HACKERMONTHLY

Issue 23 April 2012

HARVEST

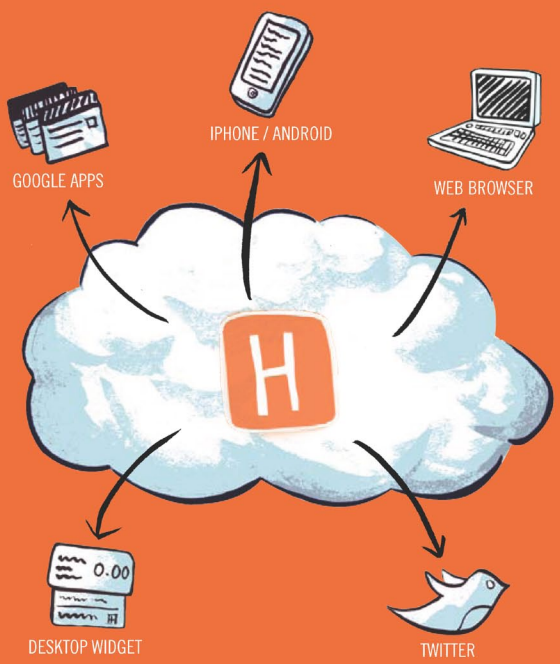
TIMESHEETS INVOICES REPORTS

Track time anywhere, and invoice your clients with ease.

Harvest is available wherever your work takes you. Whether you are working from home, on-site, or through a flight. Harvest keeps a handle on your billable time so you can invoice accurately. Visit us and learn more about how Harvest can help you work better today.

Why Harvest?

- Convenient and accessible, anywhere you go.
- Get paid twice as fast when you send web invoices.
- Trusted by small businesses in over 100 countries.
- Ability to tailor to your needs with full API.
- Fast and friendly customer support.



Learn more at **www.getHarvest.com/hackers**

Curator
Lim Cheng Soon

Contributors
Alex Walker
Tom Ryder
Nathan Marz
Nick Johnson
David Nolen
Yann Esposito
James Yu
Tom Blomfield
Des Traynor
Valdis Krebs

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrators
Teagan White
John Schwegel

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 Illustration: Teagan White (teaganwhite.com)

Contents

FEATURES

PROGRAMMING

SPECIAL

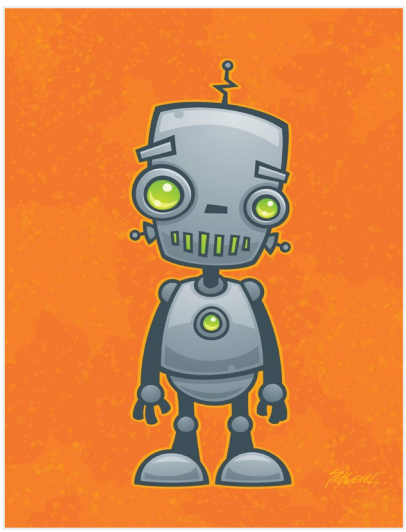


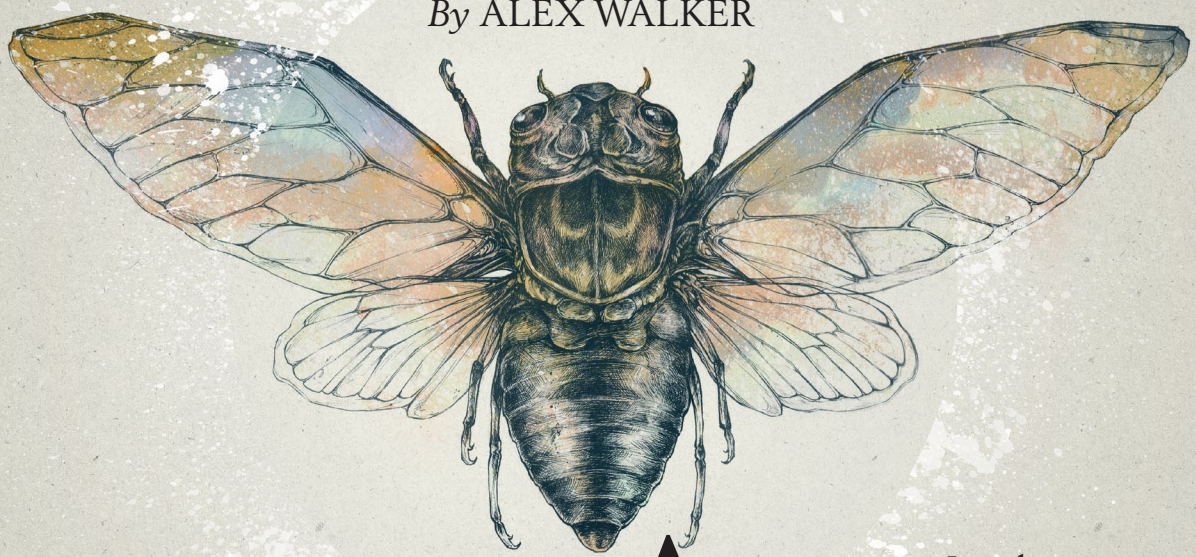
Illustration by John Schwegel (johnschwegel.com)

For links to Hacker News dicussions, visit *hackermonthly.com/issue-23*

The Cicada Principle

The Art of Seamless Tiles

By ALEX WALKER



A FEW YEARS AGO, I read some interesting stuff on periodical cicadas. We generally don't see a lot of these little guys, as they spend the vast majority of their lives quietly tunneling away underground and munching on tree roots.

However, depending on the species, every 7, 13, or 17 years these periodical cicadas simultaneously emerge en masse, transform into noisy, flying creatures, find a mate, and die not long after.

Illustration by Teagan White (teaganwhite.com)

While this is a rather rock & roll ending for our nerdy cicada, it raises an obvious question: is it just by chance that they adopted 7, 11, or 13-year life cycles, or are those numbers somehow special?

As it turns out, these numbers have something in common. They're all prime numbers — numbers that can only be divided by themselves and 1 (that is, 2, 3, 5, 7, 11, 13, 17, 19, 23, and so on).

But why does that matter?

Research has shown that the populations of creatures that eat cicadas — typically birds, spiders, wasps, fish and snakes — often have shorter 2–6 year cycles of boom and bust.

So, if our cicadas were to emerge, say, every 12 years, any predator that works in either 2, 3, 4 or 6 year cycles would be able to synchronize their boom years with this regular cicada feast. In fact, they'd probably name a public holiday after it called Cicada Day.

That's not much fun if you're a cicada.

On the other hand, if a brood of 17-year cicadas was unlucky enough to emerge during a bumper 3-year wasp season, it will be 51 years before that event occurs again. In the intervening years, our cicadas can happily emerge in their tens of thousands, completely overwhelm the local predator population, and be mostly left in peace.

Resourceful little guys, eh?

That's great. But what has all this got to do with web design?

A few weeks ago we looked at the process of making seamless tiles [hn.my/tiles]. As super-useful as seamless tiles are, it can be tough to get the balance just right.

On one hand, you want to keep the file dimensions as small as possible to best take advantage of the tiling effect. However, when you notice a distinctive feature — for instance, a knot in some woodgrain — repeating at regular intervals, it really breaks the illusion of organic randomness.

Maybe we borrow some ideas from cicadas to break that pattern?

Generating Organic Randomness with CSS

Example 1:

Enough talk. Here's a quick proof-of-concept. This is not supposed to be visually splendid, but it does a good job of showing what's going on. Keeping the "cicada principle" in mind, I've made three square, semi-transparent PNGs of 29px, 37px, and 53px respectively, and set them up as multiple backgrounds on the HTML element of a test page.

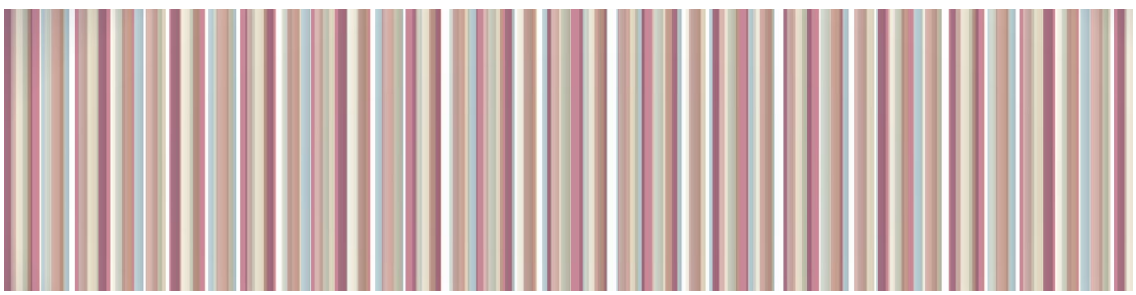
- `29-a.png (2.0kb)`

- `37-a.png (1.7kb)`

- `53-a.png (2.5kb)`

```
html {
  background-image:
    url(29-a.png),url(37-a.png),
    url(53-a.png);
  padding:0;
  margin:0;
  height: 100%;
}
```

And here's the result.



As you can see, the tiles overlap and interact to generate new patterns and colors. And as we're using magical prime numbers, this pattern will not repeat for a long, long time.

Exactly how long? 29px × 37px × 53px… or 56,869px!

Now this was something of a revelation to me. I actually had to triple-check my calculations, but the math is rock solid. Remember these are tiny graphics — less than 7kb in total — yet they are generating an area of original texture of almost 57,000 pixels wide.

You can imagine what happens if you were to add in a fourth layer of tiling — let's say a 43px tile. Or maybe you can't imagine it, as the numbers start getting a little brutal and are liable to slap you about the ears if you stare at them too long.

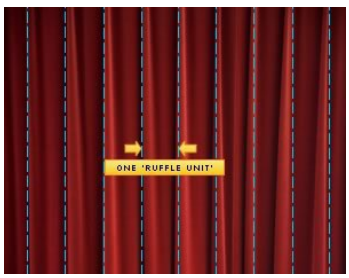
Suffice it to say, you'll get a number more relevant to planet terraformation than web design.

Ok. So, abstract, geometric stripes are nice and all, but how else can you apply this idea?

Example 2

Let's take a more photo-realistic example that we've probably all seen at some point: the red velvet theatre curtain. I found a nice curtain graphic here to use as a start point. Looking at our curtain you can see it breaks into roughly equal vertical units.

For this example I'm going to refer to this distance as one "ruffle unit," and (unlike the first example) it's going to be more important than the strict pixel dimensions of the images we're working with.



Coining a new measurement system — the "ruffle unit"

First, I'm going to pick out one of these ruffles and convert it into a seamless tile. It's a JPEG and it weighs in at a tidy 8kb.

One layer tiling curtain: Not exactly impressive

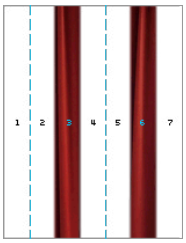
Rendered alone, this graphic is everything we don't like about tiling backgrounds. While there are no obvious visible join, it's very mechanical and wholly unconvincing.

For layer two, the prime number we're going to use is three. I'm going to pick out a new section of curtain and place it inside a transparent PNG that is three ruffle units wide. I've feathered the right and left edges so it blends smoothly with the background. The resulting file comes in a tick under 15kb.

A 3-unit-wide tile

Two layers of tiling curtain — an improvement

When we overlay this tile on our bottom layer we certainly get an improved result. There's still an unnaturally regular pattern apparent, but it's starting to break down a little.



Our third layer is a 7-unit-wide tile

The magic number for our third layer is seven.

We're creating a new transparent PNG seven ruffle units wide, and I'm going to drop in two new sections of ruffle image at positions 3 and 6. If that sounds confusing, the diagram next *should* clear things up a bit. Again, I've feathered the edges on the image to help it blend with the lower layers.

Obviously this is a larger image in both pixel dimension and file size, but it still only tips the scale at around 32kb — not outrageous by any measure.

The final result

Above's what happens when we tile this graphic over the first two layers. I'm pretty happy with that result. True, your eye can pick out small sections of image that seem to repeat (because they do), but the underlying pattern becomes so complicated that your eyes stops searching for the similarities.

To look at it another way, if we treat each ruffle purely as a number, the number pattern it produces looks like this: 1, 2, 3, 1, 2, 6, 1, 2, 1, 3, 2, 1, 6, 2, 1, 1, 3, 1, 1, 6, 1, 1, 2, 3,..

There is a pattern there but it's very difficult to discern.

In this example, a practically endless curtain background has cost us a grand total of just 53kb. And of course, it would be relatively trivial to add a fourth layer — perhaps using 11 units — if we wanted to. However, I'm not convinced that's warranted here.

Also bear this in mind: this example uses the one of the simplest possible sets of prime number — 1, 3, and 7. If we were to use, let's say, 11, 13, and 17, we could build in much more complex variation for a given distance. It really just comes down to the scale of the curtain we choose versus the screen width.

Example 3

My last example is less about pure practical applications, and more about having some fun with primes. I'm not going to break down the theory again, as the core concept is the same as the first two examples, but you're more than welcome to deconstruct it in FireBug.

> *2,200 years ago Emperor Qin Shi Huang, constructed an 8,000 man terracotta army to guard his tomb. Each soldier, chariot and weapon is a one-off, hand-crafted creation.*

Using simple CSS, prime numbers, and handful of images, we're going to raise our own mighty army. What it might lack in stature, it makes up for in sheer weight of numbers.

I give you… my Mighty Legion of Lego!

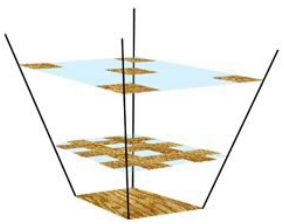
The legion is built from just eight images that mingle and weave together to produce thousands of permutations. It uses:

- 2 images for the background tiles
- 2 images for the legs
- 2 images for torsos
- 2 images for the heads

The Mighty Legion of Lego

Summary

Playing around with this idea, I've come up with some basic principles that seem to work. First, your stacking order tends to work best when it's constructed like an upside-down pyramid.



The stacking order model

You can afford to make the bottom layer quite small and repetitive as much of it gets overwritten by the layers above. In fact, it's likely that only 20–40% will remain unobscured.

On the other hand, your upper-most layer should always have the largest image dimensions but also the most thinly-scattered imagery, as these image elements will never be blocked out by other layers. It's also probably best not to include highly-distinctive, eye-catching detail on your uppermost layer. Keep it scarce and generic.

Either way, some trial and error is almost always required.

Browser Support

I've kept the markup simple by applying multiple backgrounds to the HTML element. This is supported by all the current main browsers (Firefox 4, Chrome 10, IE9, Opera 11, Safari 5) but obviously not all older versions.

However if backward compatibility is a prerequisite, tiling the `html`, `body` and perhaps a single container `div` element might be a viable option. While the container element might be non-semantic, it's potentially giving you huge sitewide value for a small concession. That's your call.

These three examples are the first ideas that came to my mind, but I'm sure there are some much cleverer takes on the idea. Perhaps:

- An endless cityscape
- Nonrepeating woodgrain
- Star fields
- Densely layered jungle
- Cloudscapes

Lastly, check out the Cicada Project [designfestival.com/cicada] to see where our community has taken this idea! ■

Alex has been a front-end developer since the table-olithic era. He enjoys doing strange things to CSS and then writing about it.

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

Google tracks you. We don't.

Vim Anti-Patterns

By TOM RYDER

The benefits of getting to grips with Vim are immense in terms of editing speed and maintaining your "flow" when you're on a roll, whether writing code, poetry, or prose, but because the learning curve is so steep for a text editor, it's very easy to retain habits from your time learning the editor that stick with you well into mastery. Because Vim makes you so fast and fluent, it's especially hard to root these out because you might not even notice them, but it's worth it. Here I'll list some of the more common ones.

Moving One Line at a Time

If you have to move more than a couple of lines, moving one line at a time by holding down `j` or `k` is inefficient. There are many more ways to move vertically in Vim. I find that the two most useful are moving by paragraph and by screenful, but this depends on how far and how precisely you have to move.

- `{` — Move to start of previous paragraph or code block.
- `}` — Move to end of next paragraph or code block.
- `Ctrl+F` — Move forward one screenful.
- `Ctrl+B` — Move backward one screenful.

If you happen to know precisely where you want to go, navigating by searching is the way to go, searching forward with `/` and backward with `?`.

It's always useful to jump back to where you were, as well, which is easily enough done with `` `` `` (two backticks), or `gi` to go to the last place you inserted text. If you like, you can even go back and forth through your entire change list of positions with `g;` and `g,`.

Moving One Character at a Time

Similarly, moving one character at a time with `h` and `l` is often a waste when you have `t` and `f`:

- `t<char>` — Move forward until the next occurrence of the character.
- `f<char>` — Move forward over the next occurrence of the character.
- `T<char>` — Move backward until the previous occurrence of the character.
- `F<char>` — Move backward over the previous occurrence of the character.

Moving wordwise with `w`, `W`, `b`, `B`, `e`, and `E` is better, too. Again, searching to navigate is good here, and don't forget you can yank, delete or change forward or backward to a search result:

```
y/search<Enter>
y?search<Enter>
d/search<Enter>
d?search<Enter>
c/search<Enter>
c?search<Enter>
```

Searching for the Word Under the Cursor

Don't bother typing it, or yanking/pasting it; just use `*` or `#`. It's dizzying how much faster this feels when you use it enough for it to become automatic.

Deleting, Then Inserting

Deleting text with intent to replace it by entering insert mode immediately afterward isn't necessary:

```
d2wi
```

It's quicker and tidier to use `c` for change:

```
c2w
```

This has the added benefit of making the entire operation repeatable with the `.` command.

Using the Arrow Keys

Vim lets you use the arrow keys to move around in both insert and normal mode, but once you're used to using `hjkl` to navigate, moving to the arrow keys to move around in text feels clumsy; you should be able to spend the vast majority of a Vim session with your hands firmly centered around home row. Similarly, while the Home and End keys work the same way they do in most editors, there's no particular reason to use them when functional equivalents are closer to home in `^` and `$`.

So wean yourself off the arrow keys, by the simple expedient of disabling them entirely, at least temporarily:

```
noremap <Up> <nop>
noremap <Down> <nop>
noremap <Left> <nop>
noremap <Right> <nop>
```

The benefits of sticking to home row aren't simply in speed; it feels nicer to be able to rest your wrists in front of the keyboard and not have to move them too far, and for some people it has even helped prevent repetitive strain injury.

Moving in Insert Mode

There's an additional benefit to the above in that it will ease you into thinking less about insert mode as a mode in which you move around; that's what normal mode is for. You should, in general, spend as little time in insert mode as possible. When you want to move, you'll get in the habit of leaving insert mode, and moving around far more efficiently in normal mode instead. This distinction also helps to keep your insert operations more atomic, and hence more useful to repeat.

Moving to Escape

The Escape key on modern keyboards is a lot further from home row than it was on Bill Joy's keyboard back when he designed vi. Hitting Escape is usually unnecessary; `Ctrl+[` is a lot closer, and more comfortable. It doesn't take long using this combination instead to make reaching for Escape as you did when you were a newbie feel very awkward. You might also consider mapping the otherwise pretty useless Caps Lock key to be another Escape key in your operating system, or even mapping uncommon key combinations like `jj` to Escape. I feel this is a bit drastic, but it works well for a lot of people:

```
inoremap jj <Esc>
```

Moving to the Start or End of the Line, Then Inserting

Just use `I` and `A`. Again, these make the action repeatable for other lines which might need the same operation.

Entering Insert Mode, Then Opening a New Line

Just use `o` and `O` to open a new line below and above respectively, and enter insert mode on it at the same time.

Entering Insert Mode to Delete Text

This is a pretty obvious contradiction. Instead, delete the text by moving to it and using `d` with an appropriate motion or text object. Again, this is repeatable, and means you're not holding down Backspace. In general, if you're holding down a key in Vim, there's probably a faster way.

Repeating Commands or Searches

Just type `@:` for commands or `n/N` for searches; Vim doesn't forget what your last search was as soon as you stop flicking through results. If it wasn't your most recent command or search but it's definitely in your history, just type `q:` or `q/`, find it in the list, and hit Enter.

Repeating Substitutions

Just type `&`.

Repeating Macro Calls

Just type `@@`.

These are really only just a few of the common traps to avoid to increase your speed and general efficiency with the editor without requiring plugins or substantial remappings. ■

Tom Ryder is a Linux systems administrator and web developer from New Zealand. He's an enthusiastic fan of the Vim text editor, the Bash shell, and free software development tools. He blogs regularly at "Arabesque" [blog.sanctum.geek.nz].

Reprinted with permission of the original author. First appeared in *hn.my/vap* (sanctum.geek.nz)

Suffering-Oriented Programming

By NATHAN MARZ

Someone asked me an interesting question the other day: "How did you justify taking such a huge risk on building Storm [hn.my/storm] while working on a startup?" (Storm is a real-time computation system). I can see how from an outsider's perspective investing in such a massive project seems extremely risky for a startup. From my perspective, though, building Storm wasn't risky at all. It was challenging, but not risky.

I follow a style of development that greatly reduces the risk of big projects like Storm. I call this style "suffering-oriented programming." Suffering-oriented programming can be summarized like so: don't build technology unless you feel the pain of not having it. It applies to the big, architectural decisions as well as the smaller everyday programming decisions. Suffering-oriented programming greatly reduces risk by ensuring that you're always working on something important, and it ensures that you are well-versed in a problem space before attempting a large investment.

I have a mantra for suffering-oriented programming: "First make it possible. Then make it beautiful. Then make it fast."

First Make It Possible

When encountering a problem domain with which you're unfamiliar, it's a mistake to try to build a "general" or "extensible" solution right off the bat. You just don't understand the problem domain well enough to anticipate what your needs will be in the future. You'll make things generic that needn't be, adding complexity and wasting time.

It's better to just "hack things out" and be very direct about solving the problems you have at hand. This allows you to get done what you need to get done and avoid wasted work. As you're hacking things out, you'll learn more and more about the intricacies of the problem space.

The "make it possible" phase for Storm was one year of hacking out a stream processing system using queues and workers. We learned about guaranteeing data processing using an "ack" protocol. We learned to scale our real-time computations with clusters of queues and workers. We learned that sometimes you need to partition a message stream in different ways, sometimes randomly and sometimes using a hash/mod technique that makes sure the same entity always goes to the same worker.

We didn't even know we were in the "make it possible" phase. We were just focused on building our products. The pain of the queues and workers system became acute very quickly though. Scaling the queues and workers system was tedious, and the fault-tolerance was nowhere near what we wanted. It was evident that the queues and workers paradigm was not at the right level of abstraction, as most of our code had to do with routing messages and serialization and not the actual business logic we cared about.

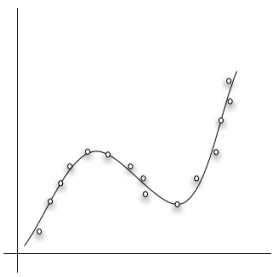
At the same time, developing our product drove us to discover new use cases in the "real-time computation" problem space. We built a feature for our product that would compute the reach of a URL on Twitter. Reach is the number of unique people exposed to a URL on Twitter. It's a difficult computation that can require hundreds of database calls and tens of millions of impressions to distinct just for one computation. Our original implementation that ran on a single machine would take over a minute for hard URLs, and it was clear that we needed a distributed system of some sort to parallelize the computation to make it fast.

One of the key realizations that sparked Storm was that the "reach problem" and the "stream processing" problem could be unified by a simple abstraction.

Then Make It Beautiful

You develop a "map" of the problem space as you explore it by hacking things out. Over time, you acquire more and more use cases within the problem domain and develop a deep understanding of the intricacies of building these systems. This deep understanding can guide the creation of "beautiful" technology to replace your existing systems, alleviate your suffering, and enable new systems/features that were too hard to build before.

The key to developing the "beautiful" solution is figuring out the simplest set of abstractions that solve the concrete use cases you already have. It's a mistake to try to anticipate use cases you don't actually have or else you'll end up over-engineering your solution. As a rule of thumb, the bigger the investment you're trying to make, the deeper you need to understand the problem domain and the more diverse your use cases need to be. Otherwise you risk the second-system effect [hn.my/sse].



"Making it beautiful" is where you use your design and abstraction skills to distill the problem space into simple abstractions that can be composed together. I view the development of beautiful abstractions as similar to statistical regression: you have a set of points on a graph (your use cases) and you're looking for the simplest curve that fits those points (a set of abstractions).

The more use cases you have, the better you'll be able to find the right curve to fit those points. If you don't have enough points, you're likely to either overfit or underfit the graph, leading to wasted work and over-engineering.

A big part of making it beautiful is understanding the performance and resource characteristics of the problem space. This is one of the intricacies you learn in the "making it possible" phase, and you should take advantage of that learning when designing your beautiful solution.

With Storm, I distilled the real-time computation problem domain into a small set of abstractions: streams, spouts, bolts, and topologies. I devised a new algorithm for guaranteeing data processing that eliminated the need for intermediate message brokers, the part of our system that caused the most complexity and suffering. That both stream processing and reach, two very different problems on the surface, mapped so elegantly to Storm was a strong indicator that I was onto something big.

I took additional steps to acquire more use cases for Storm and validate my designs. I canvassed other engineers to learn about the particulars of the real-time problems they were dealing with. I didn't just ask people I knew. I also tweeted out that I was working on a new real-time system and wanted to learn about other people's use cases. This led to a lot of interesting discussions that educated me more on the problem domain and validated my design ideas.

Then Make It Fast

Once you've built out your beautiful design, you can safely invest time in profiling and optimization. Doing optimization too early will just waste time, because you still might rethink the design. This is called premature optimization.

"Making it fast" isn't about the high level performance characteristics of a system. The understanding of those issues should have been acquired in the "make it possible" phase and designed for in the "make it beautiful" phase. "Making it fast" is about micro-optimizations and tightening up the code to be more resource efficient. So you might worry about things like asymptotic complexity in the "make it beautiful" phase and focus on the constant-time factors in the "make it fast" phase.

Rinse and Repeat

Suffering-oriented programming is a continuous process. The beautiful systems you build give you new capabilities, which allow you to "make it possible" in new and deeper areas of the problem space. This feeds learning back to the technology. You often have to tweak or add to the abstractions you've already come up with to handle more and more use cases.

Storm has gone through many iterations like this. When we first started using Storm, we discovered that we needed the capability to emit multiple, independent streams from a single component. We discovered that the addition of a special kind of stream called the "direct stream" would allow Storm to process batches of tuples as a concrete unit. Recently I developed "transactional topologies" which go beyond Storm's at-least-once processing guarantee and allow exactly-once messaging semantics to be achieved for nearly arbitrary real-time computation.

By its nature, hacking things out in a problem domain you don't understand so well and constantly iterating can lead to some sloppy code. The most important characteristic of a suffering-oriented programmer is a relentless focus on refactoring. This is critical to prevent accidental complexity from sabotaging the codebase.

Conclusion

Use cases are everything in suffering-oriented programming. They're worth their weight in gold. The only way to acquire use cases is through gaining experience through hacking.

There's a certain evolution most programmers go through. You start off struggling to get things to work and have absolutely no structure to your code. Code is sloppy and copy/pasting is prevalent. Eventually you learn about the benefits of structured programming and sharing logic as much as possible.

Then you learn about making generic abstractions and using encapsulation to make it easier to reason about systems. Then you become obsessed with making all your code generic, with making things extensible to future-proof your programs.

Suffering-oriented programming rejects that you can effectively anticipate needs you don't currently have. It recognizes that attempts to make things generic without a deep understanding of the problem domain will lead to complexity and waste. Designs must always be driven by real, tangible use cases. ■

Nathan Marz is an engineer at Twitter. Previously Nathan was the lead engineer of BackType which was acquired by Twitter in July of 2011. He is a believer in the power of open source and has authored some significant open source projects, including Cascalog, ElephantDB, and Storm. He writes a blog at *nathanmarz.com*

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

Spatial Indexing with Quadtrees & Hilbert Curves

By NICK JOHNSON

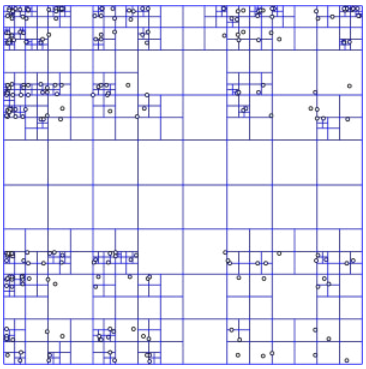
Last Thursday night after the sessions at Oredev was "Birds of a Feather" — a sort of mini-unconference. Anyone could write up a topic on the whiteboard; interested individuals added their names, and each group got allocated a room to chat about the topic. I joined the "Spatial Indexing" group, and we spent a fascinating hour-and-a-half talking about spatial indexing methods, which reminded me of several interesting algorithms and techniques.

Spatial indexing is increasingly important as more and more data and applications are geospatially-enabled. Efficiently querying geospatial data, however, is a considerable challenge. Because the data is two-dimensional (or sometimes more), you can't use standard indexing techniques to query on position. Spatial indexes solve this through a variety of techniques. In this post, we'll cover several methods: quadtrees, geohashes (not to be confused with geohashing), and space-filling curves and reveal how they're all interrelated.

Quadtrees

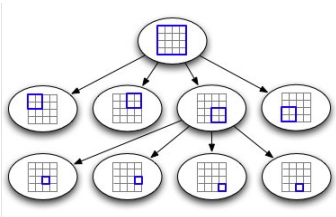
Quadtrees are a very straightforward spatial indexing technique. In a Quadtree, each node represents a bounding box covering some part of the space being indexed, with the root node covering the entire area. Each node is either a leaf node or an internal node. A leaf node contains one or more indexed points and no children while an internal node has exactly four children, one for each quadrant obtained by dividing the area covered

in half along both axes, hence the name.



Inserting data into a Quadtree is simple:

- Start at the root and determine which quadrant your point occupies.
- Recurse to that node and repeat until you find a leaf node.
- Add your point to that node's list of points.
- If the list exceeds some pre-determined maximum number of elements, split the node and move the points into the correct subnodes.



A representation of how a Quadtree is structured internally.

To query a Quadtree:

- Start at the root.
- Examine each child node and check if it intersects the area being queried for. If it does, recurse into that child node. Whenever you encounter a leaf node, examine each entry to see if it intersects with the query area and return it if it does.

Note that a Quadtree is very regular. It is, in fact, a trie since the values of the tree nodes do not depend on the data being inserted. A consequence of this is that we can uniquely number our nodes in a straightforward manner:

- Number each quadrant in binary (00 for the top left, 10 for the top right, and so forth)
- The number for a node is the concatenation of the quadrant numbers for each of its ancestors, starting at the root. Using this system, the bottom right node in the sample image would be numbered 11 01.

If we define a maximum depth for our tree, then we can easily calculate a point's node number without reference to the tree:

- Normalize the node's coordinates to an appropriate integer range (for example, 32 bits each).
- Interleave the bits from the x and y coordinates. Each pair of bits specifies a quadrant in the hypothetical Quadtree.

Geohashes

This system might seem familiar: it's a geohash! At this point, you can actually throw out the Quadtree itself. The node number, or geohash, contains all the information we need about its location in the tree. Each leaf node in a full-height tree is a complete geohash, and each internal node is represented by the range from its smallest leaf node to its largest one. Thus, you can efficiently locate all the points under any internal node by performing a query to index on the geohash for everything within the numeric range covered by the desired node.

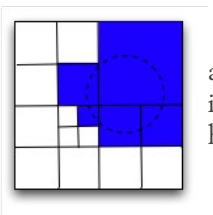
Querying once we've thrown away the tree itself becomes a little more complex. Instead of refining our search set recursively, we need to construct a search set ahead of time by finding the smallest prefix (or quadtree node) that completely covers the query area. In the worst case, this may be substantially larger than the actual query area. For example, a small shape in the center of the indexed area that intersects all four quadrants would require selecting the root node for this step.

The aim, now, is to construct a set of prefixes that completely covers the query region while including as little area outside the region as possible. If we had no other constraints, we could simply select the set of leaf nodes that intersect the query area, but that would result in a lot of queries.

Another constraint, then, is that we want to minimize the number of distinct ranges we have to query for. One approach to doing this is setting a maximum number of ranges we're willing to have:

- Construct a set of ranges, initially populated with the prefix we identified earlier.
- Pick the node in the set that can be subdivided without exceeding the maximum range count and that will remove the most unwanted area from the query region.
- Repeat this until there are no ranges in the set that can be further subdivided.
- Examine the resulting set, and join any adjacent ranges, if possible.

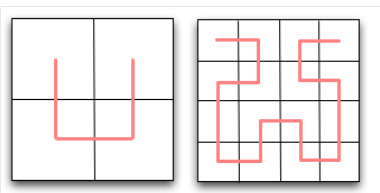
The diagram below demonstrates how this works for a query on a circular area with a limit of 5 query ranges.



How a query for a region is broken into a series of geohash prefixes/ranges.

This approach works well, and it allows us to avoid recursive lookups. The set of range lookups we do execute can all be done in parallel. Since each lookup can be expected to require a disk seek, parallelizing our queries allows us to substantially cut down the time required to return the results.

Still, we can do better. You may notice that all the areas we need to query in the above diagram are adjacent, yet we can only merge two of them (the 2 in the bottom right of the selected area) into a single range query, requiring us to do 4 separate queries. This is due in part to the order that our geohashing approach "visits" subregions, working left to right, then top to bottom in each quad. The discontinuity as we go from top right to bottom left results in us having to split up some ranges that we could otherwise make contiguous. If we were to visit regions in a different order, perhaps we could minimize or eliminate these discontinuities, resulting in more areas that can be treated as adjacent and fetched with a single query. With an improvement in efficiency like that, we could do fewer queries for the same area covered, or conversely, the same number of queries, but including less extraneous area.



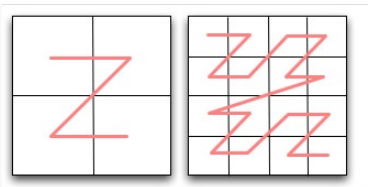
Illustrates the order in which a Hilbert Curve "visits" each quad.

Hilbert Curves

Suppose instead, we visit regions in a "U" shape. Within each quad, of course, we also visit subquads in the same "U" shape, but aligned so as to match up with neighboring quads. If we organize the orientation of these "U"s correctly, we can completely eliminate any discontinuities and visit the entire area at whatever resolution we choose continuously to fully explore each region before moving on to the next. Not only does this eliminate discontinuities, but it also improves the overall locality. The pattern we get if we do this may look familiar because it's a Hilbert Curve.

Hilbert Curves are part of a class of one-dimensional fractals known as space-filling curves, so named because they are one-dimensional lines that nevertheless fill all available space in a fixed area. They're fairly well known, in part thanks to XKCD's use of them for a map of the internet [xkcd.com/195/]. As you can see, they're also of use for spatial indexing since they exhibit exactly the locality and continuity required. For instance, if we take another look at the example we used for finding the set of queries required to encompass a circle above, we find that we can reduce the number of queries by one. The small region in the lower left is now contiguous with the region to its right and while the two regions at the bottom are no

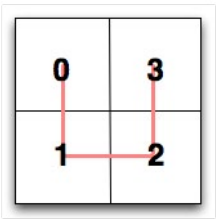
longer contiguous with each other, the rightmost one is now contiguous with the large area in the upper right.



Illustrates the order in which the geohashing approach "visits" each quad.

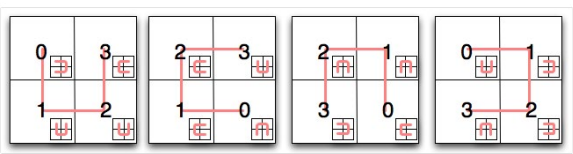
One thing that our elegant new system is lacking so far is a way to convert between a pair of (x,y) coordinates and the corresponding position in the Hilbert Curve. With geohashing it was easy and obvious — just interleave the x and y coordinates. However, there's no obvious way to modify that for a Hilbert Curve. Searching the internet, you're likely to come across many descriptions of how Hilbert Curves are drawn, but there are few, if any, descriptions of how to find the position of an arbitrary point. To figure this out, we need to take a closer look at how the Hilbert Curve can be recursively constructed.

The first thing to observe is that although most references to Hilbert Curves focus on how to draw the curve, this is a distraction from the essential property of the curve. Specifically, its importance to us: it's an ordering for points on a plane. If we express a Hilbert Curve in terms of this ordering, drawing the curve itself becomes trivial, as it is simply a matter of connecting the dots. Forget about how to connect adjacent sub-curves, and instead, focus on how to recursively enumerate the points.



Hilbert Curves are all about ordering a set of points on a 2d plane.

At the root level, enumerating the points is simple: pick a direction and a start point, and proceed around the four quadrants, numbering them 0 to 3. The difficulty is introduced when we want to determine the order in which we visit the sub-quadrants while maintaining the overall adjacency property. Examination reveals that each of the sub-quadrants' curves are a simple transformation of the original curve: there are only four possible transformations. Naturally, this applies recursively to sub-sub quadrants and so forth. The curve we use for a given quadrant is determined by the curve we used for the square it's in and the quadrant's position. With a little work, we can construct a table that encapsulates this:



Suppose we want to use this table to determine the position of a point on a third-level Hilbert Curve. For the sake of this example, assume our point has coordinates (5,2):

- Starting with the first square on the diagram, find the quadrant our point is in. In this case, it's the upper right quadrant. The first part of our Hilbert Curve position, then, is 3 (11 in binary).
- Consult the square shown in the inset of square 3. In this case, it's the second square.
- Repeat the process. Which sub-quadrant does our point fall into? Here, it's the lower left one, meaning the next part of your position is 1 and the square we should should consult next is the second one again.
- Repeat the process one final time to find that our point falls in the upper right sub-sub-quadrant. The final coordinate is 3 (11 in binary). Stringing them together, we now know the position of the point on the curve is 110111 binary, or 55.

Let's be a little more methodical and write methods to convert between (x,y) coordinates and Hilbert Curve positions. First, we need to express our diagram above in terms a computer can understand:

```
hilbert_map = {
'a': {(0, 0): (0, 'd'), (0, 1):
     (1, 'a'),
     (1, 0): (3, 'b'), (1, 1): (2,
     'a')},

'b': {(0, 0): (2, 'b'), (0, 1):
     (1, 'b'),
     (1, 0): (3, 'a'), (1, 1): (0,
     'c')},
'c': {(0, 0): (2, 'c'), (0, 1):
     (3, 'd'),
     (1, 0): (1, 'c'), (1, 1): (0,
     'b')},
'd': {(0, 0): (0, 'a'), (0, 1):
     (3, 'c'),
     (1, 0): (1, 'd'), (1, 1): (2,
     'd')},
     }
```

In the snippet above, each element of hilbert_map corresponds to one of the four squares in the diagram above. To make things easier to follow, I've identified each one with a letter: "a" is the first square, "b" the second, and so forth. The value for each square is a dict, mapping x and y coordinates for the (sub-)quadrant to the position along the line (the first part of the value tuple) and the square to use next (the second part of the value tuple). Here's how we can use this to translate x and y coordinates into a hilbert curve position:

```
def point_to_hilbert(x, y,
order=16):
  current_square = 'a'
  position = 0
  for i in range(order - 1, -1,
-1):
    position <<= 2 quad_x = 1
      if x & (1 << i) else 0
    quad_y = 1
      if y & (1 << i) else 0
    quad_position,
    current_square = hilbert_
map[current_square]
      [(quad_x, quad_y)]
    position |= quad_position
  return position
```

The input to this function is the integer x and y coordinates and the order of the curve. An order 1 curve fills a 2x2 grid, an order 2 curve fills a 4x4 grid, and so forth. Our x and y coordinates, then, should be normalized to a range of 0 to 2order-1. The function works by stepping over each bit of the x and y coordinates, starting with the most significant. For each, it determines which (sub-)quadrant the coordinate lies in by testing the corresponding bit and then fetching the position along the line and the next square to use from the table we defined earlier. The curve position is set as the least significant 2 bits on the position variable. At the beginning of the next loop, it's left-shifted to make room for the next set of coordinates.

Let's check that we've written the function correctly by running our example from above through it:

```
>>> point_to_hilbert(5,2,3)
55
```

Presto! For a further test, we can use the function to generate a complete list of ordered points for a Hilbert Curve and then use a spreadsheet to graph them and see if we get a Hilbert Curve. Enter the following expression into an interactive Python interpreter:

```
>>> points = [(x, y) for x in
    range(8) for y in range(8)]

>>> sorted_points = sorted(points,
    key=lambda k: point_to_
    hilbert(k[0], k[1], 3))

>>> print '\n'.join('%s,%s' % x
    for x in sorted_points)
```

Take the resulting text, paste it into a file called hilbert.csv, open it in your favorite spreadsheet, and instruct it to generate a scatter plot. The result is, of course, a nicely plotted Hilbert Curve!

The inverse of point_to_hilbert is a straightforward reversal of the hilbert_map; implementing it is left as an exercise for the reader.

Conclusion

There you have it — spatial indexing from quadtrees to geohashes to hilbert curves. One final observation: If you express the ordered sequence of (x,y) coordinates required to draw a Hilbert Curve in binary, do you notice anything interesting about the ordering? Does it remind you of anything?

Just to wrap up, a caveat: All of the indexing methods I've described today are only well-suited to indexing points. If you want to index lines, polylines, or polygons, you're probably out of luck with these methods. And so far as I'm aware, the only known algorithm for effectively indexing shapes is the R-tree, an entirely different and more complex beast. ■

Nick Johnson is a Developer Programs Engineer for Google App Engine, who's just seen the light and relocated to Australia. He regularly blogs about interesting computer science topics at his blog [blog.notdot.net], and when he's not saving the world there he can be found on Twitter (@nicksdjohnson) or Stack Overflow helping folks out.

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

Comparing JavaScript, CoffeeScript & ClojureScript

By DAVID NOLEN

I've been spending a lot of time recently hacking on the ClojureScript language [github.com/clojure/clojurescript]. I can say without qualification that I haven't had this much fun programming since I first taught myself JavaScript nearly seven years ago. So let's put aside logic programming for a moment and let's talk about code complexity and code expressivity.

Recently on StackOverflow someone asked how to idiomatically construct a type in ClojureScript. Before we get into that, let's consider how this is done in JavaScript:

```
// 193 characters
var Foo = function(a, b, c){
  this.a = a;
  this.b = b;
  this.c = c;
}
Foo.prototype.bar = function(x){
  return this.a + this.b + this.c
+ x;
}
var afoo = new Foo(1,2,3);
afoo.bar(3);
```

CoffeeScript gets a lot of deserved attention for its brevity for common tasks. For example the same thing in CoffeeScript:

```
# 106 characters

class Foo
  constructor: (@a, @b, @c) ->
  bar: (x) -> @a + @b + @c + x

afoo = new Foo 1, 2, 3
afoo.bar 3
```

That requires nearly half the amount of characters. Of course on real code the code compression isn't nearly that great — perhaps 10-20% in my experience. Still, I find that CoffeeScript tends to give the feeling of compression for many common tasks, and how a language feels day in and day out is important for programmer happiness.

Let's take a look at the same thing in ClojureScript:

```
;; 130 characters

(defprotocol IFoo
  (bar [this x])) ;; 93 characters
w/o this!

(deftype Foo [a b c]
  IFoo
  (bar [_ x] (+ a b c x)))

(def afoo (Foo. 1 2 3))
(bar afoo 3)
```

The ClojureScript without the strange protocol form would give even better compression than CoffeeScript! So what does this protocol form do, and why do we need that cluttering up our type definition?

ClojureScript, unlike JavaScript or CoffeeScript, promotes defining reusable abstractions. Imagine if all the types in your favorite library were swappable with your own implementations? Hmm…perhaps that's an abstraction too far for many users of JavaScript or CoffeeScript.

Well, here's a use case I think more people will get: neither JavaScript nor CoffeeScript provides any kind of `doesNotUnderstand:` hook that is fantastic for providing default implementations.

```
(defprotocol IFoo
  (bar [this x]))

(extend-type default
  IFoo
  (bar [_ x] :default))

(bar 1) ; >> :default
```

We've extended all objects including numbers to respond to the `bar` function. We can provide more specific implementations at anytime, i.e. by using `extend-type` on `string`, `array`, `Vector`, even your custom types instead of `default`. It's important to note that this extension is safe and local to whatever namespace you defined your protocol.

Still not convinced? Let's demonstrate a very powerful form of extension that even Dart is getting behind.

In ClojureScript it's simple to construct types which act like functions. While this might sound esoteric, consider very succinct operations like the following:

```
(def address {:street "1010 Foo
Ave."

     :apt "11111111"
     :city "Bit City"
     :zip "00000000"})

(map address [:street :zip])
;; >> ("1010 Foo Ave." "00000000")
```

Wow. HashMaps in ClojureScript are functions! Now this may look like some special case provided by the language, but that's not true. ClojureScript eats its own dog food; the language is defined on top of reusable abstractions.

How can we leverage this? An example: JavaScript and CoffeeScript both let you extract a range from strings and arrays. In JavaScript you have `slice` and CoffeeScript provides sugar via the `[i..j]` syntax. Neither provides you with a way to succinctly construct and manipulate the idea of a slice. For example:

```
(defprotocol ISlice
  (-shift [this]))

(deftype Slice [start end]
  ISlice
  (-shift [_] (Slice. (inc start)
(inc end)))
  IFn
  (-invoke [_ x]
    (cond
      (string? x) (.substring x
start end)
      (vector? x) (subvec x start
end))))

(def s (Slice. 0 5))
(def v ["List Processing" [0 1 2
3 4 5 6]])

(map s v)
;; >> ("List " [0 1 2 3 4])
(map (-shift s) v)
;; >> ("ist P" [1 2 3 4 5])
```

`IFn` is one of the many reusable abstractions that ships with language. We define `ISlice` to illustrate that our type has dual functionality as an object with fields that can be manipulated and as a function which can be applied to data!

Many people have the misconceived notion that Clojure/Script is only about functional programming. On the contrary Clojure/Script is very much "Object Oriented Programming: The Good Parts." ■

David Nolen is a JavaScript developer for The New York Times. In his free time he works on a variety of open source Clojure projects including core.match, core.logic, and ClojureScript.

Reprinted with permission of the original author. First appeared in *hn.my/jscs* (dosync.posterous.com)

Haskell Web Programming: A Yesod Tutorial

By YANN ESPOSITO

The Yesod documentation and particularly the book are excellent. But I missed an intermediate tutorial. This tutorial won't explain all details, but I will try to give a step by step of how to start from a five minute tutorial to an almost production-ready architecture. Furthermore, explaining something to others is a great way to learn. If you are used to Haskell [haskell.org] and Yesod [yesodweb.com], this tutorial won't teach you much. If you are completely new to Haskell and Yesod, it hopefully helps you.

During this tutorial you'll install, initialize, and configure your first Yesod project. Then there is a very minimal five-minute Yesod tutorial to heat up and verify the awesomeness of Yesod. Then we will clean up the five-minute tutorial to use some "best practices." Finally, there will be a more standard real-world example: a minimal blog system.

Before the Real Start

Install

The recommended way to install Haskell is to download the Haskell Platform [haskell.org/platform].

Once done, you need to install Yesod. Open a terminal session and do:

```
~ cabal update
~ cabal install Yesod cabal-dev
```

Initialize

You are now ready to initialize your first Yesod project. Open a terminal and type:

```
~ yesod init
```

Enter your name, choose yosog for the project name and enter Yosog for the name of the Foundation. Finally choose sqlite. Now, start the development cycle:

```
~ cd yosog
~ cabal-dev install && yesod --dev
devel
```

This will compile the entire project. Be patient: it could take a while the first time. Once finished, a server is launched, and you can visit it at *http://localhost:3000*

Some Last Minute Words

Up until here, we have a directory containing a bunch of files and a local web server listening the port 3000. If we modify a file inside this directory, Yesod should try to recompile the site as fast as possible. Instead of explaining the role of every file, let's focus only on the important files/directories for this tutorial:

- `config/routes` — is where you'll configure the map URL to code.
- `Handler/` — contains the files that will contain the code called when a URL is accessed.
- `templates/` — contains HTML, JavaScript and CSS templates.
- `config/models` — is where you'll configure the persistent objects (database tables).

Now we are ready to start!

Echo

To verify the quality of the security of the Yesod framework, let's make a minimal echo application.

> Goal: Make a server that when accessed `/echo/[some text]` should return a web page containing "some text" inside an `h1` bloc.

First, we must declare the URL of the form `/echo/...` meaningful. Let's take a look at the file `config/routes`:

```
/static StaticR Static getStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ RootR GET
```

We want to add a route of the form `/echo/[anything]` somehow and do some action with this. Add the following:

```
/echo/#String EchoR GET
```

This line contains three elements: the URL pattern, a handler name, an HTTPmethod. I am not particularly a fan of the big R notation, but this is the standard convention.

If you save `config/routes`, you should see your terminal in which you launched `yesod devel` activate and certainly displaying an error message.

```
Application.hs:31:1: Not in scope:
`getEchoR'
```

Why? Simply because we didn't write the code for the handler `EchoR`. Edit the file `Handler/Root.hs` and append this:

```
getEchoR :: String -> Handler
RepHtml
getEchoR theText = do
    defaultLayout $ do
        amlet|<h1>#{theText}|]
```

Don't worry if you find all of this a bit cryptic. In short, it just declares a function named `getEchoR` with one argument (`theText`) of type String. When this function is called, it returns a `Handler RepHtml` whatever it is. But mainly this will encapsulate our expected result inside an HTML text.

After saving the file, you should see Yesod recompile the application. When the compilation is finished you'll see the message: "Starting devel application."

Now you can visit: *http://localhost:3000/echo/Yesod%20rocks!*

TADA! It works!

Bulletproof?

Even this extremely minimal web application has some impressive properties. For example, imagine an attacker entering this URL:

http://localhost:3000/echo/<a>I'm <script>alert("Bad!");

The special characters are protected for us. A malicious user could not hide some bad script inside.

This behavior is a direct consequence of *type safety*. The URL string is put inside a URL type. Then the interesting part in the URL is put inside a String type. To pass from URL type to String type, some transformation are made. For example, replace all "%20" with space characters. Then to show the String inside an HTML document, the String is put inside an HTML type. Some transformation occurs like replace "<" by "<". Thanks to Yesod, this tedious job is done for us.

Yesod is not only fast, it helps us to remain secure. It protects us from many common errors in other paradigms. Yes, I am looking at you, PHP!

Cleaning Up

Even this very minimal example should be enhanced. We will clean up many details:

- Use a general CSS (cleaner than the empty by default)
- Dispatch handler code into different files
- Use `Data.Text` instead of `String`
- Put our "views" inside the template directory

Use a Better CSS

It is nice to note, the default template is based on HTML5 boilerplate. Let's change the default CSS. Add a file named `default-layout.lucius` inside the `templates/` directory containing:

```
body {
    font-family: Helvetica, sans-
serif;
    font-size: 18px; }
#main {
    padding: 1em;
    border: #CCC solid 2px;
    border-radius: 5px;
    margin: 1em;
    width: 37em;
    margin: 1em auto;
    background: #F2F2F2;
    line-height: 1.5em;
    color: #333; }
.required { margin: 1em 0; }
.optional { margin: 1em 0; }
label { width: 8em; display:
inline-block; }
input, textarea { background:
#FAFAFA}
textarea { width: 27em; height:
9em;}
ul { list-style: square; }
a { color: #A56; }
a:hover { color: #C58; }
a:active { color: #C58; }
a:visited { color: #943; }
```

Personally I would prefer if such a minimal CSS was put with the scaffolding tool. I am sure somebody already made such a minimal CSS which gives the impression the browser handles HTML correctly without any style applied to it. But I digress.

Separate Handlers

Generally you don't want to have all your code inside a unique file. This is why we will separate our handlers. First create a new file `Handler/Echo.hs` containing:

```
module Handler.Echo where

import Import

getEchoR :: String -> Handler
RepHtml
getEchoR theText = do
    defaultLayout $ do
        [whamlet|<h1>#{theText}|]
```

Do not forget to remove the getEchoR function inside `Handler/Root.hs`.

We must declare this new file into `yosog.cabal`. Just after `Handler.Root`, add:

```
Handler.Echo
```

We must also declare this new Handler module inside `Application.hs`. Just after the "`import Handler.Root`", add:

```
import Handler.Echo
```

Data.Text

It is good practice to use `Data.Text` instead of `String`.
To declare it, add this import directive to `Foundation.hs` (just after the last one):

```
import Data.Text
```

We have to modify `config/routes` and our handler accordingly. Replace `#String` by `#Text` in `config/routes`:

```
/echo/#Text EchoR GET
```

And do the same in `Handler/Echo.hs`:

```
module Handler.Echo where

import Import

getEchoR :: Text -> Handler
RepHtml
getEchoR theText = do
    defaultLayout $ do
        [whamlet|<h1>#{theText}|]
```

Use Templates

Some HTML (more precisely hamlet) is written directly inside our handler. We should put this part inside another file. Create the new file `templates/echo.hamlet` containing:

```
<h1> #{theText}
```

and modify the handler `Handler/Echo.hs`:

```
getEchoR :: Text -> Handler
RepHtml
getEchoR theText = do
    defaultLayout $ do
        $(widgetFile "echo")
```

At this point, our web application is structured between different files. Handlers are grouped, we use `Data.Text`, and our views are in templates. It is the time to try a slightly more complex example.

Mirror

Let's make another minimal application. You should see a form containing a text field and a validation button. When you enter some text (for example "Jormungad") and validate, the next page presents the content and its reverse appended to it. In our example it should return "JormungaddagnumroJ".

First, add a new route:

```
/mirror MirrorR GET POST
```

This time the path `/mirror` will accept GET and POST requests. Add the corresponding new Handler file:

- `Mirror.hs`

```
module Handler.Mirror where

import Import
import qualified Data.Text as T

getMirrorR :: Handler RepHtml
getMirrorR = do
    defaultLayout $ do
        $(widgetFile "mirror")

postMirrorR :: Handler RepHtml
postMirrorR =  do
        postedText <- runInputPost
$ ireq textField "content"
        defaultLayout $ do
            $(widgetFile "posted")
```

Don't forget to declare it inside `yosog.cabal` and `Application.hs`.

We will need to use the `reverse` function provided by `Data.Text` which explains the additional import.

Create the two corresponding templates:

- `mirror.hamlet`

```
<h1> Enter your text
<form method=post action=@{Mir-
rorR}>
    <input type=text name=content>
    <input type=submit>
```

- `posted.hamlet`

```
<h1>You've just posted
<p>#{postedText}#{T.reverse post-
edText}
<hr>
<p><a href=@{MirrorR}>Get back
```

And that is all. This time, we won't need to clean up. We could have used another way to generate the form, but we'll see this in the next section.

Try it here: *http://localhost:3000/mirror*

Also you can try to enter strange values. As before, your application is quite secure.

A Blog

We saw how to retrieve HTTP parameters. It is the time to save things into a database.

As before, add some routes inside `config/routes`:

```
/blog                 BlogR
GET POST
/blog/#ArticleId      ArticleR
GET
```

This example will be very minimal:

- `GET` on `/blog` should display the list of articles.
- `POST` on `/blog` should create a new article.
- `GET` on `/blog/<article id>` should display the content of the article.

First, we declare another model object. Append the following content to `config/models`:

```
Article
    title   Text
    content Html
    deriving
```

As `Html` is not an instance of `Read`, `Show` and `Eq`, we had to add the `deriving` line. If you forget it, there will be an error.

After the route and the model, we write the handler. First, declare a new Handler module. Add `import Handler.Blog` inside `Application.hs` and add it into `yosog.cabal`. Let's write the content of `Handler/Blog.hs`. We start by declaring the module and by importing some block necessary to handle HTML in forms.

```
module Handler.Blog
    ( getBlogR, postBlogR,
    getArticleR )
where

import Import

-- to use Html into forms
import Yesod.Form.Nic (YesodNic,
nicHtmlField)
instance YesodNic Yosog
```

Remark: it is a best practice to add the `YesodNic` instance inside `Foundation.hs`. I put this definition here to make things easier, but be warned about this orphan instance. To put the include inside `Foundation.hs` is left as an exercise to the reader.

```
entryForm :: Form Article
entryForm = renderDivs $ Article
    <$> areq   textField "Title"
Nothing
    <*> areq   nicHtmlField "Con-
tent" Nothing
```

This function defines a form for adding a new article. Don't pay attention to all the syntax. If you are curious you can take a look at Applicative Functor. You just have to remember `areq` is for required form input. Its arguments being: `areq type label default_value`.

```
-- The view showing the list of
-- articles
getBlogR :: Handler RepHtml
getBlogR = do
    -- Get the list of articles
    -- inside the database.
    articles <- runDB $ selectList
[] [Desc ArticleTitle]
    -- We'll need the two
    -- "objects": articleWidget
    -- and enctype to construct
    -- the form (see templates/
    -- articles.hamlet).
    ((_,articleWidget), enctype)
<- generateFormPost entryForm
    defaultLayout $ do
        $(widgetFile "articles")
```

This handler should display a list of articles. We get the list from the DB and we construct the form. Just take a look at the corresponding template:

- `articles.hamlet`

```
<h1> Articles
$if null articles
  -- Show a standard message if
  -- there is no article
  <p> There are no articles in
the blog
$else
  -- Show the list of articles
  <ul>
    $forall Entity articleId
article <- articles
      <li>
        <a href=@{ArticleR arti-
cleId} > #{articleTitle article}
<hr>
  <form method=post
enctype=#{enctype}>
    ^{articleWidget}
    <div>
      <input type=submit
value="Post New Article">
```

You should notice we added some logic inside the template. There is a test and a "loop."

Another very interesting part is the creation of the form. The `articleWidget` was created by Yesod. We have given him the right parameters (input required or optional, labels, default values). And now we have a protected form made for us. But we have to create the submit button.

Get back to `Handler/Blog.hs`.

```
-- we continue Handler/Blog.hs
postBlogR :: Handler RepHtml
postBlogR = do
  ((res,articleWidget),enctype)
    <- runFormPost entryForm
  case res of
    FormSuccess article -> do
      articleId <- runDB $ insert
article
      setMessage $ toHtml $
        (articleTitle article) <>
" created"
      redirect $ ArticleR arti-
cleId
    _ -> defaultLayout $ do
      setTitle "Please correct
your entry form"
      $(widgetFile
"articleAddError")
```

This function should be used to create a new article. We handle the form response. If there is an error, we display an error page; for example, if we left some required value blank. If things goes right:

- We add the new article inside the DB (`runDB $ insert article`).
- We add a message to be displayed (`setMessage $ ...`).
- We are redirected to the article web page.

Here is the content of the error page:

```
<form method=post
enctype=#{enctype}>
  ^{articleWidget}
  <div>
    <input type=submit value="Post
New Article">
```

Finally, we need to display an article:

```
getArticleR :: ArticleId -> Han-
dler RepHtml
getArticleR articleId = do
    article <- runDB $ get404
articleId
    defaultLayout $ do
        setTitle $ toHtml $ arti-
cleTitle article
        $(widgetFile "article")
```

The `get404` function tries to do a get on the DB. If it fails, it returns a 404 page. The rest should be clear. Here is the content of `templates/article.hamlet`:

```
<h1> #{articleTitle article}
<article> #{articleContent
article}
```

The blog system is finished. Just for fun, you can try to create an article with the following content:

```
<p>A last try to <em>cross
script</em>
   and <em>SQL injection</em></p>
<p>Here is the first try:
   <script>alert("You loose");</
script></p>
<p> And Here is the last </p>
"); DROP TABLE ARTICLE;;
```

Conclusion

This is the end of this tutorial.

If you already know Haskell and you want to go further, you should take a look at the recent i18n blog tutorial [hn.my/i18n]. It will be obvious it inspired my own tutorial. You'll learn in a very straightforward way how easy it is to use authorizations, Time, and internationalization.

If, on the other hand, you don't know Haskell, then you shouldn't jump directly to web programming. Haskell is a very complex and unusual language. My advice for quickly using Haskell for web programming is:

1. Start by trying Haskell in your browser. [tryhaskell.org]
2. Then read the excellent "Learn you a Haskell for Great Good." [learnyouahaskell.com]
3. If you have difficulties in understanding concepts like monads, you should really read these articles. For me, they were enlightening. [hn.my/monads]
4. If you feel confident, you should be able to follow the Yesod book. If you find it difficult to follow the Yesod book, you should read Haskell first (it is a must-read). [book.realworldhaskell.org]

Also, note that:

- *haskell.org* is full of excellent resources.
- hoogle [haskell.org/hoogle] will be very useful.
- Use hlint [hn.my/hlint] as soon as possible to get good habits.

As you can see, if you don't already know Haskell, the path is long but I guarantee it will be very rewarding! ■

PS: You can download the source of this Yesod blog tutorial at github.com/yogsototh/yosog

---

Yann Esposito is the author of YPassword. He co-founded GridPocket and is an active web and iOS developer. He has a post Ph.D. in Machine Learning. He has written two research tools: dees & SEDiL.

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

Designing Great API Docs

By JAMES YU

Writing documentation is one of those things that many developers dread. It takes a lot of effort and time to get right. And too often, people take shortcuts. This is sad, because well designed documentation is the key to getting people excited about your project, whether it's open source or a developer-focused product.

In fact, I argue that the most important piece of UX for a developer product isn't the homepage or the sign up process or the SDK download. It's the API documentation! Who cares if your product is the most powerful thing in the world if no one understands how to use it?

If you're a making a developer-focused product, the documentation is as core to the user experience as the endpoints themselves.

I've seen far too many projects that simply dump you to a GitHub page with a two-liner readme. The most successful API docs are carefully crafted with love. Here at Parse, we devote ourselves to this art.

So, what elements go into making great API documentation?

Documentation is a Layered Beast

Your documentation shouldn't just be a plain listing of endpoints with their parameters. Documentation is a whole ecosystem of content that aims to teach users how to interact with your API. At the very least, you should have these components:

- **Reference:** This is the listing of all the functionality in excruciating detail. This includes all data type and function specs. Your advanced developers will leave this open in a tab all day long.
- **Guides:** This is somewhere between the reference and tutorials. It should be like your reference with prose that explains how to use your API.

> **"The way to a developer's heart is great documentation."**

- **Tutorials:** These teach your users specific things that they can do with your API, and are usually tightly focused on just a few pieces of functionality. Bonus points if you include working sample code.

At Parse, we have all three of these components, and we're currently working on fleshing out more tutorials.

Another good example is Stripe's API, which has an awesome hybrid guide [stripe.com/docs/api] and reference and also a cadre of great tutorials. GitHub's reference API [developer.github.com] is also very well designed.

Remove Abstractions with Examples

You could argue that your API is one big abstraction, and that's kind of the point. However, when teaching developers, try to remove as many abstractions as possible.

Liberally sprinkle real world examples throughout your documentation. No developer will ever complain that there are too many examples. They dramatically reduce the time for developers to understand your product. In fact, we even have example code right on our homepage [parse.com].

Minimize Clicking

It's no secret that developers hate to click. Don't spread your documentation onto a million different pages. Keep related topics close to each other on the same page.

We're big fans of long single page guides that let users see the big picture with the ability to easily zoom into the details with a persistent navigation bar. This has the great side effect that users can search all the content with an in-page browser search.

A great example of this is the Backbone.js documentation [hn.my/bbdoc], which has everything at your fingertips.

Include a Gentle Quickstart

The hardest part of adopting a new API is right at the beginning, where the learning curve is steep and the developer is exposed to complicated new ideas. The solution to this is to introduce the API with a quickstart guide.

The goal of the quickstart is to walk the user through the minimal steps needed to do the smallest thing possible in your API. Nothing more. Once a user has done this, they're ready to move on to the more advanced concepts.

For example, our quickstart guide has the user download our SDKs and then save one object to our platform. We even have a button that lets users test whether they've done this correctly. This gives them the confidence to move on and learn the rest of our platform.

Use Multiple Languages

We live in a polyglot world. If appropriate, try to list examples for multiple languages that your API supports, most likely via client libraries. Learning a new API is hard enough without having to parse unfamiliar languages.

MailGun's API [hn.my/mailgun] does a great job of this by letting developers choose between curl, Ruby, Python, Java, C#, and PHP for the examples in a global menu.

You Can Never Over-communicate Edge Cases

The worst thing is to be developing with an API only to discover an error state that isn't documented. In situations like this, it can be unclear if the error is in your code, or in your understanding of the API.

Your reference should include every edge case and every assumption that is made, either implicitly or explicitly. Spending a few minutes doing this can save hours of your user's collective time.

Sample Applications

At the end of the day, developers want to see the big picture. And the best way to show that is with working sample applications. I find that application code is the best way to communicate how everything in your API ties together and how it integrates with other systems.

A great example of this is sample code in Apple's iOS Developer Library [hn.my/ioslib], which has an exhaustive selection of sample iOS apps organized by topic.

Add Personality

Reading API documentation isn't exactly a thrilling roller coaster ride. But, you can at least add some personality and fun into your writing. Surprise your reader with funny examples and variable names other than foo.

This will at least prevent them from falling asleep.

Conclusion

The way to a developer's heart is great documentation, and great documentation requires a lot of investment. But this investment is well worth it and is just as important for a developer product as the API itself. ■

James Yu is co-founder of Parse, the mobile application platform. He is a hacker, designer, and marketer passionate about building products that people love to use.

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

Automate Everything

By TOM BLOMFIELD

Illustration by John Schwegel (johnschwegel.com)

Performing manual, repetitive tasks enrages me. I used to think this was a corollary of being a programmer, but I've come to suspect (or hope) that this behavior is inherent in being human.

But being able to hack together scripts simply makes it much easier to go from a state of rage to a basic solution in a very small amount of time. As a side point, this is one of the reasons that teaching the basics of programming in schools is so important. It's hard to think of any job which wouldn't benefit from a few simple scripts to perform more automation.

When we're hiring, even for non-developer roles, we look for this kind of mentality — it's extremely useful, especially when building a software businesses, if costs don't scale linearly with revenue. The more we can invest up-front in automation, the less time our team has to spend on performing stupid, manual tasks. As we add more employees, the benefits are compounded. And less rage generally makes the workplace a much happier place.

I encountered a practical example earlier this week. It was time to submit expense reports, and I could feel the rage starting to build up. For some reason, our accountant decreed that we had to fill in a spreadsheet, line-by-line, for every expense item.

Presumably, hundreds of millions of people have to do this every month, costing millions hours of lost productivity. And in most companies, it's because a well-meaning HR or Finance person has said that It Must Be Done. But if that Finance person had a modicum of programming experience, they might be minded to try to find a better way. That's what I mean by hiring people with this kind of mentality. We don't want anyone who'll lump some stupid task on the rest of the team because they've not got the mindset to think about automation in a sensible way. Even if they can't program the solution themselves, they need to be able to figure out pretty quickly that a better solution must exist.

After briefly raging out, I decided to do something about this particular problem.

1 Quickly define the requirements

- Record details of every expense item including date, amount, and description.
- Be able to query the list to produce reports (by month and/or person submitting the receipt).
- Keep copies of receipts for HMRC.

2 Think about how receipts fit into our workflow at the moment, and the major problems

- Receipts can be physical (ex., till receipt for lunch) or electronic (ex., Rackspace email).
- They can pop up at random points in they day — not conveniently all at the same time every month.
- People have to store small pieces of paper for up to 30 days. Unreliable.
- People then have to spend an hour or two each month going through thousands of emails and hundreds of small pieces of paper to find the receipts, remember what they're for, and write them down one-by-one. At least the spreadsheet auto-sums the amounts to a total… Inefficient!

3 Apply a modicum of brainpower to automate the pain-points

To avoid the unreliability of storing small pieces of paper and the inefficiency of examining thousands of emails, perhaps we could store them all in one place, electronically. This place could be an email account called something imaginative like "expenses@mycompany.com".

To avoid having to go through each receipt one-by-one, perhaps some kind of machine could parse relevant information out of each email and persist it

somehow. Perhaps a database might be useful. To produce the report, perhaps the database could output certain information based on some kind of structured query language. So that the people at HRMC don't suffer an instantaneous and fatal brain-explosion when we send them the data, perhaps we could separate values with a comma, save them all in a file, and advise HRMC to open the file with their preferred Microsoft spreadsheet program.

As a side note: in the interest of not re-inventing the wheel, it's generally a good idea to check if someone has solved problem before. I decided to roll my own in this case because I was interested in learning about email handling after watching this great railscast from Ryan Bates [hn.my/mailman]. And because paying $9 per month per user for something I could probably write myself in a couple of hours seemed silly.

Technical Details

If you're interested in the technical details, I used a ruby gem called Mailman, hosted on Heroku's new cedar stack, to poll our POP3 mail server every minute. Attachments (pdfs, photos, etc) are saved to AWS S3, and simple details of each receipt are stored in a postgres database. A Campfire gem called Tinder immediately alerts our company chat room that someone's spent some of our hard-earned cash (just for amusement, really), and a very simple Rails app hosted on Heroku makes the data available in HTML or CSV, which can be queried by date-range or employee.

Emails are required to have a line similar to "Amount 12.50". Running OCR on photos of receipts and detecting the right line to take the Bill total from seemed like too much effort. We might switch to mechanical turk if people find this step troublesome.

Conclusion

Don't put up with repetitive, manual tasks — automate them! It's the hacker way. ■

Tom is a Ruby developer in London. He's currently working at *GoCardless.com*, an online payments company which he founded in 2010.

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

Criticism and Two Way Streets

By DES TRAYNOR

Bill Buxton is a Principal Researcher with Microsoft where his main role focuses on designing an environment that permits great design to happen. As many have learned to their peril, it's not simply a case of just dumping talent in a room full of IKEA furniture. In large companies you have to design the process that creates design. One key idea Bill advocates is an emphasis on exploring the solution space before iterating on a solution.

However, having great designers each producing great solutions to a shared problem can cause conflict if not managed correctly.

Exploring the Solution Space

Like Apple, Microsoft encourages their designers to create many different solutions to any given design problem. But picking an outright winner isn't easy. It can cause arguments and standstills. The quality of resolution here defines the quality of the design process. Who gets to decide? Is it the loudest shouter? The most senior? The highest paid? None of these are correct by default.

When Does Your Solution Suck?

Every solution is great in some circumstances and terrible in others. Design debates are best settled by inviting everyone to present their solution but also explaining under what circumstances their solution is terrible. Finally, they're asked to explain under what circumstances their colleague's

solution would be better. This is what Bill Buxton refers to as walking on both sides of the street.

The person who demonstrates most knowledge about the shortcomings of their own solution and the benefits of all the alternatives is best-equipped to make the call.

Less Time Arguing, More Time Designing

One surprising knock-on effect of this approach is a reduction in pointless design arguments. Those arguments are rarely constructive when people get offended and cling on to their own precious concepts. When everyone must be able to praise their colleague's work and criticize their own, inevitably a solution is agreed upon before a showdown is necessary. Also, by making a rule of praising alternate solutions and criticizing your own, the discussions move clear of the realm of personal preference and bias. It's simply a discussion of what is right and when.

Design decisions should always be based on what's appropriate for the task at hand. If you find your design is being beaten down, the best way to fight back is to counter with "Well, when would my design be appropriate?" Conversely, before you take pleasure in destroying someone else's hard work, first make sure that you can answer the question: "When will this solution be great?"

The quality of critique decides the quality of design output. Giving it five minutes before you criticize certainly helps, as does learning to understand what it's trying to achieve, where it would be right, and where it would be wrong.

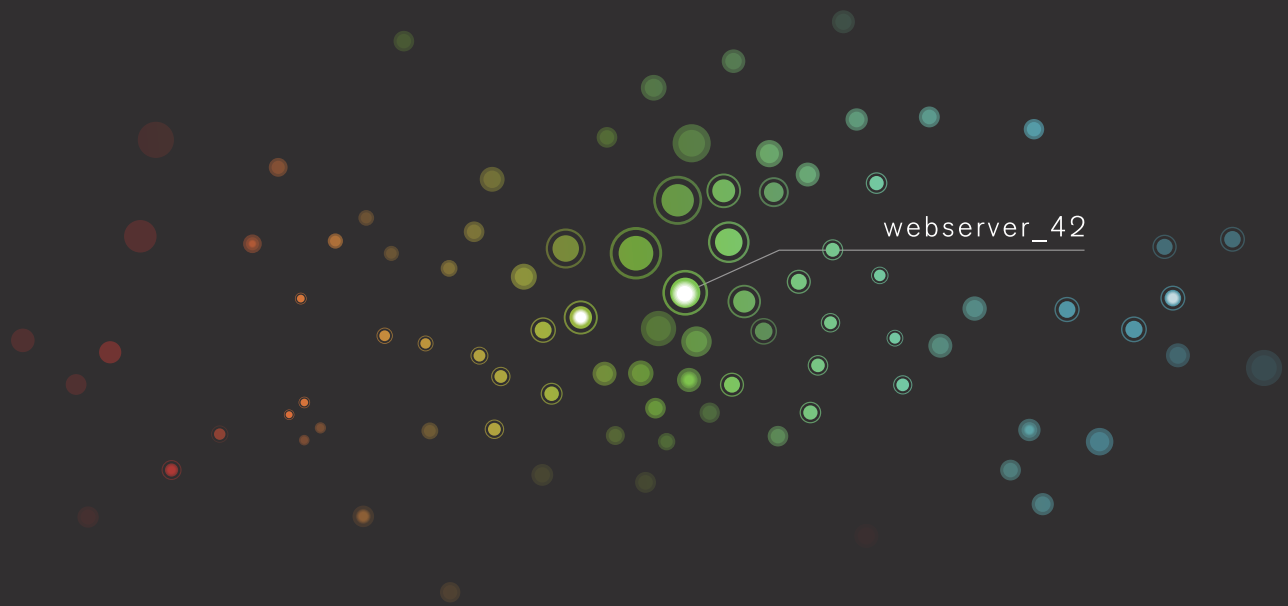
Lastly, always remember the golden rule of critique: don't be a dick. ■

Des Traynor is the COO and UX Lead at Intercom [intercom.io]. He writes regularly about design, start-ups and customer acquisition on The Intercom Blog [blog.intercom.io]. Des is on Twitter as @destraynor, and can be reached at des@intercom.io

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

These are your servers

These are your servers on Cloudkick



Any questions?

cloudkick.com
415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

Uncloaking a Slumlord Conspiracy with Social Network Analysis

By VALDIS KREBS

> **"Sunlight is the best disinfectant.**
> **— U.S. Supreme Court Justice Louis Brandeis"**

A CLIENT OF OURS — a small, not-for-profit, economic justice organization (EJO) — used social network analysis (SNA) to assist their city attorney in convicting a group of "slumlords" of various housing violations that the real estate investors had been side-stepping for years. The housing violations, in multiple buildings, included:

1. Raw sewage leaks
2. Multiple tenant children with high lead levels
3. Eviction of complaining tenants
4. Utility liens of six figures

The EJO had been working with local tenants in run-down properties and soon started to notice some patterns. The EJO began to collect public data on the properties with the most violations. As the collected data grew in size, the EJO examined various ways

they could visualize the data making it clear and understandable to all concerned. They tried various mind-mapping and organization-charting software but to no avail — the complex ties they were discovering just made the diagrams hopelessly unreadable. They turned to social network analysis to make sense of the complex interconnectivity.

The data I will present below is not the actual data from the criminal case. However, it does accurately reflect the social network analysis they performed. The names and genders of the individuals, as well as the names of real estate holdings (LLC) and other businesses have all been masked. This case will be presented in the sequence the EJO followed: first they looked at the real estate holdings, then the owners of the holdings, and then their connections, which led to other connections, and more people and entities.

The EJO worked with the tenants and city inspectors to assess the buildings and document the violations. But every time documented problems were delivered to the current LLC owners by city officials, nothing would happen. When the city's deadline approached to fix the violations, the old LLC owner would explain that the property had changed hands and they were no longer involved. The buildings continued to deteriorate as owner after owner avoided addressing the violations.

Figure 1

Figure 1 shows how a building came under new ownership. The gray links show the "sold to" flow as building ownership changed from left to right. Every time a property changed hands, it became a new LLC (Limited Liability Corporation) with new owners.

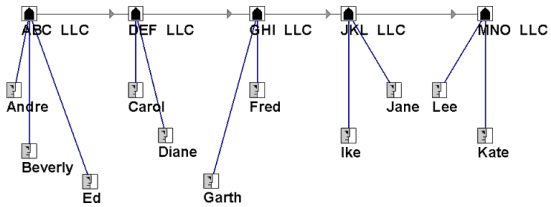


Figure 2

The blue links in Figure 2 show ownership/business ties for each LLC. This data was gathered by the EJO from public records. Everything appears normal — a different set of players in each LLC.

Yet, things were not normal. The EJO discovered that some of the LLC owners were married. As the EJO peeled the onion, more family ties were found within, and between, the LLCs.

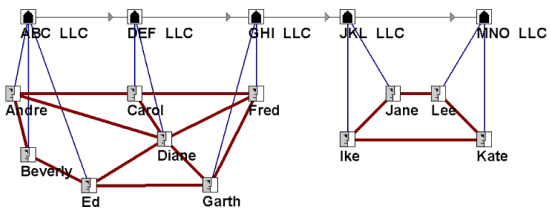


Figure 3

Figure 3 shows us that these LLCs were not as separate as they first appeared. The dark red links reveal family ties found in public records. The LLCs were not independent business entities. The business transactions were happening within extended families! A conspiracy was coming into focus.

The dark red links in Figure 3 reveal two family clusters. Yet, there was a curious gap — the transaction between ghi LLC andjkl LLC. Were these clusters connected? How? These questions soon led to a key discovery: the mastermind behind the conspiracy. Conspiracies often work in this way — masterminds are two steps or more from the events they planned.

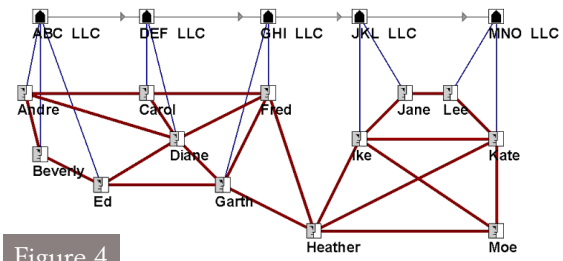


Figure 4

Figure 4 reveals the family matriarch and patriarch. The matriarch (Heather) was discovered in public records, explaining the gap. Then her current husband (Moe) was a quick deduction. The gap turned out to be the dividing line between Heather's first family and her current family. She was the point of overlap between the two groups.

Once Moe was uncloaked, the EJO's chief investigator decided to explore how he was connected — what other business ties did he have? It turned out that Moe had ownership interests in several restaurants throughout the metropolitan area...and he was on the board of a mortgage company.

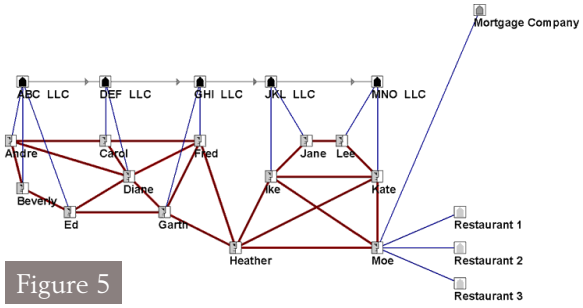


Figure 5

A mortgage company? It was not just any mortgage company, Moe was on the board of the mortgage company that had financed many of the real estate transactions we have been following here. Moe's ties completed the connections of the conspiracy — the "circle of deceit."

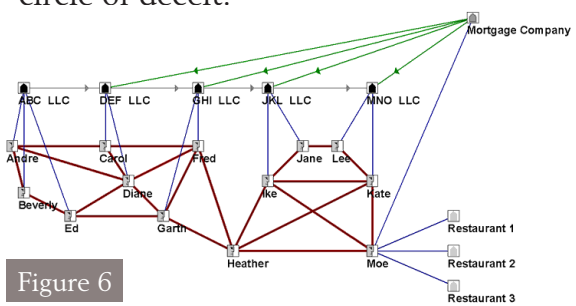


Figure 6

Figure 6 shows the complete conspiracy. It was now obvious that properties exchanged hands not as independent and valid real estate investments but as a conspiracy to avoid fixing the

building violations. The green links represent borrowed money flowing into the buildings through new mortgages. As time went on, and the buildings appreciated in value during a real estate boom. Loans from the mortgage company allowed the owners to "strip mine" the equity from the buildings. This is a common slumlord modus operandi: they suck money out of a building rather than put money back in for maintenance.

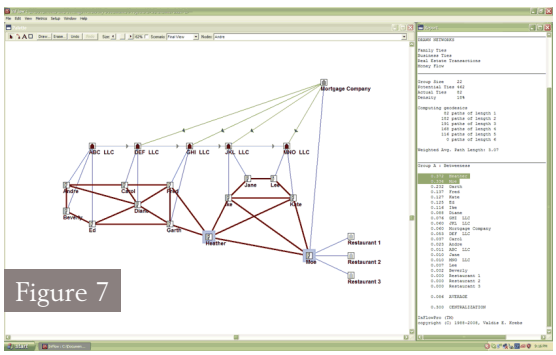


Figure 7

Network analysis is not just about maps. Once a map is drawn, you can measure it. Social network metrics reveal much about the nodes and the clusters they form. Who knows what is going on? Who wields power or influence? Who is a key connector? Who is in the "thick of things" in this conspiracy? Our metrics reveal Moe and Heather are most integrated nodes in the network. The highlighted metrics in the Report window in Figure 7, showing the InFlow software, provides mathematical support to what is quite obvious in the diagrams. InFlow [orgnet.com/inflow3.html] allows us to quickly see the relationships between "the maps and the metrics": the pictures and the numbers.

The city attorney combined the network analysis, along with the city's own extensive investigation and was able to get a conviction of key family members. Later, all of one building's tenants filed a civil suit using much of the same evidence and won a sufficient award to allow all of them to move out into decent housing. Several tenants used a part of their award to start businesses.

The common wisdom is that only big business and government use social network analysis. Yet, there are many individuals and groups that are learning the craft, and solving local problems. Although social network analysis can not be learned by reading a book, it does not require a PhD either. Any intelligent person, under the right guidance, and with the proper tools, can apply the methodology to an appropriate problem and gain enormous insight into what was previously hidden. ■

Valdis is the Founder, and Chief Scientist, at *orgnet.com*. He is a management consultant, researcher, trainer, author, and the developer of InFlow software for social and organizational network analysis (SNA/ONA).

Reprinted with permission of the original author.
First appeared in *orgnet.com/slumlords.html*

Our users are saying...

"I enjoy the feeling of being recognized but mostly I enjoy knowing my friends are happy."

"It's cool realising that people thought enough about your contribution to give props."

When your people feel appreciation every day they are **happier**, more **engaged** and **more productive**.

Get started today at dueprops.com

Due Props
Better Recognize

Dream. Design. Print.

MagCloud, the revolutionary new self-publishing web service by HP, is changing the way ideas, stories, and images find their way into peoples' hands in a printed magazine format.

HP MagCloud capitalizes on the digital revolution, creating a web-based marketplace where traditional media companies, upstart magazine publishers, students, photographers, designers, and businesses can affordably turn their targeted content into print and digital magazine formats.

Simply upload a PDF of your content, set your selling price, and HP MagCloud takes care of the rest—processing payments, printing magazines on demand, and shipping orders to locations around the world. All magazine formatted publications are printed to order using HP Indigo technology, so they not only look fantastic but there's no waste or overruns, reducing the impact on the environment.

Become part of the future of magazine publishing today at www.magcloud.com.

25% Off the First Issue You Publish

Enter promo code **HACKER** when you set your magazine price during the publishing process.

Please contact promo@magcloud.com with any questions.

MAGCLOUD