



hacker **bits**

Issue 12



new bits

Is interviewing candidates a good way to hire for the tech industry?

Developer Amir Yasin doesn't think so. In this issue's most highly-upvoted article, he shows us how to hire the best candidate for the job with 2 innovative ideas.

Who says boring is bad?

Just ask Jason Kester who'd eschewed new shiny tech for boring stacks and found happiness along the way. In this short and enlightening piece, Jason shows us how a boring stack can lead one to developer joy.

If you've ever wondered how to be a contributor on GitHub, then look no further than Davide Coppola's handy tutorial on how to contribute to an open source project on GitHub.

As we wrap up the 12th issue of *Hacker Bits*, we want to thank you all for reading and offering feedback on improving the magazine. We hope *Hacker Bits* has helped you stay current on tech, and we look forward to another exciting year where you'll learn more and read less!

Happy holidays, and see you back here next year!

— *Maureen and Ray*

us@hackerbits.com

content bits

Issue 12

6 Service workers: an introduction

16 Ways data projects fail

25 A method I've used to eliminate bad tech hires

29 To-do lists are not the answer to getting things done

33 You are not paid to write code

36 What I learned from spending 3 months applying to jobs

43 A beginner's guide to Big O notation

46 Happiness is a boring stack

48 How to contribute to an open source project on GitHub

54 It was never about ops

contributor bits



Michael Biven

Michael is an engineering manager, systems engineer and a former firefighter who has discovered he enjoys figuring out how to help scale teams as much as he does scaling the technology.



Martin Goodson

Martin is a founder of the data science consultancy [Evolution AI](#). He specialises in Natural Language Processing using internet-scale data sets. He built and led Data Science teams at 2 London Startups. His data products are in use at Time Inc, Staples, John Lewis, Top Shop, Conde Nast, NY-Times, BuzzFeed, etc.



Amir Yasin

Amir is a polyglot developer deeply interested in high performance, scalability, software architecture and generally solving hard problems. You can follow him on [Medium](#), [Twitter](#) or [GitHub](#).



Shane Parrish

Shane has spent the last several years collecting ideas that survive the test of time by reading thousands of books and having conversations with hundreds of authors and high-performers. He shares distilled versions of these on his website [Farnam Street](#).



Tyler Treat

Tyler is a backend engineer at Workiva where he works on distributed systems and platform infrastructure. In his spare time, Tyler contributes to various open source projects.



Felix Feng

Felix is a Software Engineer at Radius Intelligence where he works to build out Radius's client-facing application. Previously, he worked in Finance at 21.co and as an analyst at an investment bank. He graduated from UC Berkeley.



Rob Bell

Rob is a software engineer from London. He's been building software for the retail, media and finance industries for just over a decade. Outside of work he's built games and robotic drum machines, and blogs about that and more at [rob-bell.net](#).



Jason Kester

Jason is a serial entrepreneur building low-maintenance SaaS products such as [S3stat](#) and [Unwaffle](#) while traveling the world in search of pleasant spots to climb rocks and surf. He'll try to convince you to do the same at [expatsoftware.com](#).



Davide Coppola

Davide is a game, software and web developer founder of [VindictaGames](#) and available for hire. He's also blogging every week about tech topics on his [blog](#). You can find more about him on his personal [website](#).



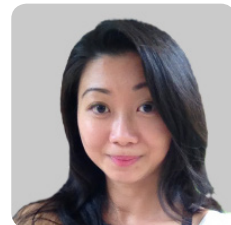
Matt Gaunt

Matt is a Senior Developer Programs Engineer at Google. He generally works on the open web these days, but used to be a full time mobile software engineer. Check out his [blog](#) or [@gauntface](#).



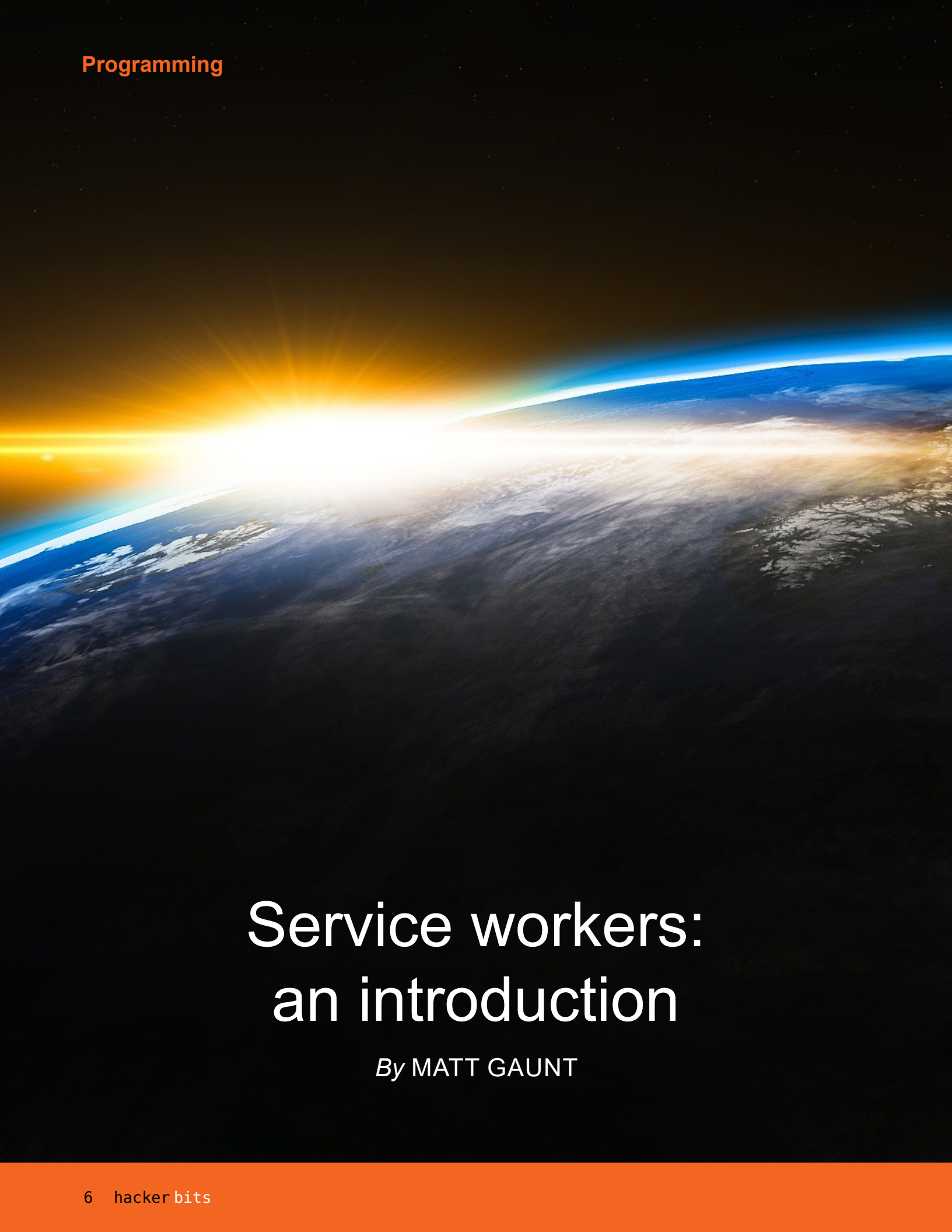
Ray Li Curator

Ray is a software engineer and data enthusiast who has been blogging at [rayli.net](#) for over a decade. He loves to learn, teach and grow. You'll usually find him wrangling data, programming and lifehacking.



Maureen Ker Editor

Maureen is an editor specializing in technical and science writing. She is the author of 3 books and 100+ articles, and her work has appeared in the *New York Daily News*, and various adult and children's publications. In her free time, she enjoys collecting useless kitchen gadgets.



Service workers: an introduction

By MATT GAUNT

Rich offline experiences, periodic background syncs, push notifications—functionality that would normally require a native application—are coming to the web. Service workers provide the technical foundation that all these features rely on.

What is a service worker

A service worker is a script that your browser runs in the background, separate from a web page, opening the door to features that don't need a web page or user interaction. Today, they already include features like [push notifications](#) and [background sync](#). In the future service workers will support other things like periodic sync or geofencing. The core feature discussed in this tutorial is the ability to intercept and handle network requests, including programmatically managing a cache of responses.

The reason this is such an exciting API is that it allows you to support offline experiences, giving developers complete control over the experience.

Before service worker, there was one other API that gave users an offline experience on the web called [AppCache](#). The major issues with AppCache are the [number of gotcha's](#) that exist as well as the fact that while the design works particularly well for single page web apps, it's not so good with for multi-page sites. Service workers have been designed to avoid these common pain points.

Things to note about a service worker:

- It's a [JavaScript Worker](#), so it can't access the DOM directly. Instead, a service worker can communicate with the pages it controls by responding to messages sent via the [postMessage](#) interface, and those pages can manipulate the DOM if needed.
- Service worker is a programmable network proxy, allowing you to control how network requests from your page are handled.
- It's terminated when not in use, and restarted when it's next needed, so you cannot rely on global state within a service worker's `onfetch` and `onmessage` handlers. If there is information that you need to persist and reuse across restarts, service workers do have access to the [IndexedDB API](#).
- Service workers make extensive use of promises, so if you're new to promises, then you should stop reading this and check out [Promises, an introduction](#).

The service worker life cycle

A service worker has a lifecycle that is completely separate from your web page.

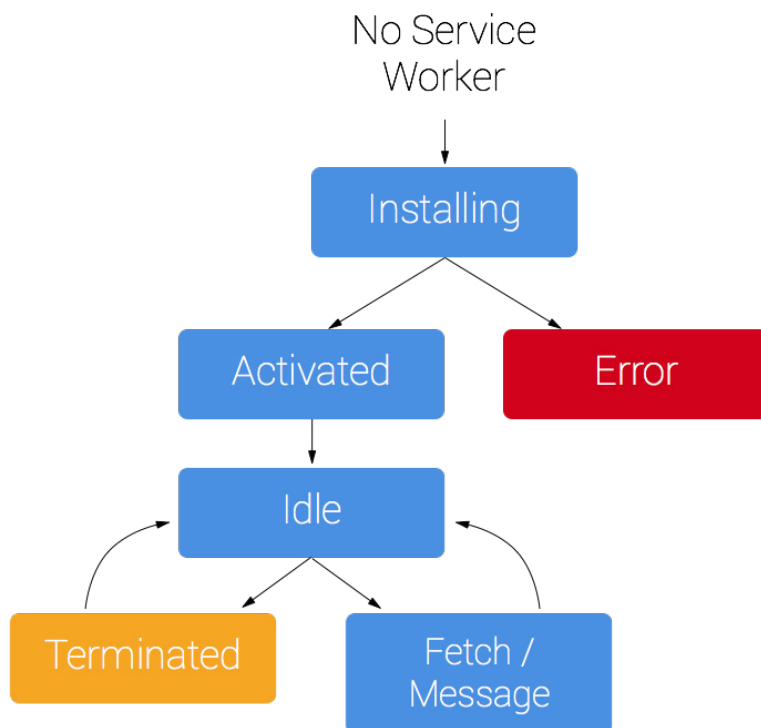
To install a service worker for your site, you need to register it, which you do in your page's JavaScript. Registering a service worker will cause the browser to start the service worker install step in the background.

Typically during the install step, you'll want to cache some static assets. If all the files are cached successfully, then the service worker becomes installed. If any of the files fail to download and cache, then the install step will fail and the service worker won't activate (i.e. won't be installed). If that happens, don't worry, it'll try again next time. But that means if it does install, you know you've got those static assets in the cache.

When we're installed, the activation step will follow and this is a great opportunity for handling any management of old caches, which we'll cover during the service worker update section.

After the activation step, the service worker will control all pages that fall under its scope, though the page that registered the service worker for the first time won't be controlled until it's loaded again. Once a service worker is in control, it will be in one of two states: either the service worker will be terminated to save memory, or it will handle fetch and message events that occur when a network request or message is made from your page.

Below is an overly simplified version of the service worker lifecycle on its first installation.



Prerequisites

Browser support

Browser options are growing. Service workers are supported by Firefox and Opera. Microsoft Edge is now [showing public support](#). Even Safari has dropped [hints of future development](#). You can follow the progress of all the browsers at Jake Archibald's [ServiceWorker ready site](#).

You need HTTPS

During development you'll be able to use service worker through localhost, but to deploy it on a site you'll need to have HTTPS setup on your server.

Using service worker you can hijack connections, fabricate, and filter responses. Powerful stuff. While you would use these powers for good, a man-in-the-middle might not. To avoid this, you can only register service workers on pages served over HTTPS, so we know the service worker the browser receives hasn't been tampered with during its journey through the network.

[Github Pages](#) are served over HTTPS, so they're a great place to host demos.

If you want to add HTTPS to your server then you'll need to get a TLS certificate and set it up for your server. This varies depending on your setup, so check your server's documentation and be sure to check out [Mozilla's SSL config generator](#) for best practices.

Register a service worker

To install a service worker you need to kick start the process by **registering** it in your page. This tells the browser where your service worker JavaScript file lives.

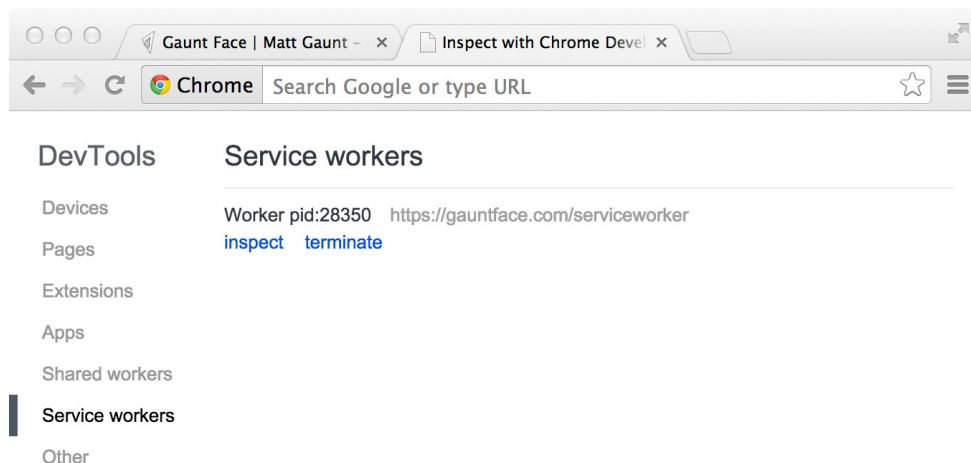
```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/sw.js').then(function(registration) {
      // Registration was successful
      console.log('ServiceWorker registration successful with scope: ', registration.scope);
    }).catch(function(err) {
      // registration failed :(
      console.log('ServiceWorker registration failed: ', err);
    });
  });
}
```

This code checks to see if the service worker API is available, and if it is, the service worker at `/sw.js` is registered [once the page is loaded](#).

You can call `register()` every time a page loads without concern; the browser will figure out if the service worker is already registered or not and handle it accordingly.

One subtlety with the `register()` method is the location of the service worker file. You'll notice in this case that the service worker file is at the root of the domain. This means that the service worker's scope will be the entire origin. In other words, this service worker will receive fetch events for everything on this domain. If we register the service worker file at `/example/sw.js`, then the service worker would only see fetch events for pages whose URL starts with `/example/` (i.e. `/example/page1/`, `/example/page2/`).

Now you can check that a service worker is enabled by going to `chrome://inspect/#service-workers` and looking for your site.



When service worker was first being implemented you could also view your service worker details through `chrome://serviceworker-internals`. This may still be useful, if for nothing more than learning about the life cycle of service workers, but don't be surprised if it gets replaced completely by `chrome://inspect/#service-workers` at a later date.

You may find it useful to test your service worker in an Incognito window so that you can close and reopen knowing that the previous service worker won't affect the new window. Any registrations and caches created from within an Incognito window will be cleared out once that window is closed.

Install a service worker

After a controlled page kicks off the registration process, let's shift to the point of view of the service worker script, which handles the install event.

For the most basic example, you need to define a callback for the install event and decide which files you want to cache.

```
self.addEventListener('install', function(event) {
  // Perform install steps
});
```

Inside of our install callback, we need to take the following steps:

1. Open a cache.
2. Cache our files.
3. Confirm whether all the required assets are cached or not.

```
var CACHE_NAME = 'my-site-cache-v1';
var urlsToCache = [
  '/',
  '/styles/main.css',
  '/script/main.js'
];

self.addEventListener('install', function(event) {
  // Perform install steps
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});
```

Here you can see we call `caches.open()` with our desired cache name, after which we call `cache.addAll()` and pass in our array of files. This is a chain of promises (`caches.open()` and `cache.addAll()`). The `event.waitUntil()` method takes a promise and uses it to know how long installation takes, and whether it succeeded.

If all the files are successfully cached, then the service worker will be installed. If **any** of the files fail to download, then the install step will fail. This allows you to rely on having all the assets that you defined, but does mean you need to be careful with the list of files you decide to cache in the install

step. Defining a long list of files will increase the chance that one file may fail to cache, leading to your service worker not getting installed.

This is just one example, you can perform other tasks in the install event or avoid setting an install event listener altogether.

Cache and return requests

Now that you've installed a service worker, you probably want to return one of your cached responses, right?

After a service worker is installed and the user navigates to a different page or refreshes, the service worker will begin to receive fetch events, an example of which is below.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        // Cache hit - return response
        if (response) {
          return response;
        }
        return fetch(event.request);
      })
  );
});
```

Here we've defined our fetch event and within `event.respondWith()`, we pass in a promise from `caches.match()`. This method looks at the request and finds any cached results from any of the caches your service worker created.

If we have a matching response, we return the cached value, otherwise we return the result of a call to `fetch`, which will make a network request and return the data if anything can be retrieved from the network. This is a simple example and uses any cached assets we cached during the install step.

If we want to cache new requests cumulatively, we can do so by handling the response of the fetch request and then adding it to the cache, like below.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        // Cache hit - return response
        if (response) {
          return response;
        }

        // IMPORTANT: Clone the request. A request is a stream and
        // can only be consumed once. Since we are consuming this
        // once by cache and once by the browser for fetch, we need
        // to clone the response.
        var fetchRequest = event.request.clone();

        return fetch(fetchRequest).then(
```

```

function(response) {
  // Check if we received a valid response
  if(!response || response.status !== 200 || response.type !== 'basic') {
    return response;
  }

  // IMPORTANT: Clone the response. A response is a stream
  // and because we want the browser to consume the response
  // as well as the cache consuming the response, we need
  // to clone it so we have two streams.
  var responseToCache = response.clone();

  caches.open(CACHE_NAME)
    .then(function(cache) {
      cache.put(event.request, responseToCache);
    });

  return response;
}
);
})
);
});

```

What we are doing is this:

1. Add a callback to `.then()` on the `fetch` request.
2. Once we get a response, we perform the following checks:
3. Ensure the response is valid.
4. Check the status is `200` on the response.
5. Make sure the response type is **basic**, which indicates that it's a request from our origin. This means that requests to third party assets aren't cached as well.
6. If we pass the checks, we [clone](#) the response. The reason for this is that because the response is a [Stream](#), the body can only be consumed once. Since we want to return the response for the browser to use, as well as pass it to the cache to use, we need to clone it so we can send one to the browser and one to the cache.

Update a service worker

There will be a point in time where your service worker will need updating. When that time comes, you'll need to follow these steps:

1. Update your service worker JavaScript file. When the user navigates to your site, the browser tries to redownload the script file that defined the service worker in the background. If there is even a byte's difference in the service worker file compared to what it currently has, it considers it *new*.
2. Your new service worker will be started and the `install` event will be fired.
3. At this point the old service worker is still controlling the current pages so the new service worker will enter a `waiting` state.
4. When the currently open pages of your site are closed, the old service worker will be killed and the new service worker will take control.
5. Once your new service worker takes control, its `activate` event will be fired.

One common task that will occur in the `activate` callback is cache management. The reason you'll want to do this in the `activate` callback is because if you were to wipe out any old caches in the install step, any old service worker, which keeps control of all the current pages, will suddenly stop being able to serve files from that cache.

Let's say we have one cache called `'my-site-cache-v1'`, and we find that we want to split this out into one cache for pages and one cache for blog posts. This means in the install step we'd create two caches, `'pages-cache-v1'` and `'blog-posts-cache-v1'` and in the activate step we'd want to delete our older `'my-site-cache-v1'`.

The following code would do this by looping through all of the caches in the service worker and deleting any caches that aren't defined in the cache whitelist.

```
self.addEventListener('activate', function(event) {

  var cacheWhitelist = ['pages-cache-v1', 'blog-posts-cache-v1'];

  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

Rough edges and gotchas

This stuff is really new. Here's a collection of issues that get in the way. Hopefully this section can be deleted soon, but for now these are worth being mindful of.

If installation fails, we're not so good at telling you about it

If a worker registers, but then doesn't appear in `chrome://inspect/#service-workers` or `chrome://serviceworker-internals`, it's likely failed to install due to an error being thrown, or a rejected promise being passed to `event.waitUntil()`.

To work around this, go to `chrome://serviceworker-internals` and check "Open DevTools window and pause JavaScript execution on service worker startup for debugging", and put a debugger statement at the start of your install event. This, along with [Pause on uncaught exceptions](#), should reveal the issue.

The defaults of fetch()

No credentials by default

When you use `fetch`, by default, requests won't contain credentials such as cookies. If you want credentials, instead call:

```
fetch(url, {
  credentials: 'include'
})
```

This behaviour is on purpose, and is arguably better than XHR's more complex default of sending credentials if the URL is same-origin, but omitting them otherwise. Fetch's behaviour is more like other CORS requests, such as ``, which never sends cookies unless you opt-in with ``.

Non-CORS fail by default

By default, fetching a resource from a third party URL will fail if it doesn't support CORS. You can add a `no-CORS` option to the Request to overcome this, although this will cause an 'opaque' response, which means you won't be able to tell if the response was successful or not.

```
cache.addAll(urlsToPrefetch.map(function(urlToPrefetch) {
  return new Request(urlToPrefetch, { mode: 'no-cors' });
})).then(function() {
  console.log('All resources have been fetched and cached.');
```

Handling responsive images

The `srcset` attribute or the `<picture>` element will select the most appropriate image asset at run time and make a network request.

For service worker, if you wanted to cache an image during the install step, you have a few options:

1. Install all the images that the `<picture>` element and the `srcset` attribute will request.
2. Install a single low-res version of the image.
3. Install a single high-res version of the image.

Realistically you should be picking option 2 or 3 since downloading all of the images would be a waste of storage space.

Let's assume you go for the low res version at install time and you want to try and retrieve higher res images from the network when the page is loaded, but if the high res images fail, fallback to the low res version. This is fine and dandy to do but there is one problem.

If we have the following two images:

Screen Density	Width	Height
1x	400	400
2x	800	800

In a `srcset` image, we'd have some markup like this:

```

```


If we are on a 2x display, then the browser will opt to download `image-2x.png`, if we are offline you could `.catch()` this request and return `image-src.png` instead if it's cached, however the browser will expect an image that takes into account the extra pixels on a 2x screen, so the image will appear as 200x200 CSS pixels instead of 400x400 CSS pixels. The only way around this is to set a fixed height and width on the image.

```

```

For `<picture>` elements being used for art direction, this becomes considerably more difficult and will depend heavily on how your images are created and used, but you may be able to use a similar approach to `srcset`.

Learn more

There is a list of documentation on service worker being maintained at <https://jakearchibald.github.io/isserviceworkerready/resources> that you may find useful.

Get help

If you get stuck then please post your questions on Stackoverflow and use the '[service-worker](#)' tag so that we can keep a track of issues and try and help as much as possible. ■

Data

Ways data projects fail

By MARTIN GOODSON

Introduction

Data science continues to generate excitement and yet real-world results can often disappoint business stakeholders. How can we mitigate risk and ensure results match expectations? Working as a technical data scientist at the interface between R&D and commercial operations has given me an insight into the traps that lie in our path. I present a personal view on the most common failure modes of data science projects.

The long version with slides and explanatory text below. Slides in one pdf [here](#).

There is some discussion at [Hacker News](#)

First, about me:


About Me

Led Data Science teams at two London Startups

Data products in use at Time Inc, Staples, John Lewis, Top Shop, Conde Nast, New York Times, BuzzFeed, etc

This talk is based on conversations I've had with many senior data scientists over the last few years. Many companies seems to go through a pattern of hiring a data science team only for the entire team to quit or be fired around 12 months later. Why is the failure rate so high?

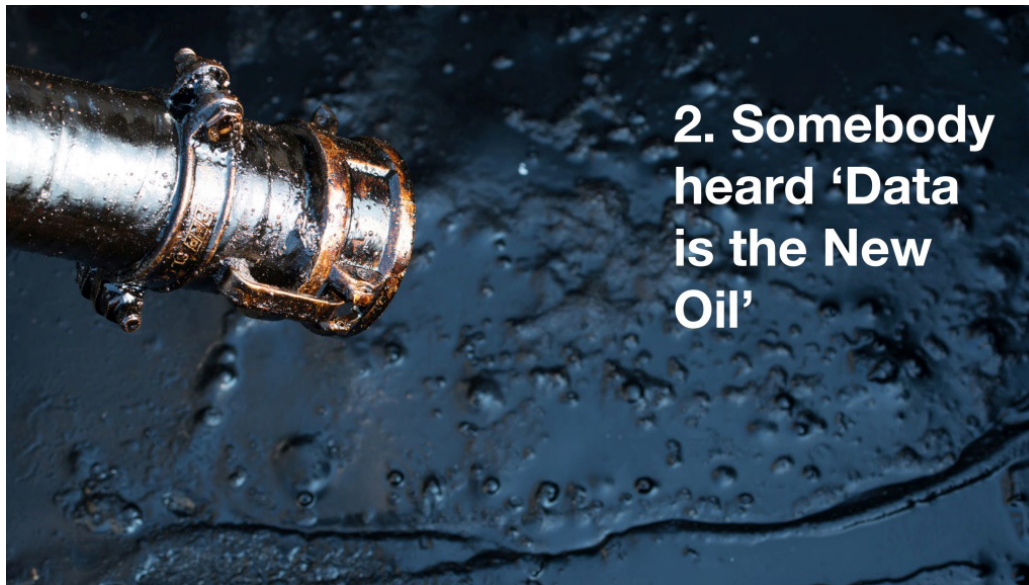
Let's begin:

The slide features a blue background on the left with the title '1. Your data isn't ready' in white. The right side of the slide has a background image of many Euro banknotes of various denominations (5, 10, 20, 50, 100, 200) scattered and overlapping. Overlaid on this image is white text that reads: 'If the data is in a database it must be usable, right? Assume it's garbage unless it's been used before. Do a data audit'.

A very wise data science consultant told me he always asks if the data has been used before in a project. If not, he adds 6-12 months onto the schedule for data cleansing.

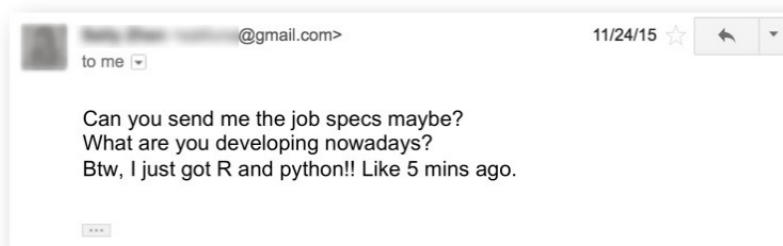
Do a data audit before you begin. Check for missing data, or dirty data. For example, you might find

that a database has different transactions stored in dollar and yen amounts, without indicating which was which. This actually happened.



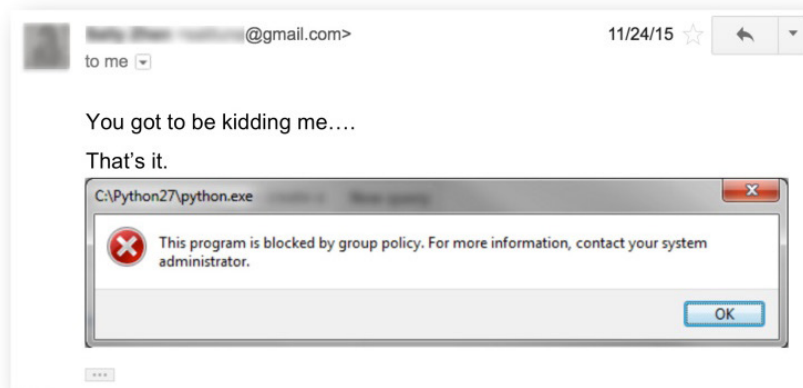
No it isn't. Data is not a commodity, it needs to be transformed into a product before it's valuable. Many respondents told me of projects which started without any idea of who their customer is or how they are going to use this "valuable data". The answer came too late: "nobody" and "they aren't".

3. Your data scientists are about to quit



Don't torture your data scientists by withholding access to the data and tools they need to do their job. This senior data scientist took six weeks to get permission to install python and R. She was so happy!

Well, she was until she sent me this shortly afterwards:



Now, allow me to introduce this guy:



He was a product manager at an online auction site that you may have heard of. His story was about an A/B test of a new prototype algorithm for the main product search engine. The test was successful and the new algorithm was moved to production.

Unfortunately, after much time and expense, it was realised that there was a bug in the A/B test code: the prototype had not been used. They had accidentally tested the old algorithm against *itself*. The results were nonsense.

This was the problem:

4. You don't have a data scientist leader

You won't know the results are trash

Selection bias, measurement bias, Simpson's paradox, statistical significance, etc.

R&D is hard

You need people who live and breathe selection bias, measurement bias, etc. or you'll never know the results are meaningless. These people are called scientists.

Oh, by the way:

This is *not* a data scientist

'Analytics leader shaping strategy for information management, BI, and analytics solutions to support organisational transformation. Proven track record of leading teams in the delivery of enterprise class solutions and maximising business value.'

This *is* a data scientist

'Specialities: Probabilistic programming, Data Analysis, Bayesian Modelling, Hidden Markov models, MCMC, LSTMs, Multitask learning, Domain Adaptation.'

Likewise - the opposite is very often true:

5. You shouldn't have hired scientists*

* See (3)

For ETL, hire data engineers

For reporting, hire BI analysts

The end

6. Your boss read a blog post about machine learning

The hype around machine learning means there is a lot of readily available content out there. This can lead to the ‘instant expert’ phenomenon: now everyone has a great machine learning idea. Symptoms are your boss using words like ‘regularize’ and ‘ensemble method’ in the wrong context. Trust me, this will not end well.



The Cost-Effective HealthCare project used data from hospitals to process Emergency Room patients who had symptoms of pneumonia. They aimed to build a system that could predict people who had a low probability of death, so they could be simply sent home with antibiotics. This would allow care to be focused on the most serious cases, who were likely to suffer complications.

The neural network they developed had a very high accuracy but, strangely, it always decided to send asthma sufferers home. Weird, since asthmatics are actually at *high* risk of complications from pneumonia.

It turned out that asthmatics who present with pneumonia are always admitted to Intensive Care. Because of this, there were no cases of any asthmatics dying in the training data. The model concluded that asthmatics were low risk, when the opposite was actually true. The model had great accuracy but if deployed in production it would certainly have killed people.

7. Your models are too complex

Use an interpretable model first

Test against a baseline

Moral of the story: use a simple model you can understand [1]. Only then move onto something more complex, and only if you *need* to.

8. Your results are not reproducible

Git

Code Review

Automated
testing

Data pipeline
orchestration

The core of science is reproducibility. Please do all of these things. Don't say I didn't warn you.

9. An R&D lab is alien to your company culture.

People prefer gut instinct

R&D is a high risk activity.

Lab meetings, talks, publishing papers etc.?

An applied science lab is a big commitment. Data can often be quite threatening to people who prefer to trust their instincts. R&D has a high risk of failure and unusually high levels of perseverance are table stakes. Do some soul searching - will your company really accept this culture?



10. Designing data products without seeing live data is like doing taxidermy without looking at live animals.

Never let UX designers and product managers design a data product (even wireframes) using fake data. As soon as you apply real data it will be apparent that the wireframe is complete fantasy. The real data will have weird outliers, or be boring. It will be too dynamic. It will be either too predictable or not predictable enough. Use live data from the beginning or your project will end in misery and self-hatred. Just like this poor leopard, weasel thing. ■

[1] See [here](#) for more.

Please get in touch with your own data disaster. Let's get some data behind this and move Data Science forwards. All results will be fully anonymised. Use the form below or contact me via:

Twitter: [@martingoodson](#), Email: martingoodson@gmail.com

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

Interesting



A method I've used to eliminate bad tech hires

By AMIR YASIN

This article is the first in the Tech Hiring & Team Building series. You can find [the second installment here](#).

Let's be real. Interviews are a terrible way to hire tech candidates. Not only do you not get a real sense of the candidate, they often weed out good candidates in favor of bad ones.

In this [fantastic article on Medium by Eric Elliott](#), he talks about many of the techniques that work and don't work for interviewing engineers.

In regards to what works the best, I found that these 2 ideas work the best when combined.

- *PAID Sample project assignment (err on the side of paying fairly—say \$100+/hour for estimated completion time—if the problem should require 2 hours to complete, offer \$200)*
- *Bring the candidate in and discuss the solution. Let the candidate talk about their design decisions, challenge them as you would any team member and let them provide their reasoning.*

Paying candidates to work on a simple project and then discussing it with our team has almost single handedly eliminated any bad hiring decisions. Paying a candidate that gives you a terrible solution (or no solution) is FAR cheaper (both financially and emotionally) than hiring the wrong person, going through a 3 month performance improvement plan to try to make them the right person and eventually firing them.

We have gone from “This candidate is exactly what we need” to “I have serious doubts about working with this candidate long term” and we've had candidates change our bad perception of them using this technique. It's very easy for someone to embellish (to put it generously) their resume, and coding trivia can be memorized, but it's really hard for someone to fake actual skill.

Here's why paying candidates to solve problems works

For the employer

- Since the candidate is getting paid, the candidate treats it as a legit consulting session and therefore gives her best effort in solving the problem.
- It's an indication to the candidate of how they will be treated in the near future if they decide to join.
- It allows a mock work interaction with the candidate at a very low risk cost to you. After the project is complete, you can bring the candidate in and ask them questions about their actual design/coding decisions. This lets you get a sense of how they communicate, take criticism/questioning and if they are able to provide solid reasoning behind their choices.

For the candidate

- It gives the candidate a real life sense of how you interact with people on the team
- Allows them to showcase some of the skills that are on their resume, but may not pop from just reading it
- Allows them to give you a sense of what they feel is important in a non-judgmental way. For example, did the person write a test for every line of code? No? Maybe that's not really important to them and maybe that's totally valid. You'd never get that from just asking them do you do TDD? Everyone will say tests are important...and they are. The thing most people disagree about is how important, this will show that.

I have 6 rules when giving this interview method

RULE #1—Give them the weekend to solve the problem.

This is where I and Eric slightly disagree. 2 hours just isn't enough time to see how well someone can come up with an appropriate solution.

What I like to do is invite them to the office on a Friday and go over the problem at hand and how I would like for them to solve it. Then I'll hand it off to them and set up time on Monday to review their solution.

I'll provide them with certain technologies that should be used for the solution and let them use other tools or technologies at their discretion. For example, I may say "please use functional reactive principles in JS to solve this problem", but the decision of using Kefir, Bacon, or RX is left up to them.

RULE #2—DON'T use a real problem because of the tribal knowledge needed to fix.

This goes hand in hand with Rule #1, but unless you're hiring a customer service rep, it's almost impossible to hand someone a computer and say OK, Fix this issue happening in our proprietary system using the tools that normally everyone gets properly trained on.

Give them a problem that is very self-contained.

RULE #3—The solution you're expecting should be clear, but open for improvement.

For example, if I'm interviewing a web developer, I'll give him a sample clear scenario:

Create a single page app that lets me enter movies in my home movie collection, store them in offline storage and search through them. I want to search by Genre, Title & Actors.

I actually don't give them further directions like use web storage vs cookies, or make it responsive on multiple platforms and use a custom stylesheet. I leave that up to them.

Some choose to do what we asked, and some do much more.

In the end, what matters is that we like the end result.

I don't say "I like movies, create me a nice movie website", or "How would you architect [IMDB](#) if you had to create it from scratch." I want the task to be simple enough that the engineer can provide a solution and challenging enough so they can use their skills to create something special.

RULE #4—Let them present their solution to a group on Monday.

The biggest issue I have seen with tech hires is that they can become very defensive over their solution. I will purposely (respectfully) challenge their solution to see how they react.

If they get defensive, it's an immediate no-go. Remember, there's a difference between defending which is good and being defensive which is bad. The difference is that the former is based on rational facts, the other is based on emotion.

A key aspect of this is that everyone in the group MUST come prepared, having looked through the solution.

RULE #5—Write the problem down for them to take home.

Be clear about what technologies, tools and look you're after, and the standards being judged. At the same time, leave the final solution open enough that the candidate can add their own flair. Let them

ask you any questions they have on Friday, and make sure to be available for their questions via email over the weekend. The goal is to create an environment of success (hopefully like you would for an employee).

RULE #6—Pay them immediately on Monday

Hire them or not, something about giving them a check after they present their solution is the best way to start or end a relationship.

To hire or not to hire that is the question

Indicators you should hire this person:

- During the meeting on Friday they asked a lot of clarifying questions
- The questions were thought out and made sure that nothing was misunderstood
- The solution addressed your problem using the technologies and techniques you prescribed
- They read the entire problem and followed the instructions correctly (i.e. if the problem said use PostgreSQL, they didn't give you queries that only work on Oracle DBs)

Indicators you shouldn't hire this person:

- They refuse to do the project because "someone will hire me without it"
- Didn't complete the project correctly
- They can't articulate their design/coding decisions and why they were made
- They get defensive when presenting their solution

In summary

Your mileage may vary, but I found this technique to work wonders for hiring talented tech talent. While my sample size may not be huge, I've yet to have it lead to a bad hire.

About me

I'm a polyglot developer deeply interested in high performance, scalability, software architecture and generally solving hard problems. You can [follow me on Medium](#) where I blog about software engineering, [follow me on Twitter](#) where I occasionally say interesting things, or check out [my contributions to the FOSS community on GitHub](#). ■

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

Interesting

2017



To-do lists are not the answer to getting things done

By SHANE PARRISH

I've [helped thousands of people](#)—from NFL coaches and Four Star Generals to best-selling authors and hedge fund managers—feel less busy and get more done.

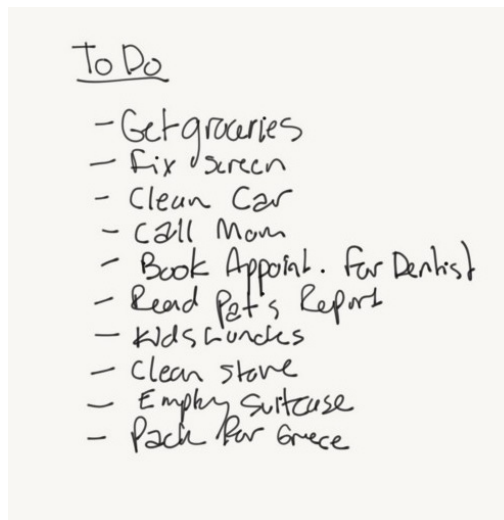
I get asked for advice on improving productivity so much so that I've packaged most of it into an online seminar that I call [Insanely More Productive](#).

Today, I want to share one of the biggest secrets I've discovered. It's the secret to some of the most productive people I know and it's non-intuitive.

Successful people don't make to-do lists

I know that sounds crazy but it's true.

Most of us who use a to-do list take out a sheet of paper and write out what we want to do, say on Saturday. It looks something like this.



That's horrible. And it keeps growing and growing because the rate at which people say "yes" to new things is not exceeded by the rate at which they get things done.

To-do lists make it easy and that's a bad thing

And because it's so simple to add things, these lists tend to grow and grow. Even worse they encourage our default to be "yes" instead of "no." Because it's easy to say yes, that's what we do.

It's no wonder we feel stressed and full of anxiety.

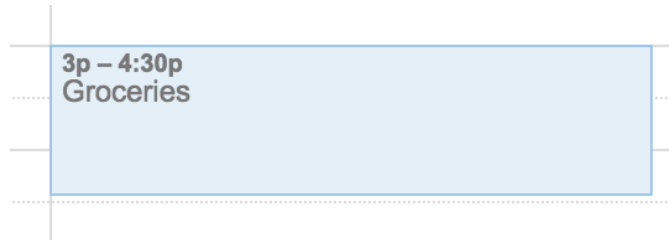
The real value in life comes from saying no.

To help you say no you need some friction. The solution to the to-do list problem is actually pretty simple. You have to make one change: schedule it.

If you need to pick up groceries, schedule it. Saturday morning at 10am. In your calendar, keep a list of the groceries you need under that block of time. *(That's how the most successful people use lists ...*

under a specific time block.)

Here is what my calendar entry for groceries looks like.



I block time in the afternoon for groceries, because I have a lot of mental energy in the mornings and I don't want to waste it on groceries.

And when you click on it, you get this:

The screenshot shows the Google Calendar event creation/editing interface. At the top, there are buttons for 'SAVE' (in red), 'Discard changes', 'Delete', and a 'More Actions' dropdown. Below this is the event title 'Groceries' in a text box. The date and time are set to '7/16/2016' from '3:00pm' to '4:30pm' on '7/16/2016', with a 'Time zone' link. There are checkboxes for 'All day' and 'Repeat...'. Below this is a section for 'Event details' with a 'Find a time' button. The 'Where' field is empty with a placeholder 'Enter a location'. The 'Video call' section has an 'Add video call' link. The 'Calendar' dropdown is set to 'Shane Parrish'. The 'Description' field contains the text 'To get: Ribeyes, Milk, Cheese, Avocado, Pine Nuts, Apples, Ingredients for meatballs.' To the right, there is an 'Add guests' section with a text box 'Enter guest email address' and an 'Add' button. Below that, the 'Guests can' section has checkboxes for 'modify event' (unchecked), 'invite others' (checked), and 'see guest list' (checked). The 'Attachment' section has an 'Add attachment' link. The 'Event color' section shows a row of color swatches with the first one (blue) selected. The 'Notifications' section has two rows: 'Email' at '10 minutes' and 'Notification' at '30 minutes', each with a dropdown arrow and a close button. Below this is an 'Add a notification' link. The 'Show me as' section has radio buttons for 'Available' and 'Busy' (selected). The 'Visibility' section has radio buttons for 'Calendar default' (selected), 'Public', and 'Private'. At the bottom, there is a note: 'By default this event will follow the sharing settings of this calendar: event details will be visible to anyone who can see details of other events in this calendar. Learn more' and a 'Publish event' link.

My grocery list

Want to spend a date night with your spouse or partner? Schedule it. And if you're smart you'll set this one to recurring weekly.

Want to set aside time for that project you never seem to get around to? Schedule it and you'll find yourself actually doing it instead of talking about it.

Why scheduling works so well

When you schedule things, you are forced to deal with the fact that there are only so many hours in a week. You're forced to make choices rather than add something to a never ending to-do list that only becomes a source of anxiety. And you can't just schedule important work and creative stuff. You need to schedule time for rest and recovery and mundane things like email.

Scheduling things also creates a visual feedback mechanism for how you actually spend your time—something we're intentionally blind to because we won't like what we see.

Being more productive isn't always about doing more, it's about being more conscious about what you work on and putting your energy into the two or three things that will really make a difference.

Of course this is only part of the solution. ■

My [productivity seminar](#) has helped students, teachers, hedge fund managers, CEOs, and even NFL coaches get more done and feel less busy. Maybe it can help you.

If you're interested in mastering the best of what other people have already figured out, join [Farnam Street](#).

Reprinted with permission of the original author. First appeared at medium.com/personal-growth.

You are not paid to write code

By TYLER TREAT

"Taco" by Karl-Martin Skontorp is licensed under [CC BY 2.0](#)

“**T**[aco Bell Programming](#)” is the idea that we can solve many of the problems we face as software engineers with clever reconfigurations of the same basic Unix tools. The name comes from the fact that every item on the menu at Taco Bell, a company which generates almost *\$2 billion* in revenue annually, is simply a different configuration of roughly eight ingredients.

Many people grumble or reject the notion of using proven tools or techniques. It’s boring. It requires investing time to learn at the expense of shipping code. It doesn’t do this one thing that we need it to do. It won’t work for us. For some reason—and I continue to be *completely baffled* by this—everyone sees their situation as a unique snowflake despite the fact that a million other people have probably done the same thing. It’s a weird form of tunnel vision, and I see it at every level in the organization. I catch myself doing it on occasion too. I think it’s just human nature.

I was able to come to terms with this once I internalized something a colleague once said: **you are not paid to write code**. You have *never* been paid to write code. In fact, code is a nasty byproduct of being a software engineer.

Every time you write code or introduce third-party services, you are introducing the possibility of failure into your system.

I think the idea of Taco Bell Programming can be generalized further and has broader implications based on what I see in industry. There are a lot of parallels to be drawn from *The Systems Bible* by John Gall, which provides valuable commentary on general systems theory. Gall’s Fundamental Theorem of Systems is that **new systems mean new problems**. I think the same can safely be said of code—more code, more problems. **Do it without a new system if you can.**

Systems are seductive and engineers in particular seem to have a [predisposition for them](#). They promise to do a job faster, better, and more easily than you could do it by yourself or with a less specialized system. But when you introduce a new system, you introduce new variables, new failure points, and new problems.


But if you set up a system, you are likely to find your time and effort now being consumed in the care and feeding of the system itself. New problems are created by its very presence. Once set up, it won’t go away, it grows and encroaches. It begins to do strange and wonderful things. Breaks down in ways you never thought possible. It kicks back, gets in the way, and opposes its own proper function. Your own perspective becomes distorted by being in the system. You become anxious and push on it to make it work. Eventually you come to believe that the misbegotten product it so grudgingly delivers is what you really wanted all the time. At that point encroachment has become complete. You have become absorbed. You are now a systems person.

The last systems principle we look at is one I find particularly poignant: **almost anything is easier to get into than out of**. When we introduce new systems, new tools, new lines of code, we’re with them for the long haul. It’s like a baby that doesn’t grow up.

We’re not paid to write code, we’re paid to add value (or reduce cost) to the business. Yet I often see people measuring their worth in code, in systems, in tools—all of the output that’s easy to measure. I see it come at the expense of attending meetings. I see it at the expense of supporting other teams. I see it at the expense of cross-training and personal/professional development. It’s like full-bore coding has become the norm and we’ve given up everything else.

Another area I see this manifest is with the siloing of responsibilities. Product, Platform, Infrastructure, Operations, DevOps, QA—whatever the silos, it’s created a sort of [responsibility lethargy](#). “I’m paid to write software, not tests” or “I’m paid to write features, not deploy and monitor them.” Things of that nature.

I think this is only addressed by stewarding a strong engineering culture and instilling the right values and expectations. For example, engineers should understand that they are not defined by their tools but rather the problems they solve and ultimately the value they add. But it's important to spell out that this goes beyond things like commits, PRs, and other vanity metrics. We should embrace the principles of systems theory and Taco Bell Programming. New systems or more code should be the last resort, not the first step. Further, we should embody what it really means to be an engineer rather than measuring raw output. You are not paid to write code. ■



What I learned from spending 3 months applying to jobs after a coding bootcamp

By FELIX FENG

```

7
8
9
10 function getDevJob(studying, hardWork, luck) {
11   var isPrepared = studying && hardWork && luck;
12   if (isPrepared) {
13     return true;
14   } else {
15     return false;
16   }
17 }
18
19

```

A less-talked about part of the bootcamper’s journey is what happens after you graduate—when you’re searching for that six-figure developer position.



< 3% of applications became offers

I completed Hack Reactor in July 2016 and took almost 3 months before accepting an offer with Radius Intelligence. I applied to 291 companies, did 32 phone screens, 16 technical screens, 13 coding challenges, 11 on-sites, and received 8 offers. The offers ranged from \$60-125k in salary from companies all over the US, and for both front end and full stack roles. In total, 2.8% of applications became offers.

Here are 5 things I wish I’d known before I began my job search.

Insight #1: Get through to real people

At first, I applied for companies using the shotgun approach. I applied through Indeed.com, Angellist, LinkedIn, StackOverflow, Hacker News, company websites, and even Craigslist.

I’d submit a resume for any role that wanted React, Node, or JavaScript experience. In the first week, I applied to 15–20 companies a day.

Pro-Tip: Find companies using this [easy-application repo](#).

My yield was low. Less than five percent of companies responded to me. I was throwing applications into a black hole.

Everything changed when one of my cohort-mates, a former recruiter, shared a guide to the job search. He told us to send emails directly to real people with each application. It could be anybody. As long as someone read it.

From then on, whenever I submitted an application, I searched for the company on LinkedIn and emailed someone on their engineering or hiring team.

For most small companies or C-level executives, the email format is usually `firstName@dreamCompany.com`. For larger companies, it may be `firstName.lastName@dreamCompany.com`.

To verify emails, I used [Rapportive](#) to cross-check emails with social media accounts.

The results were amazing. With 150+ emails sent, my response rate was a whopping 22%.

It also felt great to hear from real people. Surprisingly, CEOs and CTOs responded to me. Sometimes they even interviewed me themselves.

Takeaway: If you're applying through the front door, make sure you're getting to human beings.

Insight #2: Start small and work your way up

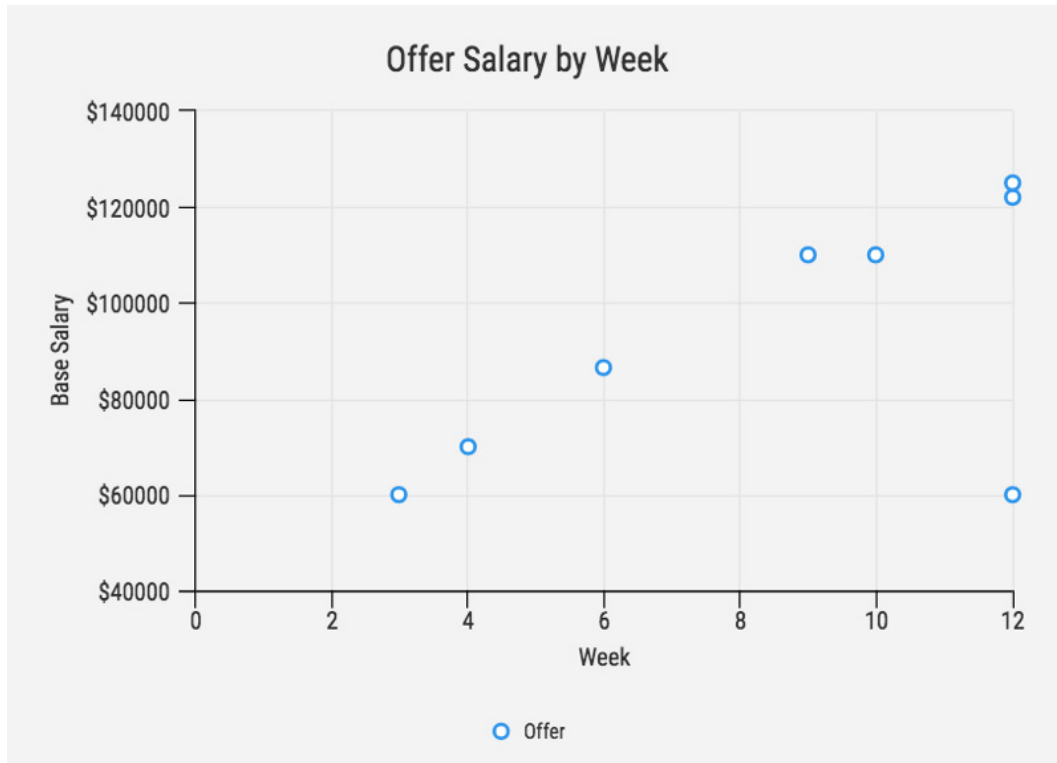
You will face Level 1 interviews (a non-tech company that needs any dev), where interviewers ask you nothing more than JavaScript trivia.

You will face Level 9 interviews (Google/Facebook level), where interviewers ask difficult data structure and algorithm questions.

I strategically set up my process so that I had lower-level interviews earlier, and higher-level interviews later on.

Early on, I gained experience, built confidence, and secured offers from companies that had less intensive interviews.

As I got more experience, I effectively “leveled up.” I became capable of completing interviews at companies with higher hiring bars. This is illustrated below as a linear correlation between the number of weeks I was into the process and the base salary I was offered.



There's a direct correlation between time spent interviewing and offer salary.

I unlocked tougher questions. I unlocked higher salaries. And eventually, I unlocked the job I took.

Takeaway: Plan to tackle easier interviews early on and more difficult ones later on.

Insight #3: Study like your future job depends on it (because it does)

I hate to break it to you, but the most important thing you could be doing at any point is studying and preparing.

Why? Because you won't get the offer if you don't have good answers to the questions they ask you.

People won't refer you if they don't think you're prepared for their interviews.

Coming out of Hack Reactor, my weaknesses were data structures and algorithms. [A study by Triple-byte](#) has found that bootcamp grads are weaker in these areas than computer science grads.

So I learned and practiced. Every day.

I devoted entire days to learning sorting algorithms. Other days, I focused on understanding how the internet worked.

If I didn't fully understand a concept, I'd spend the day watching YouTube videos or searching Stack-Overflow until I did.

I found the following study materials useful:

- [InterviewCake](#): My favorite resource for data structures and algorithms. It breaks down solutions into step-by-step chunks—a great alternative to Cracking the Code Interview (CTCI). My only gripe is that they don't have more problems!
- [HiredInTech's System Design Section](#): A great guide for system design interview questions.
- [Coderust](#): If you're avoiding CTCI like the plague, Coderust 2.0 may be perfect for you. For \$49, you get solutions in almost any programming language, with interactive diagrams.
- [Reddit's How to Prepare for Tech Interviews](#): I constantly used this as a benchmark for how prepared I was.
- [Front End Interview Questions](#): An *exhaustive* list of front-end questions.
- [Leetcode](#): The go-to resource for algorithm and data structure questions. You can filter by company, so for example, you could get all the questions that Uber or Google typically ask.

Takeaway: There's no such thing as too much preparation.

Insight #4: Put your best foot forward

Breaking into the industry is hard. You have to perform well, even when you're not fully prepared. In order to succeed, you have to be your own advocate.

Sell yourself

At Hack Reactor, we're trained to mask our inexperience. In our personal narratives, we purposely omit our bootcamp education.

Why? Otherwise, companies automatically categorize us into junior developer roles or tag us as "not enough experience."

In one interview with a startup, the interview immediately went south once they realized I'd done a bootcamp. One company used it against me and made me a \$60k offer, benchmarking against junior developers.

Ultimately, you need to convince companies that you can do the job.

At the same time, you need to convince *yourself* that you can do the job.

You can. Focus on your love for programming. Focus on what you've built with React and Node. Focus on demonstrating your deep knowledge in JavaScript and any other languages you've learned.

Only then can they justify giving you the job.

It's a two-way conversation

Interviewing is a mutual exploration of fit between an employee and an employer. While it's your job to convince employers to hire you, it's also their job to win you over.

Don't be ashamed of using the interview as an opportunity to evaluate the job opportunity.

I talked to any company, even if I had only the slightest interest.

I did on-sites all over the country with any company that invited me out. I asked questions, and sucked up knowledge on engineering team organization, technologies and tools used, company challenges, and system architecture.

Pro-Tip: During interviews, ask the following questions:

What are some technical challenges you've recently faced?

What do you enjoy about working at X company?

How are teams structured and how are tasks usually divided?

I treated every interaction as a learning opportunity. Each interaction helped me improve my presentation, interview, and technical skills. Each failure helped me find my blind spots.

Takeaway: Don't sell yourself short! And remember, it's a mutual exploration.

Insight #5: It's a marathon, not a sprint

The journey is by no means easy. For 3 months, I grinded 6 days a week. But I tried to take care of myself.

```
10
11 function aTypicalDay(hasOnSiteInterview, hasCodingChallenge) {
12   if (hasOnSiteInterview) { return crushInterview(); }
13
14   // Morning
15   drinkCoffee();
16   var applicationSubmitted = 0;
17   while (applicationSubmitted < 5) {
18     applicationSubmitted++;
19   }
20
21   // Afternoon
22   haveLunch();
23   var topics = ['dataStructures', 'systemDesign', 'javascript'];
24   for (var i = 0; i < topics.length; i++) {
25     study(topics[i]);
26   }
27
28   // Evening
29   haveDinnerWithFriend();
30   if (hasCodingChallenge) {
31     doCodingChallenge();
32   } else {
33     studyMore();
34   }
35   workOut();
36
37   return sleep();
38 }
```

What a typical day could look like in JavaScript

Some days, I'd study with friends. Other days, I'd go find a cafe and study alone, or hang out at Hack Reactor's alumni lounge. And every week I'd check in with our career counselor to talk about my progress.

It's easy to burn out during the process. Eat well, sleep, and exercise.

It can get lonely. Spend time with friends who are going through the same experience.

Takeaway: Prepare for the long game and make sure you take care of yourself.

In summary, the key takeaways are:

1. Get through to real people
2. Start small and work your way up
3. Study like your future job depends on it
4. Put your best foot forward
5. It's a marathon, not a sprint

The process may seem endless, but you're going to make it. Keep putting in the hours. Keep sending in the applications. Keep taking care of yourself. All of it pays off in the end. ■

A beginner's guide to Big O notation

By ROB BELL

A beginner's guide to Big O notation

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

Anyone who's read [Programming Pearls](#) or any other Computer Science books and doesn't have a grounding in Mathematics will have hit a wall when they reached chapters that mention $O(N \log N)$ or other seemingly crazy syntax. Hopefully this article will help you gain an understanding of the basics of Big O and Logarithms.

As a programmer first and a mathematician second (or maybe third or fourth) I found the best way to understand Big O thoroughly was to produce some examples in code. So, below are some common orders of growth along with descriptions and examples where possible.

$O(1)$

$O(1)$ describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
    return elements[0] == null;
}
```

$O(N)$

$O(N)$ describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(IList<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

$O(N^2)$

$O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.

```
bool ContainsDuplicates(IList<string> elements)
{
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
```

```

    {
        // Don't compare with self
        if (outer == inner) continue;

        if (elements[outer] == elements[inner]) return true;
    }

    return false;
}

```

O(2^N)

O(2^N) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2^N) function is exponential – starting off very shallow, then rising meteorically. An example of an O(2^N) function is the recursive calculation of Fibonacci numbers:

```

int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}

```

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

[Binary search](#) is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as **O(log N)**. The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

This article only covers the very basics of Big O and logarithms. For a more in-depth explanation take a look at their respective Wikipedia entries: [Big O Notation](#), [Logarithms](#). ■

A silhouette of a person holding three balloons is on the left, and a large, leafless tree is on the right. The background is a warm orange and yellow gradient, suggesting a sunset or sunrise.

Interesting

Happiness is a boring stack

By JASON KESTER

I spend way too much time on [Hacker News](#). It's a fun place, and a good way to keep up to date on all the new tech that us developer folk seem to need to know about. But it also leaves a fella feeling like he **really needs to keep up** with all this stuff. I mean, if you don't have a side project using the latest client-side framework, well, good luck ever finding a job again in this industry.

Thinking about it, I find that I straddle the line on this. As a long-time contractor, I try to stay up to date on the New Shiny and will happily run with whatever flavor of the month language, framework, and programming paradigm that a given gig wants. Yeah, sure, Node.js with tons of functional stuff mixed in pulling from a NoSQL store and React on the front end. I'm your guy. We're gonna change the world!

But for [my own stuff](#), there's no way I'd use any of that crap. Good old C#, SQL Server and a proper boring stack and tool set that I know won't just up and fall over on a Saturday morning and leave me debugging NPM dependencies all weekend instead of bouldering in the forest with the kids. This stuff is my proper income stream, and the most important thing is that it works. If that means I have to write a "for" loop and declare variables and risk 19 year old kids snooting down at my code, so be it.

I can't tell you how nice it is to have software in production on a boring stack. It gives you freedom to do other things.

I can (and often do) go entire months without touching the codebase of my main rent-paying products. It means I can, among other things, pick up a full-time development gig to sock away some extra runway, take off and go backpacking around the world, or better still, build yet another rent-paying product without having to spend a significant amount of time keeping the old stuff alive.

It seems like on a lot of stacks, keeping the server alive, patched and serving webpages is a part-time job in itself. In my world, that's Windows Update's job. Big New Releases come and go, but they're all 100% backwards compatible, so when you get around to upgrading it's just a few minutes of point and clicking with nothing broken.

I see it as analogous to **Compound Interest, but to productivity**. The less effort you need to spend on maintenance, the more pace you can keep going forward.

But yeah, the key is to never get so far down in to that comfy hole that you can't hop back into the present day when it's time to talk shop with the cool kids. Shine on, flavor of the week! ■

How to contribute to an open source project on GitHub

By DAVIDE COPPOLA

A step by step guide that will show you how to contribute to an open source project on GitHub, one of the most popular and used git repository hosting services.

GitHub is the home of many popular open source projects like Ruby on Rails, jQuery, Docker, Go and many others.

The way people (usually) contribute to an open source project on GitHub is using pull requests. A pull request is basically a patch which includes more information and allows members to discuss it on the website.

This tutorial will guide you through the whole process to generate a pull request for a project.

1. Choose the project you want to contribute to

If you decided to contribute to an open source project on GitHub it's probably because you've been using that project and you found a bug or had an idea for a new feature.

You can also [explore featured and trending projects](#) on GitHub or use the website search to find something in particular. When deciding to contribute to an open source project make sure to check it's still active otherwise your work might remain a pull request forever.

If you don't have a feature or bugfix in mind you can check out the *issues* section of a project to find open tasks. It's often useful to filter them using the labels created by the maintainers to find out available tasks not assigned to anyone yet.



Sometimes maintainers highlight easy tasks to encourage new contributors to join the project, like for example the one tagged "easy fix" in libgit2.

Before proceeding with the contribution you might want to check the (software) license of the project, just to make sure you are happy with its requirements.

2. Check out how to contribute

This is a very important step as it will avoid you (and the project maintainers) to waste a lot of time trying to help a project in a wrong way.

For example some popular projects like the Linux kernel and git use GitHub as a mirror, but they don't consider any contribution received on GitHub.

Once you are on the main page of the project you want to contribute to look for notes and files that explain how the maintainers expect you contribute to the project.

CODE_OF_CONDUCT.md	Introduce Contributor Covenant	9 months ago
CONTRIBUTING.md	CONTRIBUTING: document the optional tests	8 months ago
CONVENTIONS.md	CONVENTIONS: update to include general public API principles	8 months ago
COPYING	COPYING: include winhttp definition copyright	a year ago
PROJECTS.md	PROJECTS: consistently quote directories	5 days ago
README.md	README: adjust URL to libqgit2 repository	2 months ago

Often there's a dedicated file with detailed instruction called CONTRIBUTING.md, but sometimes you'll find notes in the README.md file which is displayed at the bottom of the page as well.

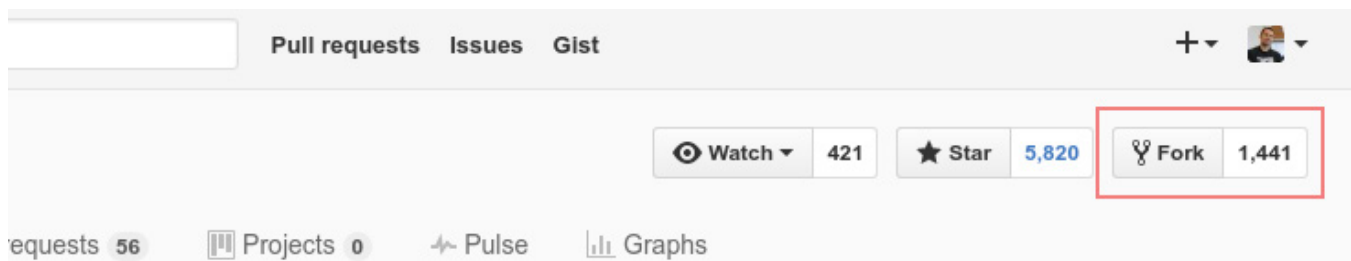
Before starting to work on your contribution, it's a good idea to check out existing issues and pull requests to be sure you're not going to do something which is already being done by someone else.

At this stage you might also open an issue to check if maintainers are interested in what you're going to work on.

3. Fork the project

Once you have established that the project accepts pull requests and that your feature/bugfix has not already been taken, it's time to fork the project.

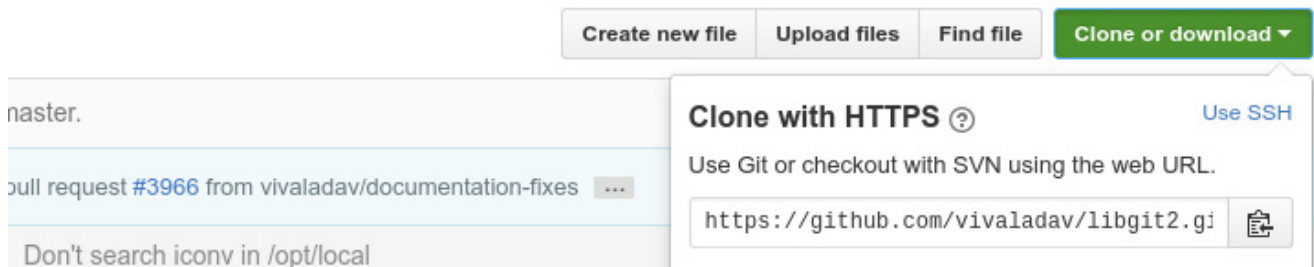
Forking the project creates a personal copy which will appear in your GitHub profile. to fork a project on GitHub simply click the Fork button on the top-right corner of a project page.



4. Clone the forked project

After you forked a project you need to clone it to have a copy on your machine you can work on.

To clone a forked project, go to the *repositories* section of your GitHub profile and open it. There you can click on the "clone or download" button to get the address to clone.



GitHub gives you 2 protocols to clone a project: HTTPS and SSH. For more details about which one to use check out their detailed [guide](#) on the topic. From now on let's assume you decided to use HTTPS.

Once you have copied an URL you can clone the project using a git client or git in your shell:

```
$ git clone https://github.com/YOUR_USERNAME/PROJECT.git
```

Cloning a project will create a directory on your disk which contains the project and all the files used by git to keep track of it.

5. Set up your cloned fork

Enter the cloned directory and add the URL of the original project to your local repository so that you will be able to pull changes from it:

```
$ git remote add upstream https://github.com/PROJECT_USERNAME/PROJECT.git
```

I used *upstream* as remote repository name because it's a convention for GitHub projects, but you can use any name you want.

Now listing the remote repositories will show something like:

```
$ git remote -v
origin https://github.com/YOUR_USERNAME/PROJECT.git (fetch)
origin https://github.com/YOUR_USERNAME/PROJECT.git (push)
upstream https://github.com/PROJECT_USERNAME/PROJECT.git (fetch)
upstream https://github.com/PROJECT_USERNAME/PROJECT.git (push)
```

6. Create a branch

Before starting to work on your feature or bugfix you need to create a local branch where to keep all your work. You can do that with the following git command:

```
$ git checkout -b BRANCH_NAME
```

This will create a new branch and will make it the active one in your local repository. Be sure to use a descriptive name for the branch name.

You can check you are in the right branch using git:

```
$ git branch
master
* BRANCH_NAME
```

The current active branch is the one with a * on the left.

7. Work on your contribution

Now it's time to work on the project. It's very important you keep this very specific and focused on a single feature or bugfix. Trying to squeeze multiple contributions in a single pull request means chaos because it makes it impossible to handle them separately.

While working on your contribution make sure to pull changes from *upstream* (the original project) frequently or at least before pushing your changes to *origin* (your fork). That will force you to fix any possible conflict before submitting your pull request to the project.


8. Create a pull request

Once you finished working on your contribution, it's time to push it to your forked repository on GitHub:

```
$ git push origin BRANCH_NAME
```

Now go back to your forked project on GitHub in your browser and you will find a new button at the top of the page to create a pull request:

Your recently pushed branches:

 **test_branch** (less than a minute ago)

 Compare & pull request

Click the button and you will get a new page which contains all the information on your pull request and where you can submit it to the original project.

Before finalising the pull request make sure to have checked everything is fine and to include as much information as possible to help the maintainers of the project understand what you have done and why.

9. Follow up

Hopefully some of the project maintainers will check your pull request and will give you feedback or notify you they decided to merge your changes soon. They might also ask you to change something or decide not to use your contribution. Anyway everything will be discussed on GitHub and you will receive notifications via email every time someone comments your pull request.

10. Clean up

After your contribution has been merged to the main project (or rejected) you can delete the branch you used for it.

To delete the branch in your local repository:

```
git branch -D BRANCH_NAME
```

To delete the branch on GitHub:

```
git push origin --delete BRANCH_NAME
```

Conclusion

I hope you enjoyed this tutorial explaining how to contribute to an open source project on GitHub. If you have any question, feel free to leave a comment. ■

A photograph of a server room. On the left, a person's arm and hand are visible, holding a yellow clipboard and pointing towards a server rack. The server racks are filled with various hardware components, including circuit boards and drives. The lighting is dim, with a blueish tint, and the server racks have a perforated metal front. The word "Opinion" is written in orange in the top left corner.

Opinion

It was never about ops

By MICHAEL BIVEN

"20110714-RD-LSC-0047" by U.S. Department of Agriculture is licensed under [CC BY 2.0](#)

For a while I've been thinking about Susan J. Fowler's [Ops Identity Crisis](#) post. Bits I agreed with and some I did not.

My original reaction to the post was pretty pragmatic. I had concerns (and still do) about adding more responsibilities onto software engineers (SWE). It's already fairly common to have them responsible for QA, database administration and security tasks but now ops issues are being considered as well.

I suspect there is an as of yet unmeasured negative impact to the productivity of teams that keep expanding the role of SWE. You end up deprioritizing the operations and systems related concerns, because new features and bug fixes will always win out when scheduling tasks.

Over time I've refined my reaction to this. The **traditional operations role is the hole** that you've been stuck in and **engineering is how you get out of that hole**. It's not that you don't need Ops teams or engineers anymore. It's simply that you're doing it wrong.

It was never solely about operations. There's always been an implied hint of manual effort for most Ops teams. We've seen a quick return from having SWE handle traditional ops tasks, but that doesn't mean that the role won't be needed anymore. Previously we've been able to add people to ops to continue to scale with the growth of the company, but those days are gone for most. What needs to change is how we approach the work and the skills needed to do the job.

When you're unable to scale to meet the demands you'll end up feeling stuck in a reactive and constantly interruptible mode of working. This can then make operations feel more like a burden rather than a benefit. This way of thinking is part of the reason why I think many of the tools and services created by ops teams are not thought of as actual products.

Ever since we got an API to interact with the public cloud and then later the private cloud we've been trying to redefine the role of ops. As the ecosystem of tools has grown and changed over time we've continued to refine that role. While thinking on the impact Fowler's post I know that I agree with her that the skills needed are different from they were eight years ago, but the need for the role hasn't decreased. Instead it's grown to match the growth of the products it has been supporting. This got me thinking about how I've been working during those eight years and looking back it's easy to see what worked and what didn't. These are the bits that worked for me.

First don't call it ops anymore. Sometimes names do matter. By continuing to use "Ops" in the team name or position we continue to hold onto that reactive mindset.

Make **scalability** your main priority and start building the products and services that become the figurative ladder to get you out of the hole you're in. I believe you can meet that by focusing on three things: **Reliability**, **Availability**, and **Serviceability**.

For anything to scale it first needs to be reliable and available if people are going to use it. To be able to meet the demands of the growth you're seeing the products need to be serviceable. You must be able to safely make changes to them in a controlled and observable fashion.

Every product built out of these efforts should be considered a first class citizen. The public facing services and the internal ones should be considered equals. If your main priority is scaling to meet the demands of your growth, then there should be no difference in how you design, build, maintain, or consider anything no matter where it is in the stack.

Focus on scaling your engineers and making them force multipliers for the company. There is a cognitive load placed on individuals, teams, and organizations for the work they do. Make sure to consider this in the same way we think of the load capacity of a system. At a time where we're starting to see

companies break their products into smaller more manageable chunks (microservices), we're close to doing the exact opposite for our people and turning the skills needed to do the work into one big monolith.

If you've ever experienced yourself or have seen any of your peers go through burnout what do you think is going to happen as we continue to pile on additional responsibilities?

The growth we're seeing is the result of businesses starting to run into web scale problems.

Web scale describes the tendency of modern sites – especially social ones – to grow at (far-) greater-than-linear rates. Tools that claim to be “web scale” are (I hope) claiming to handle rapid growth efficiently and not have bottlenecks that require rearchitecting at critical moments. The implication for “web scale” operations engineers is that we have to understand this non-linear network effect. Network effects have a profound effect on architecture, tool choice, system design, and especially capacity planning.

[Jacob Kaplan-Moss](#)

The real problem I think we've always been facing is making sure you have the people you need to do the job. Before we hit web scale issues we could usually keep up by having people pull all nighters, working through weekends or if you're lucky hiring more. The ability for hard work to make up for any short comings in planning or preparations simply can no longer keep up. The problem has never been with the technology or the challenges you're facing. It's always been about having the right people.

In short you can:

1. Expect services to grow at a non-linear rate.
2. To be able to keep up with this growth you'll need to scale both your people and your products.
3. Scale your people by giving them the time and space to focus on scaling your products.
4. Scale your products by focusing on Reliability, Availability, and Serviceability.

To think that new tools or services will be the only (or main) answer to the challenges you're facing brought on from growth is a mistake. You will always need people to understand what is happening and then design, implement, and maintain your solutions. These new tools and services should increase the impact of each member of your team, but it is a mistake to think it will replace a role. ■



food bit *

Sweet Preservation

Is there any other cake more divisive than the humble fruitcake?

Despite being a holiday staple on tables across the world, the fruitcake is a much mocked and loathed confection in America. But before you turn your nose up at the modest fruitcake, consider this: the fruitcake is virtually non-perishable.

In fact, the world's oldest fruitcake, a surprisingly un-moldy specimen that was baked in 1878, is still edible! (We said edible, not tasty.)

The science behind the immortal fruitcake is surprisingly simple: sugar acts like a powerful vacuum cleaner that sucks out water from the cells of food and microorganisms. Without water, microbes cannot survive and hence, the food doesn't rot. And fruitcake, with its copious amounts of candied fruit is basically one sweet water vacuum. And best of all, fruitcake can be also be soaked in rum and brandy to *ahem*, extend its shelf-life.

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

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

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