance 0.05)
```

We can now use the work-energy principle to calculate the impact force. On one side of the equation we will have the forces at play, and on the other side we'll have the energy:

- F_i = impact force
- d = impact distance
- m = mass
- g = gravity
- v = velocity at impact

$-F_i d + mgd = 0 - \frac{1}{2} mv^2$

This equation can be rearranged to:

$F_i = mg + mv^2/2d$

For simplicity of code, we are just going to consider the y axis, since this is most important when it comes to working out the impact force of the paint into the canvas. The above equation can therefore be expressed as:

```
(defn impact-force [mass velocity]
  (let [y-gravity (second gravity)
        y-velocity (second velocity)]
    (+ (* mass y-gravity) (/ (* mass y-velocity
y-velocity)
                        (* 2 impact-distance)))))
```

Based on this function for calculating the impact force, we can define a predicate that will tell us whether paint should splatter based on its mass and velocity. It is idiomatic in Clojure to end predicates with a ?. We are going to add some randomness to this function so that we don't necessarily just get a uniform line of points. Also defined is a minimum force for us to consider whether some paint could splatter.

```
(def min-impact-force-for-splatter 30)

(defn does-impact-splatter? [mass velocity]
  (and (> (impact-force mass velocity)
       min-impact-force-for-splatter)
       (> (rand) 0.8)))
```

If an impact splatters, then we will need to bounce its velocity vector, since this is the direction in which it will leave its current position.

The equation to bounce a vector, V, off a plane with normal, N, is:

- N is the normal vector of the plane
- V = the incoming vector
- B is the outgoing, bounced, vector

$B = V - (2 * (V . N) * N)$

We are missing a few of the required vector operations used in this equation, so we should define some more functions before trying to implement it. The first is the vector dot product. This is defined as the sum of the multiples of each dimension. Otherwise we need subtraction of two vectors and a function to multiply a vector by a constant.

```
(defn dot-product [vector1 vector2]
  (reduce + (map * vector1 vector2)))

(defn vector-subtraction [vector1 vector2]
  (map - vector1 vector2))
```

This function will introduce a shorthand for defining functions that is very useful in combination with functions like `map` and `reduce`. Rather than writing `(fn [args...] body)`, you can use `#(body)`, and if you want access to the arguments, use `%n` where `n` is the position of the argument. If you are only expecting one argument, then you can use just `%` on its own.

```
(defn vector-multiply-by-constant
[vector constant]
  (map #(* % constant) vector))
```

Using the above functions we can now implement the vector bouncing equation. I have pulled $B = V - (2 * (V . N) * N$ out into a variable called extreme for clarity.

```
(defn bounce-vector [vector normal]
  (let [vector-dot-normal
(dot-product vector normal)
    extreme (vector-multiply-by-constant normal
(* 2 vector-dot-normal))]
    (vector-subtraction vector extreme)))
```

When an impact splatters, it will only take a fraction of the velocity; this is an inelastic rather than elastic collision. We can define a constant that will be used to reduce the total velocity of the bounced vector to reflect this elasticity.

```
(def splatter-dampening-constant 0.7)

(defn splatter-vector [velocity]
  (let [bounced-vector (bounce-vector velocity
canvas-normal)]
    (vector-multiply-by-constant bounced-vector
                splatter-dampening-constant)))
```

Paths vs Points

All gestures Pollock makes are fluid paths, even if the velocity along the path might be rather erratic. We now need to work out how to generate a path of points that we can project and splatter using the code we have written above.

A Bezier curve is a commonly used curve for generating a smooth curve that can be scaled indefinitely, allowing us to have as many points along our path as we care to calculate.

Bezier curves are defined by a list of control points, so we need to be able to generate a potential unbounded list of random control points that should give us limitless different paths to paint.

In order to generate a list of control points, we need to:

- Get a random number between two points for distance and steps
- Get a random unit vector for the initial direction of the generation
- Add vectors together to move between our control points

```
(defn random-between [lower-bound upper-bound]
  (+ lower-bound (rand (- upper-bound
lower-bound))))
```

Below is an algorithm that will give well-distributed random unit vectors. It was ported from code found in the GameDev forums. [hn.my/ruv]

```
(defn random-unit-vector []
  (let [asimuth (* (rand) 2 Math/PI)
        k (- (rand 2) 1)
        a (Math/sqrt (- 1 (* k k)))
        i (* (Math/cos asimuth) a)
        j (* (Math/sin asimuth) a)]
    [i j k]))

(defn vector-add [vector1 vector2]
  (map + vector1 vector2))
```

Now that we have a random direction in which to move, we need to generate an unbounded path that will move in that direction, but randomize the position of each point within provided bounds.

First, we can define a function that will generate a random vector inside of lower and upper bounds that can be combined with the non-randomized position to provide a randomized path.

```
(defn random-vector-between [lower upper]
  [(random-between lower upper)
   (random-between lower upper)
   (random-between lower upper)])
```

In order to provide an unbounded path, we can use a lazy sequence. This function returns a value that is somewhat akin to a list that never ends. Every time you try to look at the next value in the list, it will generate another one.

In this function the first value returned should always be the initial starting position, and each following value should be a step along the path. You can see this below: it returns the position argument cons'd with another iteration of random-path with the position randomized.

```
(defn random-path [position step-vector bounds]
  (cons position
        (lazy-seq (random-path
   (vector-add (vector-add position step-vector)
   (random-vector-between (- 0 bounds) bounds))
                          step-vector bounds))))
```

We can now use this random-path lazy sequence to generate a list of control points given an initial starting point and some bounding variables. The distance, step, and variation allow us to request long winding paths or short flicks.

```
(defn control-points [position min-distance max-
distance min-steps max-steps variation]
  (let [direction       (random-unit-vector)
        distance        (random-between min-dis-
tance max-distance)
        steps           (random-between min-
steps max-steps)
        step-vector     (vector-multiply-by-con-
stant direction (/ distance steps))
        random-positions (take steps (random-
path position step-vector variation))
        end-position    (vector-add position
                                    (vector-
multiply-by-constant step-vector steps))]
    (conj (vec random-positions) end-position)))
```

In order to turn this list of control points into a list of points that represent a path we need an algorithm. The most commonly used is a recursive algorithm proved by De Casteljau. There is a great video on YouTube explaining this algorithm [hn.my/casteljau] that I recommend you watch.

At the core of the algorithm is an equation that will return a point along a line weighted by a variable, t which dictates how close it is to each end of the line:

$$P = (1 - t)P_0 + tP_1$$

For example, if a line runs from P_0 to P_1 and t is 0 then the outputted point with be equal to P_0 and if it is 1 then P_1.

De Casteljau's algorithm recursively creates a new set of points by using the above equation for a fixed t against all the lines created by the control points. It does this until there is just a single point; this is a point on the Bezier curve. It's t from 0 to 1 and for each step gets a point along the curve.

```
(defn recur-relation [t a b]
  (+ (* t b) (* a (- 1 t))))

(defn for-component [t component-vals]
  (if (= (count component-vals) 1)
    (first component-vals)
    (for-component t
      (map #(recur-relation t %1 %2) component-
vals (rest component-vals)))))

(defn for-t [t components]
  (map #(for-component t %) components))

(defn de-casteljau [control-points step-amount]
  (let [x-vals (map first control-points)
        y-vals (map second control-points)
        z-vals (map #(nth % 2) control-points)
        points (map #(for-t % [x-vals y-vals
z-vals]) (range 0 1 step-amount))]
    points))
```

This can generate paths that go below the canvas, so we should set these to 0 as it is the equivalent of painting on the canvas.

```
(defn ensure-above-canvas [path]
  (map (fn [[i j k]] [i (if (< j 0) 0 j) k])
path))
```

Motion, going through the paces

All the points along the generated path should have an associated velocity. To start with we can generate a linear velocity along the path, given a randomized total time to traverse the path and the total length of the path.

In order to calculate the length of the paths, we will want to do something similar to a map but with pairs of values. Using this we can take two points, calculate the distance between them, and then sum all the distances.

```
(defn map-2 [f coll]
  (when-let [s (seq coll)]
    (let [s1 (first s)
          s2 (second s)]
      (if (not (nil? s2))
        (cons (f (first s) (second s)) (map-2 f
(rest s)))))))
```

In order to find the distance between two points, we need to subtract the two vectors, square and sum the resultant dimensions, and then take the root.

```
(defn vector-multiply [vector1 vector2]
  (map * vector1 vector2))

(defn distance-between-points [point1 point2]
  (let [difference-vector (vector-subtraction
point1 point2)
        summed-vector (reduce + (vector-multiply
difference-vector difference-vector))]
    (Math/sqrt summed-vector)))

(defn path-length [path]
  (reduce + (map-2 distance-between-points
path)))

(defn vector-divide-by-const [vector const]
  (map #(/ % const) vector))

(defn velocity-between [point1 point2 total-time
total-distance]
  (let [difference-vector (vector-subtraction
point1 point2)
        time-between (* total-time (/ (distance-
between-points point1 point2)
                                      total-distance))]
    (vector-divide-by-const difference-vector
time-between)))
```

This calculation will leave off the last point's velocity, so we can just set it to 0.

```
(defn path-velocities [path total-time]
  (let [total-distance   (path-length path)
        number-of-points (count path)]
    (conj (vec (map-2 #(velocity-between %1 %2
                                         total-time
                                         total-distance)
                      path))
          [0 0 0])))
```

In addition to the velocity at each point along the path, we also need to know the quantity of paint falling. Again to keep life simple we are going to model this as a linear flow along the path with there always being no paint left.

```
(defn path-masses [path initial-mass]
  (let [number-of-points (count path)
        step (- 0 (/ initial-mass number-of-
points))]
    (take number-of-points (range initial-mass 0
step))))
```

Putting it all together

I've pulled a bunch of colors that Pollock used in his seminal work "Number 8" so that each flick of paint can be rendered in a random color from this palette.

```
(def canvas-color [142 141 93])

(def paint-colors
  [[232 51 1]
   [248 179 10]
   [247 239 189]
   [29 16 8]])

(defn pick-a-color []
   (nth paint-colors (rand-int (count
paint-colors))))
```

Now we need to assemble all of the above functions into something that resembles Jackson Pollock applying paint to a canvas. We start with a point, project a path, work out masses and velocities, project, and then splatter. This is all then packaged up with a color for drawing onto our canvas.

```
(defn fling-paint []
  (let [position       (starting-point)
        total-time     (random-between 1 5)
        path           (ensure-above-canvas (de-
casteljau (control-points position 0.1 2 3 15
0.4) 0.01))
        velocities     (path-velocities path
total-time)
        masses         (path-masses path (ran-
dom-between 0.1 1))
        projected-path (map #(project-point %1
%2) path velocities)
        splatter       (map (fn [[position
velocity] mass]
                              (if (does-impact-
splatter? mass velocity)
                                [position
(splatter-vector velocity) (* mass splatter-
dampening-constant)]
                                nil))
                            projected-path
masses)
        projected-splatter (map (fn [[position
velocity mass :as point]]
                                  (if (nil? point)
                                   nil
                                  (conj (vec
(project-point position velocity)) mass)))
                                  splatter)]
    {:color (pick-a-color)
     :air-path path
     :canvas-path (map #(conj %1 %2) projected-
path masses)
     :splatter (filter #(not-any? nil? %)
(partition-by nil? projected-splatter))}))
```

Rendering the canvas

We need to know the available size for the outputted image to fit in. To work this out we are going to have to interface with JavaScript directly. Luckily ClojureScript makes this very easy using the `js` namespace.

```
(def image-width (.-clientWidth (.querySelector
js/document "#pollock")))
```

Now we have the width of the image we can use the dimensions of the space to work out the pixel size of the image and how to convert between meters and pixels.

```
(def pixels-in-a-metre
  (let [[width _ _] space]
    (/ image-width width)))

(defn meters-to-pixels [meters]
  (Math/floor (* meters pixels-in-a-metre)))
```

We can now use this function to work out the size the sketch should be and how to convert a position in meters over to a position to be drawn in the image.

```
(def sketch-size
  (let [[width _ height] space]
    [(meters-to-pixels width)
     (meters-to-pixels height)]))

(defn position-to-pixel [[i j k]]
  [(meters-to-pixels i)
   (meters-to-pixels k)])
```

Now that the dimensions of the image are calculated, we can use Quil to define the sketch that we will draw into. We also need to define a function that will initialize the image into the state we want it. This function will be run when the sketch is defined.

```
(defn setup-image []
  (apply q/background canvas-color)
  (q/fill 0))

(q/defsketch pollock
  :setup setup-image
  :host "pollock"    ;; the id of the <canvas>
                     ;; element
  :size sketch-size)
```

To draw the trails of paint across the canvas, we need draw a path following the defined positions, taking into account the amount of paint at each position and using this to set the width of the path. In order to do this cleanly in Quil we need to consider the path as pairs of positions that we draw paths between using the initial paint amount as the stroke-weight. This allows for a smooth decrease in the width of the path.

```
(defn mass-to-weight [mass]
  (* 50 mass))

(defn draw-path [path]
  (doall
   (map-2 (fn [[position1 _ mass] [position2 _
_]]
            (q/stroke-weight (mass-to-weight
mass))
            (apply q/line (concat (position-
to-pixel position1) (position-to-pixel posi-
tion2))))
          path)))
```

For splatter we are just going to draw a point that has a stroke-weight proportional to the amount of paint.

```
(defn draw-splats [path]
  (doall (map (fn [[position _ mass]]
                (q/stroke-weight (mass-to-weight
mass))
                (apply q/point (position-to-
pixel position)))
              path)))
```

Now that we can render the result of flinging some paint around, we need a function that will fling the paint and render the result.

```
(defn fling-and-render [& any]
  (q/with-sketch (q/get-sketch-by-id "pollock")
    (let [{:keys [color canvas-path splatter]}
(fling-paint)]
      (q/stroke (apply q/color color))
      (draw-path canvas-path)
      (doall (map draw-splats splatter)))))
```

Lastly, we attach to the buttons, and our image comes to life.

```
(.addEventListener (.querySelector js/document
"#add")
                   "click"
                   fling-and-render)

(def interval-ref (atom nil))
(def fill-count (atom 0))
(.addEventListener (.querySelector js/document
"#fill")
                   "click"
                   (fn [e]
                     (reset! interval-ref
                          (js/setInterval (fn []

(if (> @fill-count 500)

(js/clearInterval @interval-ref)

(do (fling-and-render) (swap! fill-count inc))))
100)))) ■
```

Tom Booth is a developer currently twiddling bits at the Government Digital Service. He spends my days working with Python and Javascript, and his nights playing with Clojure, Golang or anything he can get his hands on.

Reprinted with permission of the original author.
First appeared in *hn.my/painting* (tombooth.co.uk)

SSH Kung Fu

By TYLER LANGLOIS

OpenSSH [openssh.com] is an incredible tool. Though primarily relied upon as a secure alternative to plaintext remote tools like telnet or rsh, OpenSSH (hereafter referred to as plain old ssh) has become a swiss army knife of functionality for far more than just remote logins.

I rely on ssh every day for multiple purposes and feel the need to share the love for this excellent tool. What follows is a list for some of my use cases that leverage the power of ssh.

Public-Key Cryptography

This is kind of a prerequisite for supercharging ssh usage. It's a pretty straightforward concept:

- Generate a public key and private key. The private key can prove ownership of a public key
- Place the public key on any servers you need to log in to
- No more password prompts

Sound good? Let's do it:

```
$ ssh-keygen -t rsa
$ cat ~/.ssh/id_rsa.pub
... your public key here ...
```

Paste that into the `~/.ssh/authorized_keys` file of any server account you need to log in to, then any subsequent

```
$ ssh user@host
```

will be automatic. Now you're prepared to really make ssh useful.

Bonus: ssh-copy-id

As this process of copying public keys is a fairly common task, you can usually get the `ssh-copy-id` command on most platforms do to this automagically for you. For example, if you've generated your keys and want to set up key login for the user bob at the host fortknox:

```
$ brew install ssh-copy-id # (if needed)
$ ssh-copy-id bob@fortknox
```

Done!

Tunneling

Need access to a port behind a firewall? ssh has got you covered. If you need to access the remote endpoint http://no-public-access:80 but can reach that host from another host that you can log in to (let's call it ssh-host), just try this:

```
$ ssh -L 8080:no-public-access:80 user@ssh-host
```

Then browse to http://localhost:8080. Your requests are being routed through `ssh-host` and hit `no-public-access:80` and are routed back to you. Tremendously useful.

Mounting Filesystems

NFS, while venerable, is pretty shoddy when it comes to securely sharing directories. You can share to an entire subnet, but anything beyond that gets tricky. Enter SSHFS, the ssh filesystem.

I've got a machine called `wonka` trying to share the directory `/srv/factory` with the host `bucket`. Assuming that the user `charlie` on `bucket` has keys set up on `wonka`, and assuming that I've got SSHFS available (install it via your package manager of choice if `which sshfs` returns nothing), try this out:

```
$ sshfs wonka:/srv/factory /mnt -o idmap=user
```

Everything is pretty self-explanatory except for the filesystem mount option `idmap`. That just tries to map the Unix numeric IDs from the server to the user you're mounting the filesystem as (it's the least troublesome option.)

And presto, you're sharing a filesystem quickly, easily, and securely. The caveat here is that filesystem changes can take a little longer to propagate than with similar file-sharing schemes like SMB or NFS.

Remote File Editing

This trick relies on the netrw capabilities of vim (a directory browsing function) and SCP (a sister program of ssh.)

Netrw allows vim to browse the filesystem on the local filesystem easily (try it in a terminal with `$ vim [path]`, such as `$ vim ..` However, we can pass vim a remote path using `scp://` to achieve remote file editing:

```
$ vim scp://wonka/
```

You're now browsing your home directory's contents on `wonka` within vim. Any file edits you make will be synced over on write via SCP.

Tab-Completion

Not a trick, but really useful.

Have you ssh-ed into a machine like this in the past?

```
$ ssh supercalifragilisticexpialidocious
```

Good news, you can probably do this:

```
$ ssh sup<TAB>
```

and the shell will auto-complete hostnames drawn from the `~/.ssh/known_hosts` file.

Lightweight Proxy

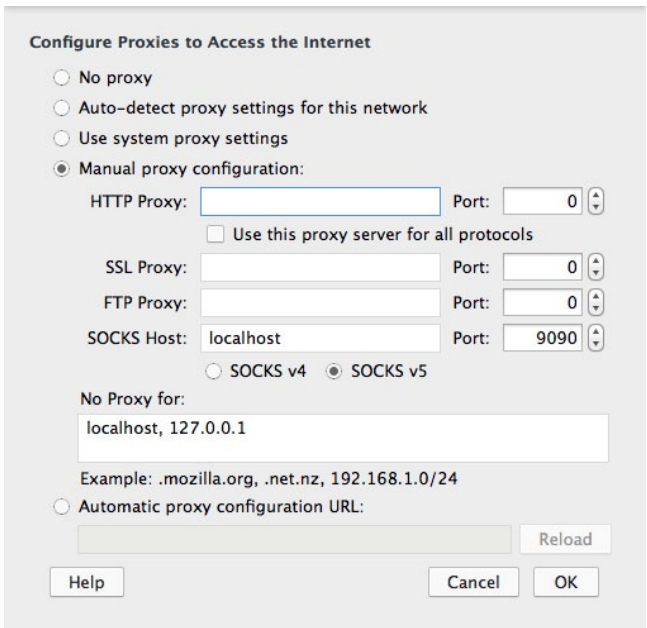
This is probably one of my favorite tricks, and is very useful when the need for it arises.

The previous tunneling example was simple: we're forwarding a local port such as `8080` to a remote endpoint, and ssh handles passing packets through your intermediate ssh host. However, what if you want to place a program entirely behind an ssh host? Doing so enables you to essentially create a very lightweight proxy, independent of some sort of VPN solution like PPTP or OpenVPN.

ssh has an option for this, the `-D` flag. Invoking ssh with the following options:

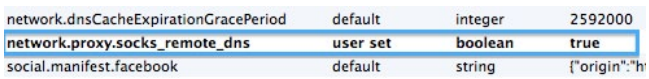
```
$ ssh -D 9090 user@host
```

exposes the local port `9090` as a SOCKS proxy. You can then alter your browser settings to use your local SOCKS proxy to route browsing traffic. My configuration in Firefox for the previous example is shown here:



ssh SOCKS proxy settings in Firefox

Note that for Firefox at least, I had to set the following flag in my http://about:config in order for everything to work correctly:



Firefox SOCKS DNS settings

Try browsing to a site like geoiptool and you should find that your IP is now originating from your ssh host.

This may initially seem a little useless, but when you consider that you can also use a cheap (or free) EC2 instance anywhere in the world… there are a lot of possibilities. For example, I've used this trick to access US-only services when traveling abroad and it has worked very well.

Accessing NAT-ed Hosts Directly

Before diving into this one, I want to emphasize that if you aren't taking full advantage of your `~/.ssh/config`, you really should. Stop using `alias` and starting using shorthand hostnames in the ssh config file. Doing so allows you do use the same shortname in any ssh-friendly application (SCP, rsync, vim, etc.) and lends itself well to providing a unified environment if you distribute your dotfiles across machines.

Moving on. ssh config files follow this syntax:

```
Host foobar
    Option value
```

For example, you can reduce the command

```
$ ssh -p12345 foo@bar.baz.edu -i ~/.ssh/
customkey
```

to

```
$ ssh bar
```

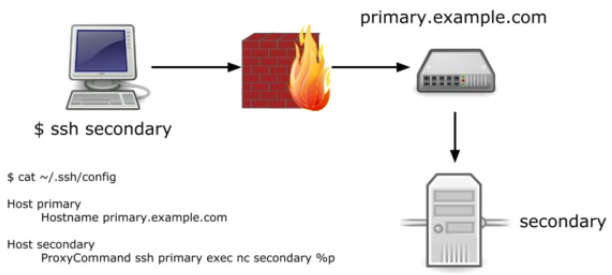
by adding the following to `~/.ssh/config`:

```
Host bar
    User foo
    Port 12345
    IdentityFile ~/.ssh/customkey
    HostName bar.baz.edu
```

Now that we've dabbled in the config, look at this configuration stanza:

```
Host behind.bar
    # ProxyCommand ssh bar exec nc behind %p
    # I've since been educated that invoking
    # netcat for forwarding is deprecated,
    # use the -W flag instead:
    ProxyCommand ssh -q -W %h:%p bar
```

`ProxyCommand` directs ssh how to connect to `behind.bar`: connect to `bar` (previously defined) and connect to the port that ssh would normally connect to. ssh's `-W` flag intelligently forwards traffic to the interpolated %h (hostname) and %p (port) variable placeholders. The following diagram demonstrates.



ssh ProxyCommand diagram

If you've ever tried to copy a file from a NAT-ed machine, you'll see the usefulness in this: you can essentially treat the NAT-ed host as if there were no intermediate hosts between the ssh client and daemon.

Sharing Connections

I often find that when I'm working on a remote host, there's a pretty good chance I'll need to SCP a file over or log in again over another ssh session. While you could negotiate another asymmetric ssh handshake, why not use your pre-existing connection to make speedier file copies or logins?

Use `Controlpath`:

```
Host busyserver
    Controlmaster auto
    Controlpath ~/.ssh/ssh-%r@%h:%p.sock
```

This means upon first connection to `busyserver`, ssh will create a socket in `~/.ssh/` which can be shared by other commands. If you're using commands like rsync or SCP, this can make repetitive copy tasks much quicker.

Bonus Round!

There's been great reception for this post both on reddit and twitter, so I couldn't help but add some of the great tips that people have been throwing at me.

Credit goes to the ssh savants commenting on this blog post and over at the reddit thread.

Remote File Editing in Emacs

I'm a vim man, so I hadn't the slightest clue that you could leverage ssh equally well using Emacs by opening a file path of the format `//user@host:/file` in order to remotely edit files.

The Secure Shell… Shell

This one was new to me: try the key combination `~C` while in an ssh session and you'll get a prompt that looks like this:

```
ssh>
```

(I have to hit return before the `~C` key combo in zsh, but your shell's behavior may vary.)

Try entering `help` at this prompt to get a summary of the commands you can enter at the prompt. Essentially you can dynamically allocate forwarded ports from within your active ssh session. Who knew?

I've actually used the key combination `~.` many times to kill a hung ssh session but had no idea that I was using this feature of ssh.

Agent Forwarding

I actually use this pretty heavily but failed to mention it because I got lazy.

When you negotiate pubkey authentication with ssh, you're just validating that your key gives you the rights to log in the remote server. What if you want to get from that remote server to another server?

You could use password authentication to get to the other machine (ugly), place your private on the intermediate host (not a good idea to spread your private key around), or you could use agent forwarding.

Agent forwarding allows you to validate against that second machine by verifying that you're the owner of the permissioned private key somewhere down the chain. I don't want to make more diagrams so I'll make some ASCII art. Here are your hostnames:

```
sol --------------> terra --------------> luna
```

Your pubkey from `sol` is listed in `terra:~/.ssh/authorized_keys`. Great! You ssh into `terra`:

```
sol ==============> terra --------------> luna
```

Now you want to get to `luna`. You can get there without your private key on `terra` by using a plain old

```
$ ssh luna
```

Nice! With any key credentials stored on `terra`, you've authenticated to `luna` using the private key stored on `sol`. The only requirements for this feature are:

- On the client (`sol` in this example)
  - Make sure you have the `ssh-agent` program running (check if it's running with `ssh-add -L` to list cached private keys)
  - Cache your key with `ssh-add`
  - Prepend the line `ForwardAgent yes` to your `~/.ssh/config`
- Each intermediate server
  - Ensure the line `AllowAgentForwarding yes` is in `/etc/ssh/sshd_config`

That's all there is. If this fails to work for you for some reason, the most common problem is that your key isn't cached in `ssh-agent` on your local machine (again, confirm it's cached with `ssh-add -L`.)

Note that this technique of caching your key in ssh-agent also alleviates the annoyance of having to unlock a password-protected private key every time it's used by caching it for an extended period (with the associated security/convenience tradeoff of keeping your private key cached in memory.) ■

Tyler Langlois designs, builds, and maintains systems at scale as a site reliability engineer at Qualtrics [qualtrics.com], a research solutions company based in Provo, Utah. Outside of systems engineering, he also enjoys security research, all things Linux, and writes about these topics and more on his blog [blog.tjll.net].

Reprinted with permission of the original author.
First appeared in *hn.my/sshfu* (tjll.net)

A Proper Server Naming Scheme

By AARON BULL SCHAEFER

Here at MNX, we've been busy setting up a brand new data center for our cloud hosted services. We started off as a consulting company providing managed Linux services, which means we have been exposed to a ton of different customer environments and an equal number of schemes for naming equipment… not all of them good. It's a problem that goes back as far as computers have existed, and everyone has their own opinion on the "best" way to name hosts. Most methods start out fine at the beginning, but quickly become unwieldy as infrastructure expands and adapts over time.

Since we're starting fresh with this data center, we wanted to come up with our own naming scheme to address the common problems we've seen elsewhere. We gleaned ideas from numerous sources such as data published by large-scale companies, various RFCs on the topic, and blog/forum posts a'plenty. Taking all of that into account, we've developed some best practices that should work for most small-to-medium businesses naming their own hardware.

First, I'll go over the naming scheme then cover some of the finer points and justification for our choices.

A Records

To start off, name each host (via the appropriate method for your operating system) and set its DNS A record to a randomly chosen word pulled from a list:

```
crimson.example.com.    A
192.0.2.11
```

There are many pools of words to choose from, but the specific word list we recommend comes from Oren Tirosh's mnemonic encoding project. These 1633 words were chosen very specifically to be short (4-7 letters), phonetically different from one another, easy to understand over the phone, and also recognizable internationally. The mnemonic word list should be much less prone to typos and transposed characters when compared to more structural names. A lot of time and research went into these words, and their properties make them ideal for our purpose.

Essentially, the hostname should not have any indication of the host's purpose or function, but instead should act as a permanent, unique identifier to reference a particular piece of hardware throughout its lifecycle (try not to reuse names when hardware dies). This name should be used to physically label the equipment and will mostly be useful to operations engineers, remote hands, and for record keeping. It's also what the reverse DNS PTR record should resolve to.

CNAME Records

Next, assign one or more DNS CNAME records to cover useful functional details about the machine such as geography, environment, work department, purpose, and so on. This is all information that will be mirrored in your CMDB and easily referenced.

The CNAME records are what developers should know and use for interconnecting services. Keeping the structure of these names consistent will lower the mental effort necessary to remember a hostname when you need it…

Standardized CNAME Structure

Start with your registered domain, and segment each piece of additional information as a proper subdomain going down from there. DNS is hierarchical by design, so taking advantage of that will provide us with some benefits later on.

```
<wip>.example.com.    CNAME
crimson.example.com.
```

Specify Geography

After your domain name, add a subdomain referencing the geography of the host. Use the 5-character United Nations Code for Trade and Transport Locations (UN/LOCODE) value based on the address of the host's data center. It covers more specific locations than something like the IATA airport codes, and is still a well-defined standard.

In most cases, you can drop the 2-character country code portion and just use the remaining 3-character location code. That is, unless you have data centers in multiple countries and the locations happen to use conflicting codes, just use nyc.example.com and not nyc.us.example.com.

```
<wip>.nyc.example.com.    CNAME
crimson.example.com.
```

Specify Environment

Next up, specify the environment that the host is a part of:

- dev – Development
- tst – Testing
- stg – Staging
- prd – Production

These should be based on whatever process model you follow for release management…you may have more or less designations as well as environments like sandbox, training, etc..

```
<wip>.prd.nyc.example.com.
CNAME    crimson.example.com.
```

Specify Purpose and Serial Number

Last up, specify the basic category of the host's function and append a serial number:

- app – Application Server (non-web)
- sql – Database Server
- ftp – SFTP server
- mta – Mail Server
- dns – Name Server
- cfg – Configuration Management (puppet/ansible/etc.)
- mon – Monitoring Server (nagios, sensu, etc.)
- prx – Proxy/Load Balancer (software)
- ssh – SSH Jump/Bastion Host
- sto – Storage Server
- vcs – Version Control Software Server (Git/SVN/CVS/etc.)
- vmm – Virtual Machine Manager
- web – Web Server

For the serial number, use zero-padded numbers based on your expected capacity. Plan for expansion, but usually two digits will be more than sufficient.

```
web01.prd.nyc.example.com.
CNAME    crimson.example.com.
```

Increment the serial numbers sequentially and segment them based on the type of server in a particular data center, rather than a globally-unique index. That means you may have a web01 in multiple data centers.

Convenience Names

Beyond the standardized structure, you may want additional CNAME records for convenience; words like webmail, cmdb, puppet, etc..

```
webmail.example.com.    CNAME
crimson.example.com.
```

Special Cases

Networking and Power Equipment

For networking and power equipment, the hardware dictates the purpose and it's not likely that you can just move them without reconfiguration. Knowing that, ignore the random word naming convention and use functional abbreviations for the DNS A record itself:

- con – Console/Terminal Server
- fwl – Firewall
- lbl – Load Balancer (physical)
- rtr – L3 Router
- swt – L2 Switch
- vpn – VPN Gateway
- pdu – Power Distribution Unit
- ups – Uninterruptible Power Supply

…you'll probably want the data center geographical info in there as well. You can still add CNAME records for more specific info like core/dist, public vs. private, etc. if desired.

```
rtr01.nyc.example.com.    A
192.0.2.1
```

Secondary and Virtual IP Addresses

The tricky part with secondary and virtual IPs (used for high availability, web services, network migrations, VLAN tagged traffic, etc.) are that they might be floating and not tied to a specific piece of hardware. That being the case, it's easiest to just assign the functional name directly to the DNS A record and follow the normal naming convention.

Mail and Name Servers

For your mail and name servers, you have to utilize DNS A records since MX and NS records must never point to a CNAME alias. That said, you can have more than one DNS A record, so stick with the regular scheme and add something else for the public MX and NS records to utilize.

```
puma.example.com.     A    192.0.2.20
mta01.example.com.    A    192.0.2.20
```

DNS Configuration

Since we used proper DNS subdomains for each unit of data, we can set the search domains on each host to only pay attention to their own local category of machines:

```
search  prd.nyc.example.com  example.com
```

This makes it convenient when working on the machines, as you can use the shorter version of hostnames to, for instance, ping sql01 rather than having to type in the full ping sql01.prd.nyc.example.com when communicating within a data center.

In general, our naming scheme also allows you to prevent inadvertent information disclosure by publicly exposing only the short random hostname while resolving the functional names solely on the internal network. It's a bit of security through obscurity, but something to consider. (Note: you'd have to tweak the "special cases" naming convention if you want to hide those as well.)

Private Network and Out-of-Band Addressing

You can also take advantage of internal DNS resolution to expose private network addresses and out-of-band/IPMI/iDRAC addresses. The domains should match the other records, but once again, utilize a proper subdomain. Note that best practices dictate not using a fake TLD, as ICANN could register them at any time and combining networks becomes trickier.

Complete Naming Scheme Example

```
crimson.example.com.          A       192.0.2.11
crimson.lan.example.com.      A       10.0.2.11
crimson.oob.example.com.      A       10.42.2.11
web01.prd.nyc.example.com.    CNAME   crimson.example.com.

melody.example.com.           A       192.0.2.12
melody.lan.example.com.       A       10.0.2.12
melody.oob.example.com.       A       10.42.2.12
web02.prd.nyc.example.com.    CNAME   melody.example.com.

verona.example.com.           A       192.0.2.13
verona.lan.example.com.       A       10.0.2.13
verona.oob.example.com.       A       10.42.2.13
cfg01.prd.nyc.example.com.    CNAME   verona.example.com.
mon01.prd.nyc.example.com.    CNAME   verona.example.com.
puppet.example.com.           CNAME   verona.example.com.
nagios.example.com.           CNAME   verona.example.com.

banjo.example.com.            A       192.0.2.104
banjo.lan.example.com.        A       10.0.2.104
banjo.oob.example.com.        A       10.42.2.104
web01.dev.pdx.example.com.    CNAME   banjo.example.com.
martinlutherkingsr.melblanc.kugupu.stevejob.kenkesey.
music.filmhistory.calligraphy.example.com   CNAME   banjo.
example.com.
```

Capacity

This naming scheme will easily support 1500+ global servers. If you have more servers than that, you could add in the geography portion for the random names and then reuse words from the list. The downside being that crimson.nyc.example.com might have a completely different purpose than crimson.pdx.example.com, so there's a bit of a mental barrier. Alternatively, you could expand the initial word list, trying to add words similar in spirit to the mnemonic encoding words.

If you're managing 10,000+ servers, the host is much more likely to only have a single segmented purpose, so ignore everything we've written above and just go with a location-based or functional naming scheme.

Tips & Tricks

- You should remove potentially confusing words like "email" from the mnemonic encoding word list if it's technical jargon for your environment.
- Keep the purpose abbreviations consistent in length and always have the serial number padding match (i.e., don't have 01 some places and just 1 in others; always use the longer 01 for everything).
- The actual purpose abbreviations you use aren't important, just pick a scheme, make sure it's documented, and stick to it.
- It's easiest to keep the purpose abbreviations somewhat generalized, as more detailed information can be pulled from your CMDB.
- No matter what, all info should be in a CMDB and easily accessible!
- Set multiple CNAME records when logical, but keep in mind that the more records you have, the more there will be to maintain.
- Automate as much of this as possible.
- We have written a short script named genhost that can help you to randomly pick and keep track of the words you've used for hostnames.

Conclusion

Our server naming scheme lowers the mental effort required to keep track of machines and makes connecting services and maintaining proper hardware records straightforward. The aspects of a machine that are likely to change over time are contained only within the CNAME records. That means if a server dies, you don't have to go and update all references to that host on other machines, as you can just update the CNAME records to point to a new host altogether. While our scheme does add some complexity up front, it strikes a good balance between usability, maintainability, and support for long-term growth. ■

Aaron Bull Schaefer is a Linux Systems Engineer and habitual Hacker News lurker from Spokane, WA. He can be found at his standing desk and has worked in environments ranging from small-scale startups to large-scale educational institutes.

Reprinted with permission of the original author.
First appeared in *hn.my/servername* (mnx.io)

Finding The Perfect House Using Open Data

By JUSTIN PALMER

Growing up in rural Mississippi had its perks. Most days my brother and I would come home from school, grab a snack from the fridge, and head outside to fish in the pond tucked behind the tree line or play basketball with the neighborhood kids. It was very much a community where everyone knew everyone. I had known many of the kids from my graduating class since second grade and my parents knew their parents since high school and earlier.

As beautiful as my hometown was, it was, like many small towns, economically depressed and void of all but the necessities. As I grew older, I became frustrated by this. We had one small grocery store, one stop light, two movie rental places, and not a single fast food restaurant. We had no book stores, no electronic stores, no big-box stores, and only a couple of places to grab a bite to eat.

Truth be told, the gas station (one of those big ones, forever known as "The BP" long after a name change) was where we picked up most of our take out. When highway 72 was expanded to four lanes, a nearby gas station was converted to a "super" station. It was packed with a pizza place, fresh sub sandwiches, and the best chicken strips that have ever graced the space below a heat lamp. It was the community watering hole.

The lack of access eventually wore on me. As I started to grow my design skills (my Hail Mary escape from factory work), I would hear of my peers in larger cities going out to eat, to the movies, or just having a beer at a nearby bar. Beer, by the way, was illegal where I grew up. Not just the consumption of it, the mere possession of it.

By 2007, after a couple of years on the outskirts of Memphis, TN, I had finally had enough and convinced my wife to move to Portland, OR. We knew very little about Portland at the time. In fact, we knew so little about the Pacific Northwest, we moved in the dead of winter, in a sports car, driving through the Cascade mountain range. It's not exactly something I'm proud of; a hilariously ill-conceived cross-country trip.

When we moved to Portland we decided we wanted to be in the center of it all. There was so much life around us; so much happening relative to our rural upbringing. I wanted to be as close to Downtown as I could be so I could go as often as I liked.

We eventually settled on the Pearl District. A relatively new residential development that was previously occupied by a rail yard. Finally, I would have the access that I once craved. Almost anything I wanted was a short walk or streetcar ride away. All that I wished for growing up, I would have.

Fast forward 7 years and we're still in the Pearl District. We've added a member to our family who now desires access to the things I had while growing up. He wants a yard to play in, a basketball goal, and a proper house where he can get excited without disturbing our neighbors.

It's an interesting scenario. The Pearl District and Downtown aren't exactly teaming with affordable single family homes and the further you move out of the city center, generally the less access you have

to the many things a dense urban area has to offer. But, in reality, I only wished for a couple of things out of a new location.

The journey begins

After thinking about the problem, I decided to list out a set of criteria for a location I would want to live. Other factors will eventually come into play, but I wanted to narrow down the city into "target zones" — that is, zones that meet a set of defined criteria.

- **Walking distance to a grocery store:** Living across the street from a grocery store has spoiled me.
- **Walking distance to a rail stop:** This will allow me to get to other locations in the city without a car relatively quickly. One could argue the bus system is just as good, but I would argue that it isn't and I much prefer rail.

I defined walking distance as ~5 blocks, but ~10 blocks is still a pretty sane distance. I want to be close to a grocery store and close to a MAX or Streetcar stop. Unfortunately, none of the real estate applications I tried had a feature like this so I decided to create what I needed using open data that I had already been working with for some time now.

Gathering the data

We'll need 3 open datasets.

- Open Streetmap polygons
- Trimet rail stops
- Portland building footprints

After downloading the data, we'll want to re-project the building dataset to EPSG:4326 so that all of the data shared the same projection. EPSG:4326 (WGS84) is a common projection and the projection of the other two datasets so I went with that.

```
ogr2ogr -t_srs EPSG:4326 -f "ESRI Shapefile"
building-footprints.shp  Building_Footprints_
pdx.shp
```

Now we need to create a PostGIS-enabled Postgres database:

```
createdb portland
psql -c "create extension postgis" -d portland
```

Finally, we need to import the datasets to our freshly created Postgres database. See the shp2pqsql docs for an explanation of the flags used. Essentially we want to import as latin1 (there are some encoding errors in the building dataset), force 2d geometry, and create an index on the geometry column.

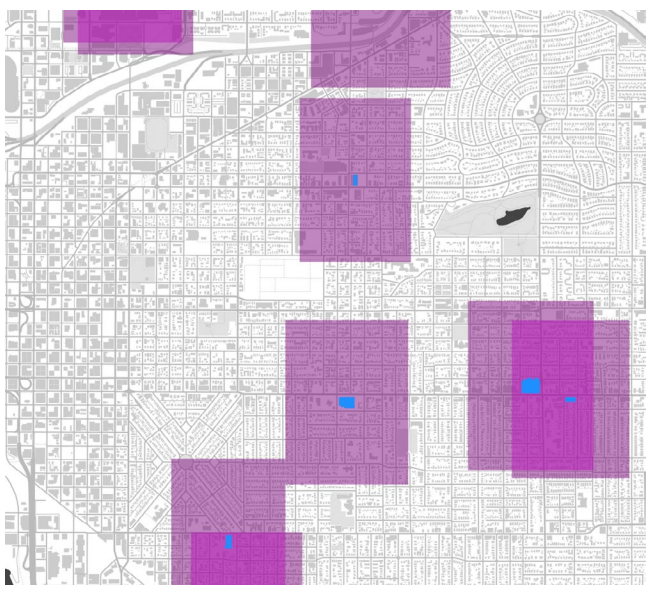
```
shp2pgsql -W "latin1" -t 2D -I -D -d -s 4326
building-footprints.shp building_footprints |
psql -d portland

shp2pgsql -W "latin1" -t 2D -I -D -d -s 4326
osm-polygons.shp osm_polygons | psql -d portland

shp2pgsql -W "latin1" -t 2D -I -D -d -s 4326
trimet-rail-stops.shp trimet_rail_stops | psql
-d portland
```

After we have the data imported into Posgtres, we can begin to find target geometries (buildings) that meet the criteria we set. The first thing we want to do is find the zones around all supermarkets using st_expand. Our units are decimal degrees and 0.0045 is about the desired distance of ~5 blocks. We're not too worried about being a little off here.

```
select st_expand(geom, 0.0045) as zone, 5 as
score from osm_polygons where osm_polygons.
shop='supermarket'
```



At this point, we have large rectangle geometries. As you can tell from the image above, some buildings lie in overlapping zones. We want to score those buildings

for every zone they intersect with. In similar fashion, we want to find the zones around rail stops and public parks.

The next thing we want to target individual buildings instead of merely drawing a large box around a zone. We can find all buildings that intersect a zone using st_intersects.

```
select * from supermarket_zones inner join
buildings on st_intersects(supermarket_zones.
zone, buildings.geom) where buildings.
subarea='City of Portland'
```

Finally, we want to group identical geometries, sum the score, and stuff the target buildings into a new table. We don't necessarily need to add the target buildings to a new table, but continuously running these queries across the entire building set can be slow.

```
select sum(score) as rank, gid, geom into
target_homes from target_buildings group by 2,
3;
```

Here's what everything looks like combined which can be run using psql -f score-buildings.sql -d portland.

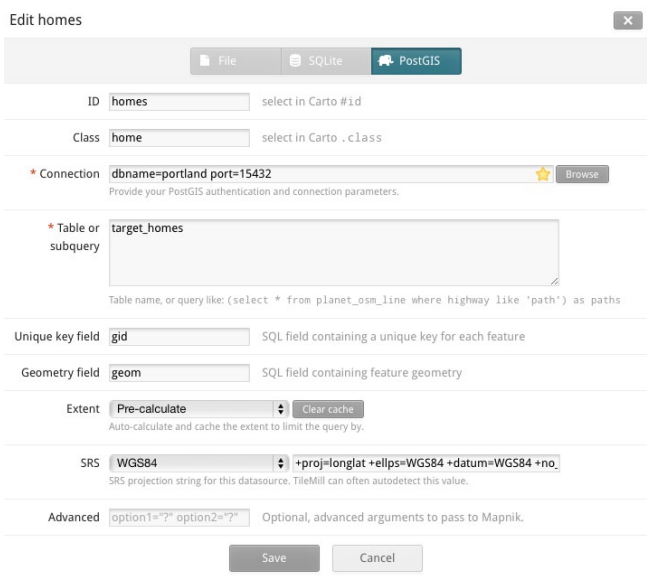
```
drop table if exists target_homes;

with
supermarket_zones as (select st_expand(geom,
0.0045) as zone, 5 as score from osm_polygons
where osm_polygons.shop='supermarket'),
rail_stop_zones as (select st_expand(geom,
0.0045) as zone, 5 as score from trimet_rail_
stops),
park_zones as (select st_expand(geom, 0.0045) as
zone, 2 as score from osm_polygons where osm_
polygons.leisure='park'),
target_buildings as (
  select * from supermarket_zones inner join
buildings on st_intersects(supermarket_zones.
zone, buildings.geom) where buildings.
subarea='City of Portland'
  union select * from rail_stop_zones inner join
buildings on st_intersects(rail_stop_zones.zone,
buildings.geom) where buildings.subarea='City of
Portland'
)

select sum(score) as rank, gid, geom into
target_homes from target_buildings group by 2,
3;
create index target_homes_gix on target_homes
using gist (geom);
```

Styling the results

The last major thing we need to do is to visualize the results on a map. For this, I'll use the open source editor Tilemill by Mapbox. This step is pretty straight forward since we've already done the hard work of extracting and scoring the buildings we're interested in. We only need to supply Tilemill with the name of the table we stored our target homes in.



Finally, let's fill each building with a color based on its score using CartoCSS.

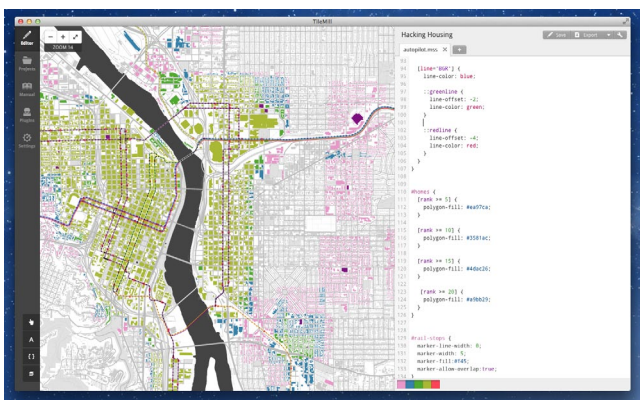
```
#homes {
  [rank >= 5] {
    polygon-fill: #ea97ca;
  }

  [rank >= 10] {
    polygon-fill: #3581ac;
  }

  [rank >= 15] {
    polygon-fill: #4dac26;
  }

   [rank >= 20] {
    polygon-fill: #a9bb29;
  }
}
```

We can add more layers to help give some context to our results. Here's what I ended up with after adding layers for rail lines, rail stops, and supermarkets.



And that's it. Now I have a good idea of locations I can check out in my quest to find a house nearby a supermarket and rail line. Nothing like quickly grabbing a beer for a backyard barbecue and a little game of basketball. And hey, when the Trail Blazers play? I'll just walk a few blocks and hop on the MAX. ■

Justin is a Designer and Engineer at GitHub who lives and works from Portland, OR.

Reprinted with permission of the original author.
First appeared in *hn.my/househunt* (dealloc.me)

So You Want To Write Your Own CSV Code?

By THOMAS BURETTE

So you want to write your own CSV code? Fields separated by commas and rows separated by newline. Easy right? You can write the code yourself in just a few lines.

Hold on a second...

What if there are commas inside the fields?
You need to enclose the field with quotes ("). Easy right?

But can only some fields but not all be quoted?

What if there are quotes in the fields?
You need to double each instance of quote in the field, and god forbid you forget to enclose the field in quotes.

Also make sure not to mistake a quoted empty field (...,"",...) for a double quote.

What if there is a newline inside a field?
Of course you must enclose the field using quotes.

What are the accepted newline characters?
CRLF? CR? LF? What if there are multiple newlines?

What if the newline characters change?
For example, what if newlines within a fields are different from newlines at the end of a line?

Still with me?

What if there is an extra comma at the end of a line?
Is there an empty field at the end, or is that just a superfluous comma?

What if there is a variable amount of field per line?

What if there is an empty line?
Is that an EOF, a single empty field, or no field at all?

What about whitespace?
What if there are heading/trailing whitespaces in the fields?

What if the CSV you get always has a space after a comma but it's not part of the data?

What if the character separating fields is not a comma?
Not kidding.

Some countries use a comma as decimal separator instead of a colon. In those countries Excel will generate CSVs with semicolon as separator. Some files use tabs instead of comma to avoid this specific issue. Some even use non-displayable ASCII characters.

Don't forget to account for it when reading an arbitrary CSV file. No, there's no indication which delimiter a file uses.

What if the program reading CSV use multiple delimiters?
Some program (including Excel) will assume different delimiters when reading a file from the disk and reading it from the web. Make sure to give it the right one!

What if there is non ASCII data?
Just use utf8 right? But wait...

What if the program reading the CSV uses an encoding depending on the locale?
A program can't magically know what encoding a file is using. Some will use an encoding depending on the locale of the machine.

Meaning if you save a CSV on a machine and open it on another, it may silently corrupt the data.

Do you really still want to roll your own code to handle CSV?

CSV is not a well-defined file-format. The RFC4180 does not represent reality. It seems like every program handles CSV in subtly different ways. Please do not inflict another one onto this world. Use a solid library.

If you have full control over the CSV provider, supplier, and the data they emit, you'll be able to build a reliable automated system.

If a supplied CSV is arbitrary, the only real way to make sure the data is correct is for a user to check it and eventually specify the delimiter, quoting rule....Barring that, you may end up with an error, or worse, silently corrupted data.

Writing CSV code that works with files out there in the real world is a difficult task. The rabbit hole goes deep. Ruby CSV library is 2321 lines. ■

Thomas Burette is a Software Developer who has lived and worked in Brussels, Ottawa and Paris. When not crafting robust applications for clients, you'll find him traveling thousands of miles on a bicycle and musing on various personal software projects.

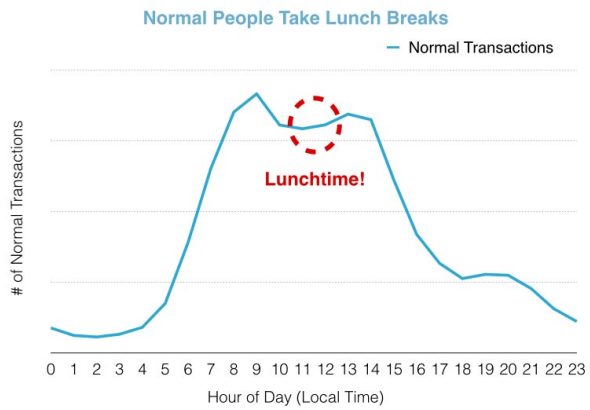
Reprinted with permission of the original author. First appeared in *hn.my/csv* (tburette.github.io)

Seven Habits of Highly Fraudulent Users

By ISABELLE PARK

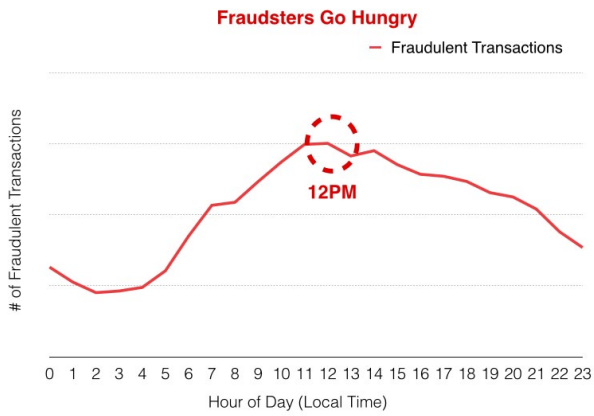
At Sift Science, we analyze a lot of data. We distill fraud signals in real-time from terabytes of data and more than a billion global events per month. Previously, we discovered that the U.S. has more fraud than Nigeria and solved the mystery of Doral, FL. At our "Cats N' Hacks" Hackathon last week, I decided to put some of our fraud signals to the test. Working with our Machine Learning Engineer, Keren Gu, we discovered some interesting fraud patterns[1]:

Habit #1: Fraudsters Go Hungry



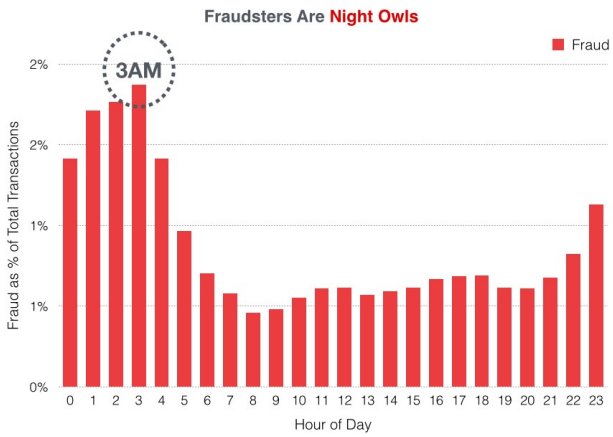
When we looked at total non-fraudulent (normal) transactions by hour, normal users had slow starts to their mornings. We noticed a slight dip in transaction volume around lunchtime and suspect that's because people are taking lunch breaks! Happily fed, they resumed activity in the afternoon and activity petered out as users went home for the day.

What about fraudsters?



Fraudsters, however, work through lunch. We don't see the same dip in activity during lunchtime in the fraudulent sample. It seems that fraudsters are too busy scheming their next move.

Habit #2: Fraudsters Are Night Owls

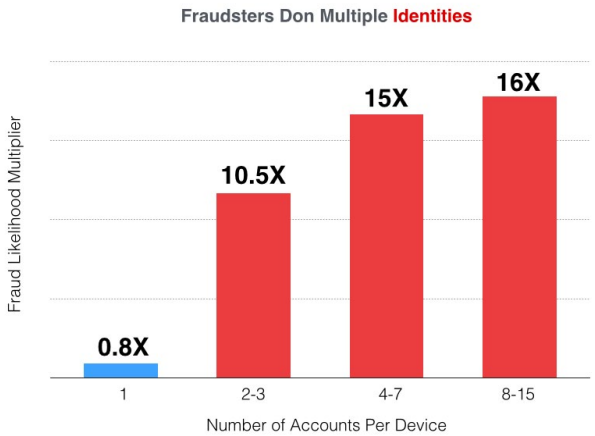


When we analyzed fraudulent transactions as a percentage of all transactions, 3AM was the most fraudulent hour in the day, and night-time in general was a more dangerous time. This finding is consistent with our historical findings and it makes sense: fraudsters are more likely to execute attacks outside of normal business hours when employees aren't around to monitor fraud.

Habit #3: Fraudsters Are International

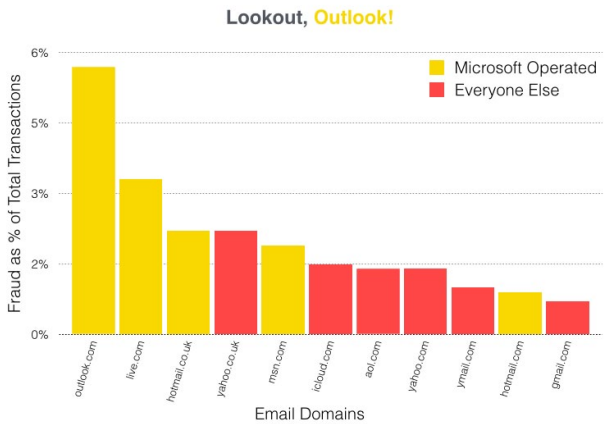
Indian email address domains had one of the highest fraud rates when compared to other top-level domains. However, don't give up on those great Bollywood movies just yet! We're only looking at data from the past three months. We've seen this list fluctuate quite a bit depending on what new tactics fraudsters use.

Habit #4: Fraudsters Don Multiple Identities



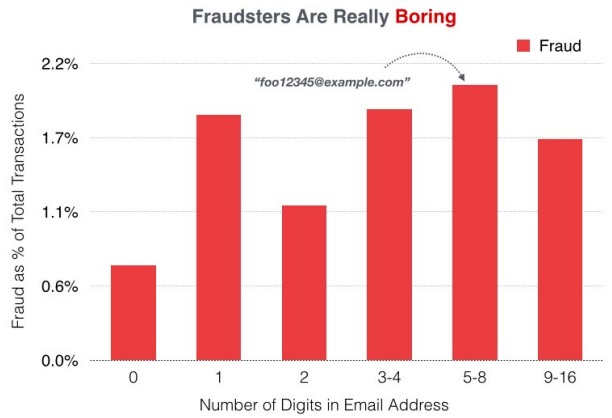
Fraudsters tend to make multiple accounts on their laptop or phone to commit fraud. When multiple accounts are associated with the same device, the higher the likelihood of fraud. The graph above shows how many times more likely a user is fraudulent given the number of accounts associated with the user's device. Phew, that was a mouthful! Said in another way, a user who has 6 accounts on her laptop is 15 times more likely to be fraudulent than the average person. Users with only 1 account however, are less likely to be fraudulent.

Habit #5: Fraudsters Still Use Microsoft



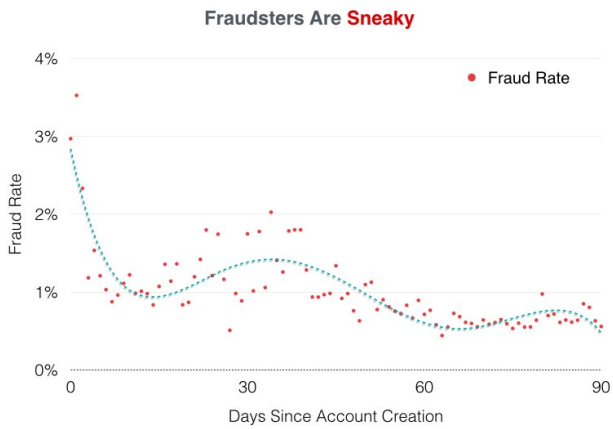
Some of the most fraudulent email domains are operated by Microsoft. Why could this be? Two possible reasons are that 1) Microsoft has been around for a lot longer and 2) email addresses were easier to create back in the day. Today, websites use challenge responses such as image verification or two-factor authentication to verify your legitimate identity.

Habit #6: Fraudsters Are Really Boring



One of the most widely recognized predictors of fraud is the number of digits in an email address. The more numbers, the more likely that it's fraud. Why? Because fraudsters are boring (and lazy). They use computer programs to sequentially generate email addresses so they don't have to think of new ones. Emails such as "foo1234@test.com" or "foo1234568@testing.com" are highly suspicious. However, detecting fraud using email address alone can be really difficult. The only way to really get good at detecting fraud is to look at hundreds of signals, sometimes in the thousands (that's where machine learning can help).

Habit #7: Fraudsters Are Sneaky



Fraudsters like to create disposable accounts that are short-lived. In analyzing the age of fraudulent user accounts (meaning, the amount of time between account creation and a fraudulent transaction), we found that they sign up on sites and then quickly commit fraud. The longer the account age, the less likely the user is committing fraud. Nonetheless, experienced fraudsters know that fraud detection companies track this type of signal. In the graph above, we noticed "sleeper" fraud agents became active after 30 and 60 days of account creation. Fraudsters are sneaky!

Obviously, the above is not a definitive sample set. Data can help us find potential answers as to why fraudsters behave in the ways that they do, but as statisticians say, "correlation is not causation"! It's important to use common sense and human intuition when it comes to dealing with fraud. ■

[1] Data was collected from the past three months over our entire network. From the hundreds of millions of transactions we processed during that time, we analyzed about 6 million. Our "fraud" sample consisted of transactions confirmed fraudulent by our customers; our "normal" sample consisted of transactions confirmed by our customers to be non-fraudulent, as well as a subset of unlabeled transactions. Please keep in mind that every company faces different type of fraud, and that our findings may not be representative of what you see. All transaction timestamps are local to the user.

Izzy is pursuing her MBA at Wharton with a major in Statistics. This summer, she interned at Sift Science, where she applied machine learning methods and visualized big data to predict fraudulent behavior. She envisions a not too distant future where knowledge is democratized for all.

Reprinted with permission of the original author.
First appeared in *hn.my/fraudster* (siftscience.com)

The Technology

By PAUL BUCHHEIT

Note: This is the talk I gave at Startup School Europe.

You've heard a lot of great startup advice today. This is going to be a little different.

I often advise startups that it's better to seek deep appeal, to create something that a few people love, even if most people don't get it right away. In that spirit, I've decided to share the technology and dreams that matter to me, with the hope that it will be very appealing to the right person. This is, after all, a business defined by outliers. Someone in this room is going to create something very important. That's the person I'm hoping to reach.

We talk a lot about technology, and its ability to transform and improve the world. But technology is more than just transistors and algorithms. Those are just patterns on silicon. The technology that really drives the world are the patterns in your head. Those are the patterns that give rise to the patterns in silicon, the patterns in our society, and our whole concept of reality. Change those patterns, and you change your world. Maybe not overnight, but like steering the rudder on a great ship, a small change now makes a big difference later.

We often sweat life's big decisions, but it's the little decisions that matter the most — the ones we make thousands of times a day, often without even realizing it. The big decisions are the inevitable result of those small decisions. They steered the ship into port and created the conditions that gave rise to the situation. And then perhaps we feel that our hand has been forced — the big decision must be made — but really it was made by the thousands of small decisions leading up to it.

We all know the power of defaults. This is about my defaults, the things I keep top of mind and return to when I'm stuck, confused, or doubtful. It's an effort to tune and improve my patterns, my technology.

First, I don't know anything. That's a warning. If you take this all on my authority, then you're missing the point. You must own your own programming.

It's also the first pattern.

If I believe that I already know the answer and possess the truth, then I'm not genuinely open to learning larger truths.

This is the danger of experience. We already know better and we already know that an idea or business won't work. This is one reason that naive, young founders are often the ones who start the most successful companies — they just don't know any better, and they're often too arrogant to listen to those who do.

I don't want to downplay the value of experience. This whole event is about sharing and learning from the experiences of others. But don't be limited by our experiences. Just because it didn't work in the past doesn't mean it won't work in the future. Likewise, what worked before may not work again.

This is especially important for startup founders. The best opportunities live in our collective blind spots. To most, they appear to be bad ideas, or simply unimportant. If everyone could see the opportunity, someone else would have already taken it.

In 1997, Larry and Sergey tried to sell Google for a million dollars. Fortunately, they were unable to find a buyer. The conventional wisdom of the time was that search was neither important, nor valuable.

Of course experience isn't the only danger. Dogma and ideology are even worse. They provide us with the answers, and put boundaries around our thinking. Ignoring the dogma invites ridicule, or even punishment. I suspect that's why more ideological societies are less innovative. If we aren't free to wander outside the realms of conventional thinking, then we won't happen upon the opportunities that others have missed.

Escaping dogma is hard. From the inside, it simply looks like truth and reality. Watch out for any belief that limits the range of your thinking and exploration. This includes logic and reason. They are useful tools, but just as often work to keep us trapped inside of exclusionary belief systems. If you believe yourself to be a rational person, then you're in the trap.

To be innovative in our work, we need to evade the limitations of established thinking.

Which brings me to the second pattern: Kill all daemon processes.

For those of you who aren't familiar with operating system internals, daemons are computer programs that run in the background performing various services, often invisible to the user. Sometimes they get out of control and start consuming all of the machine's memory, processor, or other computing resources. This is one reason why your computer or phone often works better after a reboot.

I like using this as an analogy for the same kinds of loops that operate in our brains, like when a song gets stuck in your head. The more insidious loops are the voices of doubt, anger, and self-loathing that infect our minds. Often they are the internalized voices of our parents, peers, the media, or just random people on the Internet. Other times, they pose as our own voice, possibly one that has been there for as long as we can remember. Either way, these loops are often parasitic and limiting. Anytime we take a risk or move in a new direction, they are there to doubt and criticize us. Anytime we seek to escape dogma, they are there to ridicule and condemn us.

Creating an innovative new product often means spending years working on something that most people doubt the value of. It's hard to do that with a head full of noise, voices telling us that we're being foolish and should just cut our losses.

Before we launched Gmail, many people inside of Google thought that the whole project should be scrapped. One notable executive predicted that we would never even get to a million users. We can't let those voices drag us down.

In order to grow, be free, and reclaim our mental resources, it helps to clear out these voices. It's simple, yet very difficult, because they'll keep coming back. But with practice, we get better.

Right now, stop, observe your breath, and enjoy a moment of stillness in your mind. The voices that keep interrupting the silence are the runaway processes. Keep dismissing them until there aren't any left.

Our days are full of spare moments. Instead of filling them with Flappy Bird or Facebook, take the opportunity to find a calm and clear mind. Even if you don't always succeed, it's the practice that matters. Walking in nature also helps.

The voices will resist of course. Continuing to assert their own importance is one way they survive.

My response: Yes, and thank you. That's the third pattern.

Life rarely goes the way we want it to. When we're taking risks and trying something new, we should expect that it often won't work out the way we had planned. And even if we try to keep our lives narrow and risk free, things still won't work out the way we had planned. We can get angry and frustrated and stuck, or we can accept and move forward, assuming that whatever happened is somehow for the best.

I've found that this is a great predictor of success among startups. They all face setbacks, but some are able to take those setbacks and use them to their advantage. Others just keep slamming their head against the same wall, never making any real progress. Uber has been rather masterful at this. Here in London, they turned the taxi strike into a huge growth opportunity for themselves.

In my own life, I've observed that many of the best things are rooted in some of the worst events, and that I would not have one without the other. But this about the small decisions more than the big ones. Every day is full of setbacks and disappointments, but I do my best to say, "Yes, and thank you," accepting it as a gift, however improbable that may seem at the time. This pattern has an almost magical way

> **If your startup has only one definition of success, then you're setting yourself up for failure.**

of transforming reality and maintaining the forward flow of life.

The ability to accept a greater range of outcomes opens the door to pattern number four: Choose the more interesting path.

People often ask how I decide which startups to invest in. There's no simple answer, but this is a big part of it.

When I heard about Justin.tv in early 2007, my first response was to laugh and ask if they were serious. They said yes, so I offered to invest. The plan at the time was for Justin Kan to attach a camera to his head and stream it live on the Internet, 24/7. It seemed a little insane, but I was very curious to find out what would happen. I've found that that kind of interestingness is a very useful signal.

The immediate answer to, "What would happen?" was a lot of people trolling Justin. Next they added the ability for anyone to stream their lives. Most of it was boring, or possibly illegal, but one thing really caught on: video game streaming. Eventually they changed their name to Twitch.tv to focus exclusively on competitive gaming. They are now one of the most valuable properties on the Internet. Their average daily viewer watches over 100 minutes per day, and they are the 4th largest source of US Internet traffic after Netflix, Google, and Apple.

I had no idea that would happen. I mainly invested because it sounded like an interesting experiment, and the founders seemed to genuinely believe that they were on to something.

Interestingness is a sign of unexplored or under-explored territory. If I already know what the outcome is going to be, that's not very interesting. If it's completely random, like gambling, that's also not interesting. But I find that great startups exist in a space of productive uncertainty. Regardless whether they succeed or fail, I'm likely to learn something interesting.

That was my logic when joining Google in 1999. I expected that they would likely get squashed by the much larger Alta Vista, but the people were really smart, so I believed that I could learn a lot in the process.

In fact, I can guarantee success by simply redefining success to include learning something interesting. In this way, I've always succeeded, and also learned a lot. :)

If your startup has only one definition of success, then you're setting yourself up for failure.

It's tragic how many people are sacrificing their lives on some startup that they don't really care about, in pursuit of some external success they'll likely never achieve. Personally, I think it's a mistake.

Which leads me to pattern number five: Love what you do.

It's often said that you should "Do what you love," but that's mostly bad advice. It encourages people to grind away their lives in pursuit of some mostly unattainable goal, such as being a movie star or a billionaire startup founder. And even if they do make it, often the reality is nothing like they imagined it would be, so they're still unhappy.

Do what you love is in the future. Love what you do is right now. As with the other patterns, it's meant to guide the small decisions that we make every moment of every day. It's less about changing what you do, and more about changing how you do it.

One of the problems with having a goal-oriented, extrinsic mindset is that it treats the time between now and task completion as an annoying obstacle to be endured. If you're doing something that is difficult, uncertain, and takes a long time, such as building a new

> **“I’d rather fail at something awesome, than succeed at something inconsequential.”**

product or company, and you have that mindset, then you’re likely gambling away a big chunk of your life. Subconsciously, you may also compensate by choosing smaller, more realistic goals, and that’s unfortunate. Plus, it’s unpleasant.

When I was working long hours at Google, it wasn’t because they were whipping us to work harder. I would have quit. I was doing it because I genuinely love building things. It wasn’t all fun of course, but I typically enjoyed at least 80% of my day.

“Do what you love” treats “what you love” as a fixed thing, but it’s not. I used to hate running. I would sometimes force myself to run a few miles because it’s supposed to be healthy, but I never liked it. Then I read a book that said we are born to run, and that it can be fun. Inspired, I decided to try running just for fun, focus on the quality of every step, and forget about the goal completion aspect of it. Very quickly, I learned to enjoy running, and over time I’ve transformed my entire relationship with fitness and exercise to be oriented more toward enjoyment.

Naturally, this more intrinsic approach ultimately improves the quality of our efforts, which generally leads to greater extrinsic rewards as well. Intrinsic motivation and extrinsic motivation are best when they are both pointed in the same direction.

Real work always seems to involve a certain amount of unpleasant, grinding effort though, and startups often have a lot of it. It’s like having a baby. It’s 5% cute, adorable moments, and 95% dirty diapers and vomit.

The key to loving these more unpleasant moments is meaning. If we genuinely care about and believe in our mission, then those difficult times begin to take on a more heroic quality.

Although it’s critical for a startup to have very immediate and actionable plans, such as write code and talk to users, I believe it’s also important to maintain a meaningful and inspiring vision.

The sixth, and final pattern for today is one that I borrowed from Google: Maintain a healthy disregard for the impossible.

I think Larry Page learned this as a kid at summer camp, and to me it represents the true innovative spirit of the company. Now that Google is huge and many have grown cynical about the company, it’s easy to dismiss such things. But I remember when it was a tiny startup that nobody had heard of, and I had to explain to people that it was like Yahoo! minus all of the features other than search. People would just give me this sad look that seemed to say, “I’m sorry you can’t get a real job.”

But inside the company there were these absurdly ambitious ideas that made it feel like we were going to take over the world. It was an exciting place to be.

Larry wanted to store and search the whole web in memory, even though our machines only had 1/4 GB of RAM. It was unrealistic at the time, but Moore’s law moves fast and very soon we were doing it, but only because everyone’s thinking was already oriented in that direction.

He also wanted self-driving cars that would deliver hamburgers. That hasn’t happened yet, but I bet it will.

For me, potentially impossible goals are much more inspiring than realistic ones. I’d rather fail at something awesome, than succeed at something inconsequential.

As with many of the other patterns, this one is about continually shedding the limitations of outdated thinking.

When I decided to write the Gmail interface in JavaScript, pretty much everyone who knew anything about JavaScript or web browsers told me that it was a bad idea. It had been tried in the past, and always ended in disaster. But times change fast, and fortunately I was in an environment where doing impossible things was not just permitted, but encouraged. After we launched, the impossible quickly became the new normal, completely changing how we think about web apps. That's fun.

For me, startups are more than just a clever way to make money. They are machines for harnessing the fire of human self-interest, creating a self-sustaining reaction capable of rapidly transforming the world. Self-interest is often treated as if it were dirty or wrong, but NASA didn't get to the moon by vilifying gravity.

It's often assumed that business is all about money, but to me that's like saying that rockets are all about rocket fuel. On some level it's true. You won't even make it off the launch pad without fuel. But that myopic view misses out on the larger purpose and mission of the machine. Certainly some businesses really are about nothing more than making money, but among the truly significant founders I've known, there's always a larger purpose. It's not just a nihilistic pursuit of rocket fuel.

Before I finish, I want to mention my impossible goal.

We now, for the first time ever, have the technology and resources necessary to make the world a great place for everyone. We can provide adequate food, housing, education, and healthcare for everyone, using only a fraction of our labor and resources. This means that we can put an end to wage-slavery. I don't have to work. I choose to work. And I believe that everyone deserves the same freedom I have. If done right, it's also economically superior, meaning that we will all have more wealth.

We often talk about how brilliant or visionary Steve Jobs was, but there are probably millions of people just as brilliant as he was. The difference is that they likely didn't grow up with great parents, amazing teachers, and an environment where innovation was the norm. Also they didn't live down the street from Steve Wozniak.

Economically, we don't need more jobs. We need more Steve Jobs. When we set everyone free, we enable the outliers everywhere. The result will be an unprecedented boom in human creativity and ingenuity.

And now the impossible part. First we have to learn how to get along with each other, and with ourselves.

I'm looking for full-stack hackers. People who understand that technology is more than just patterns in silicon. The same patterns and systems of patterns exist everywhere. Capitalism is a technology. Like the internal combustion engine, it's tremendously valuable and transformative, but it's not beyond improvement. The same goes for government, religion, and everything else. We have an incredible future ahead of us, but we won't get there by clinging to obsolete patterns.

As founders, we must start small, and work with the grain of what is. The path is never obvious, and innovation happens in the most unexpected ways. The personal computer was originally dismissed as a toy. If you think Instagram is just a collection of photo filters, you're missing the big picture. Maybe photo sharing won't lead directly to world peace, but helping people to see the world through the eyes of others looks like a step in the right direction to me. And they grew to over 200 million users in less than four years. That's larger than most countries. That's the power of a startup.

As Richard Feynman said, "The worthwhile problems are the ones you can really solve or help solve, the ones you can really contribute something to." Don't be discouraged by people who dismiss your efforts as trivial just because you aren't curing cancer or traveling to Mars. The patterns I've presented today are about developing an independent mind, unburdened by the limitations of other people's thinking. Then you can judge for yourself what is worthwhile, and move forward with the conviction necessary to do something great. A journey of a thousand miles begins with a single step.

Thank you. ■

Paul Buchheit is a partner at the venture capital firm Y Combinator. He previously co-founded FriendFeed, which was acquired by Facebook in 2009, and was one of the first engineers at Google. At Google, he started Gmail, suggested the "Don't be evil" motto, and created the first AdSense prototype.

Reprinted with permission of the original author.
First appeared in *hn.my/tech* (paulbuchheit.blogspot.com)

BETTER SENDING, BETTER INSIGHTS

ANALYZE, REACT, ENGAGE

NEW

- REST API
- Parse API
- Real-Time event API
- features coming soon!

mailjet™