How I Became a Self-Taught Software Engineer at a Major Tech Company

…and how you can too.

[](https://code.likeagirl.io/@lexishanson?source=post_page-----c8d8e7bde704--------------------------------)

[Lexis Hanson](https://code.likeagirl.io/@lexishanson?source=post_page-----c8d8e7bde704--------------------------------)

[Follow](https://medium.com/m/signin?actionUrl=%2F_%2Fsubscribe%2Fuser%2Fe8be76253eb&operation=register&redirect=https%3A%2F%2Fcode.likeagirl.io%2Fthoughts-on-becoming-a-self-taught-software-engineer-c8d8e7bde704&source=-e8be76253eb-------------------------follow_byline-----------)

[May 31, 2018](https://code.likeagirl.io/thoughts-on-becoming-a-self-taught-software-engineer-c8d8e7bde704?source=post_page-----c8d8e7bde704--------------------------------) · 16 min read

Image for post



**Up until now I’ve been working in non-technical role. After studying for many months, I just got my first engineering job at a major Bay Area tech company. This journey has been rewarding but incredibly difficult. For any of you hoping to do the same, here’s how you can land your coveted software engineering job too.**

**A little background…**

**Why engineering?**

I’ve always been interested in logic, numbers, and critical thinking. I loved “whodunit” puzzles as a kid. I thought algebra was fascinating the first time I solved for 𝑥. While studying for my undergraduate degree in Finance, I built some pretty gnarly things in Excel using Visual Basic. That experience was my first exposure to how frustrating syntax errors could be, and how rewarding it is when something you write *finally* works! I took pride in thinking, “these hands built this!”

These past couple of years I felt that I wasn’t challenging myself enough in the work I was doing, and I really missed that feeling of building things. I previously felt that making huge career switch to software engineering was too far out of reach, but I started to reconsider. I decided to sign up for a [RailsBridge](http://www.railsbridge.org/) workshop in San Francisco, and though I didn’t know anything about Rails (or programming, quite frankly), I loved every minute of it.

Soon after, one of my software engineer friends pointed me to some of his favorite learning resources online. The spark was ignited.

I began by spending 20–25 hours per week studying and building things, all while working full time. I considered going the bootcamp route, but I didn’t want to quit my job at the time. I also wanted built-in test of “if I really like writing software, I’ll stay committed to this.”

When I finally got to the job application process, I got rejected several times. That was an emotional rollercoaster in its own right. However, I stuck with it and I ultimately had my choice of 3 really great engineering offers —one from a FAANG company, one from a startup, and one from the company I’ve worked for the past couple of years. I decided to accept the offer with my current company. I networked like hell before getting these offers and put myself out there more than ever.

Below are my insights on this whole process, with my own experiences interlaced, on how to build your learning plan and navigate networking in order to become a software engineer:

**1. Pick one programming language. Stick to it.**

This was likely the best advice I was given at the beginning of my learning journey. When I first started out, it was overwhelming. Did I want to do front-end, back-end, something in between? Which languages are best for which part of the stack? I spent a few weeks reading and doing tutorials in different languages to test the waters. I was wasting time and confusing myself.

The general advice is to find a project you want to do first and then use the language that best accomplishes the task at hand. I didn’t have a project in mind though (yet), so this wasn’t working for me.

I started out by spending a few months to get familiar with programming fundamentals (data structures, functions, for loops, etc.). Around this time, I had a chat with one of my engineer friends. I thought I needed to be expert in HTML and CSS before diving further into JavaScript. That sounds silly now, but I imagine it’s how a lot of people feel — you think you need X and Y in order to be good at Z. He promptly said that unless I’m specifically interested in HTML and CSS (I wasn’t), I didn’t need to learn them right then. **“Focus on one programming language, and get really good at it. Don’t look at libraries or frameworks yet. Just get really good at programming.”**

This advice was transformational and forced me to stay focused on getting extremely comfortable with vanilla JavaScript before diving into frameworks, which made that next step significantly easier.

I also made a concentrated effort to understand the tradeoffs between object-oriented programming (OOP) and functional programming. Functional programming resonated with me (you might have other preferences), and I challenged myself to approach problems from a “functional-first” perspective. In my opinion, this forced me to think more flexibly about programming and how functions and components should “compose” with one another.

**2. Get your time in. Be consistent.**

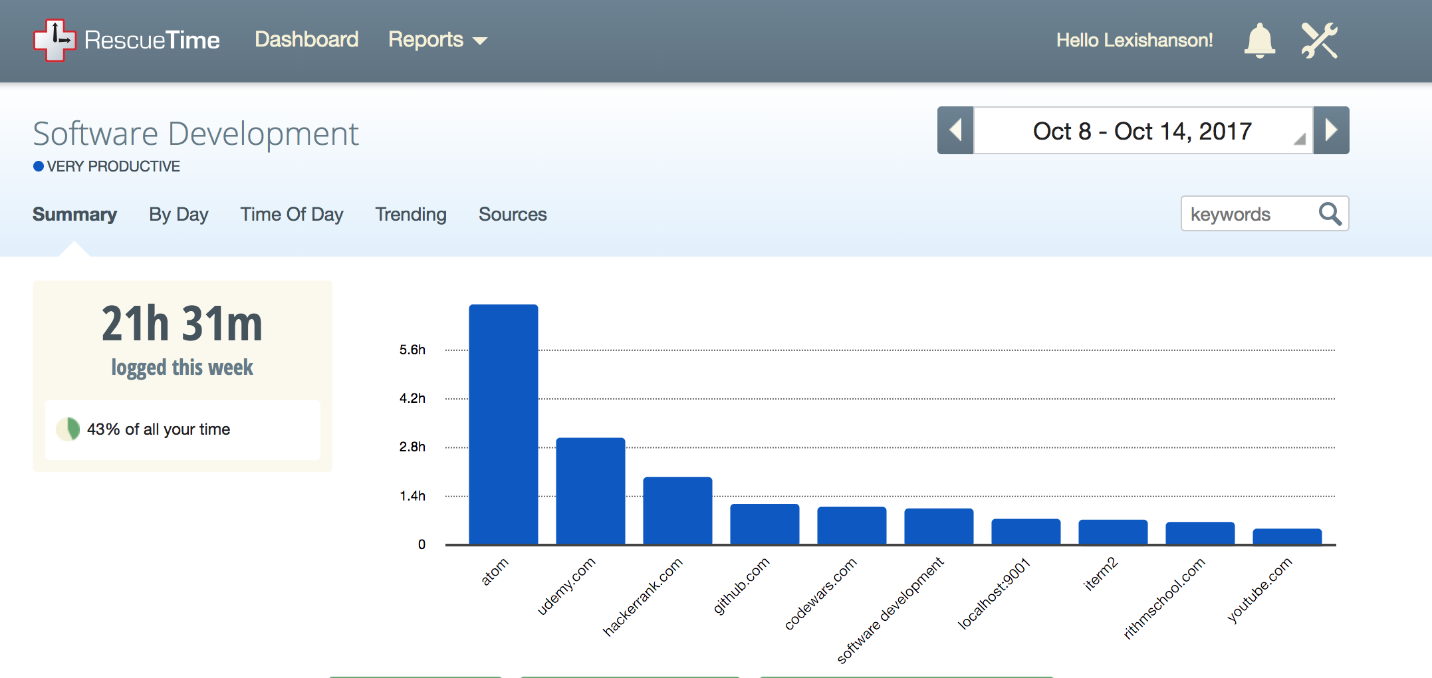
Second to information overload, the biggest reason I’ve seen people fall off track in their engineering learning journey is that they aren’t consistent with the time they’re spending on programming. You should be able to commit some time everyday (or at least 6 of 7 days). It’s an entirely new skill-set that people typically spend four years in college learning, just to get an entry-level understanding.

For some more tactical advice, think about your overall schedule and come up with a weekly number of hours you can commit to. I was working full time, but my goal was to do 20 hours minimum of programming per week. That could consist of anything — working through tutorials, reading CS books, reading documentation, building a project, etc.

I then broke that 20 hours/week down into more manageable chunks. I made sure to do 2.5 hours per day on average Monday through Saturday, and Sundays was my heavy day — 5 hours.

If you’re looking for a way to keep track of all of the time you’re spending, I recommend using RescueTime. It’s amazing! It runs in the background on your computer and automatically tracks and categorizes the websites and applications you’re using. Their free plan is all you need. I often overestimated how much time I was spending on engineering work, and RescueTime kept me honest.

Image for post



Wowee, look at those stats! I categorized Youtube as Software Development too— I was watching [Fun Fun Function](https://www.youtube.com/channel/UCO1cgjhGzsSYb1rsB4bFe4Q) and all of MPJ’s teachings on functional programming.

There wasn’t a ton of room for social stuff (sorry, friends!), but that was part of the sacrifice. I had to say “no” to lots of weekend trips, happy hours, and my weekly soccer practice, but I made sure my schedule was flexible. If I wanted to go to a meet-up on a week night, I would try to get ahead on my “hours” beforehand. I’d sometimes swap my Sunday “long day” to take a hike with friends and spend more hours programming on Saturday. Whenever I was on vacation, I went offline.

**3. Shortlist your best resources.**

Whether you’re just starting out, you’ve done a bootcamp, or you’re somewhere in between, there will be no shortage of people recommending resources to you. Information overload is evil — don’t let it creep in on you.

Quite a few people also recommended things to me like Codecademy and Project Euler. For whatever reason, those resources didn’t resonate with me. I felt like I was offending people if I didn’t take them up on their recommendations. I got over this quickly. There’s too much to learn to waste time on things that don’t “click” with your learning style.

Since I was focused on web development, I built my entire learning plan around vanilla JS, Node, and React, and studying algorithms and data structures.

Here are some of the resources I swore by, but you should find what works best for you:

* [**Treehouse**](https://teamtreehouse.com/)— Wonderful site with short video tutorials and hands-on challenges and quizzes in between to test your knowledge. It’s also game-ified which adds a lot of fun to the mix. Bonus: if you have a San Francisco Library card, [you get Treehouse for free](https://sfpl.org/?pg=2000883301)!
* [**Watch and Code**](https://watchandcode.com/) — [Gordon Zhu](https://medium.com/u/c513d629787d?source=post_page-----c8d8e7bde704--------------------------------) is a mastermind at teaching. In terms of understanding the depth ofexactly*how programming works*, I don’t think there is a better resource out there. Gordon uses JavaScript as a tool to develop an engineering mentality — his teachings apply to any language. He intentionally doesn’t bring in frameworks into his material — it’s noise at this stage. Also, check out his article on [**How to be Great at Asking Questions**](https://medium.com/@gordon_zhu/how-to-be-great-at-asking-questions-e37be04d0603)— this is an important skill.
* [**Udemy**](https://www.udemy.com/)— The courses you take will vary widely depending on your interests, but in the way of web development, I started with [JavaScript: Understanding the Weird Parts](https://www.udemy.com/understand-javascript/learn/v4/overview), [Modern React with Redux](https://www.udemy.com/react-redux/learn/v4/content), and [Node with React: Full Stack Web Development](https://www.udemy.com/node-with-react-fullstack-web-development/learn/v4/content). I’ve found the courses from [Stephen Grider](https://medium.com/u/d058882d8cd2?source=post_page-----c8d8e7bde704--------------------------------) and [Andrew Mead](https://medium.com/u/d02908cbd91e?source=post_page-----c8d8e7bde704--------------------------------) to be fantastic.
* [**Egghead.io**](https://egghead.io/)**—**the content here is slightly more advanced, but great for short videos on more specialized topics. [Dan Abramov](https://medium.com/u/a3a8af6addc1?source=post_page-----c8d8e7bde704--------------------------------) has a fabulous course on Redux.
* **Algorithm practice**—[**Hack Reactor Prep**](https://www.hackreactor.com/prep-programs/)**, [CodeWars](http://codewars.com/), [HackerRank](https://www.hackerrank.com/),**[**Cracking the Coding Interview**](https://www.amazon.com/Cracking-Coding-Interview-Programming-Questions/dp/0984782850)**(CTCI).**When I first started working on algorithms, I began with problems that were way too difficult for me. Hack Reactor Prep is a great launchpad for a ton of free, beginner problems before you jump into CTCI or even HackerRank “medium” problems.

**4. Don’t context switch, except when you should.**

Breaking focus is generally not great, although I’d argue there’s a time and place for context switching. If you’re spending more than an hour stuck on a coding problem, or even less than that and you’re hating what you’re doing, stop. Revisit it later. Taking a short break is often the right thing to do, but you might not have to step away completely— instead, try to switch to another task that’s still related to your learning.

When I would get bogged down and frustrated over a coding problem I couldn’t solve, I would try to switch over to something less cognitively-intense. I’d watch a Youtube video from a tech conference, or work on something I felt more excited about for a bit, such as an app I was trying to publish. It’s important to boost your confidence and find “small wins” to keep yourself going. There were times when I didn’t pause or context switch when I should have, and it was always a terrible decision.

One weekend, I couldn’t get an API request working the way I was expecting, and I sent myself into a vicious spiral for 8 hours trying (and failing) to figure it out. I became emotionally self-destructive. “If I can’t figure this ‘small thing’ out, how the hell can I expect to do more complicated stuff in a real job?” DON’T THINK LIKE THIS THIS. It’s bad, very bad! ([7 Most Important Mindsets That Will Set You Up For Long Term Success](https://medium.com/the-mission/7-most-important-mindsets-that-will-set-you-up-for-long-term-success-78a10de0281c))

As the situation escalated, I was eventually grappling with giving up on my engineering goals altogether. Of course, the next morning when I made another attempt with a clear head, the API issue I was working on was no trouble at all.

Often, when you do end up revisiting the thing that was previously frustrating you, the solution comes together like magic. When you’re feeling burnt out, do one of the following:

* Work on something less intensive (go for those small wins!)
* Work on something you’re more excited about
* Take a short break…you deserve it!

Use context switching wisely, when it will allow you to get in that extra bit of productivity and satisfaction. Find ways to get yourself unstuck and move another step in the right direction.

**5. Persist**

When the going gets tough, revisit items 1–4 above. When you’re really at your wit’s end — whether you’re dealing with a problem you can’t figure out, you’re feeling like you keep forgetting everything, or you don’t see any other option besides stopping and picking things up “a few months later” when you think things will somehow be different — don’t quit.

Image for post



Source: Rforce <http://rforce.com.au/news-004-6domains.html>

I can guarantee you will be overwhelmed at times, face emotional hurdles, rejection, imposter syndrome, and likely more. These things won’t disappear from showing up now and then, but it gets easier over time as you learn to deal with these hurdles (Check out [7 Most Important Mindsets That Will Set You Up For Long Term Success](https://medium.com/the-mission/7-most-important-mindsets-that-will-set-you-up-for-long-term-success-78a10de0281c)).

**6. Find others to learn with.**

It’s important to find accountability with others and to learn from people around you. Most of the learning I did was on my own, but the times I worked with other people were extremely helpful in accelerating me forward. I would recommend setting up a working group with a few other people to focus on something specific together, like an app or a project.

Once I felt comfortable with basic algorithms, I started practicing technical interview questions with two of my software engineer friends. We would meet weekly to practice algorithms and data structures. I was quite terrible in comparison to them, but they were super supportive and I learned from seeing their approaches to the problems we were working on. They were getting ready interview for new positions, so we all had an interest in working together. I’d usually do an easier problem, while they both did a more complicated one, first separately, and then they’d talked about each of their solutions when they were done.

This was my first exposure to seeing just how many ways a single problem could be solved. One of them would solve a problem with memoization, while the other used dynamic programming. One of them would use fancy ES6 JavaScript to solve a problem in two lines, while the other would add variable names and helper functions to make things more readable. We held each other accountable and pushed each other to keep tinkering at the problems until we found our way to solutions.

When you are accountable to someone or a group of people for doing what you said you would do, you can more easily get stuff done because you engage the power of social expectations. — [Thomas Oppong](https://medium.com/u/9bb9d25bfad2?source=post_page-----c8d8e7bde704--------------------------------)

If you’d like to set up a similar working group, connect with your engineer friends or hit up your cohort if you went to a bootcamp. You can meet in-person or virtually. If you don’t know other software engineers yet, now’s the time to meet them! Find a [Slack community](https://standuply.com/slack-groups) that suits you or start going to meet-ups.

Seek out quality meet-ups that relevant content you care about, or ones that have a group of attendees you’re interested in working/networking with.A couple of the meet-ups I frequented in San Francisco were [SFHTML5](https://www.meetup.com/sfhtml5/) and [Women’s JavaScript Study Group](https://www.meetup.com/Women-Who-Code-SF/events/250320253/). Most meet-ups cost anywhere between $10 — $15, and if you need help justifying the cost, they usually include dinner, beer, and wine !

**7. Get practical experience.**

One of the major reasons companies shy away from hiring junior developers is lack of practical experience. Proving that you can effectively collaborate on an engineering team is a critical component to your marketability. Hackathons are a great start, and finding open-source projects to contribute to is even better.

If you’re just getting started with open-source, you can look for projects that use the “[First Timers Welcome](https://github.com/MunGell/awesome-for-beginners)” label on open issues. You can also look at the companies you want to work for and see if they have any open-source projects. If so, doing a pull request or two is a great way to get their attention!

I eventually used the knowledge I’d accumulated to build an app I published for Google Assistant, [CryptoPrices](https://assistant.google.com/services/a/uid/00000020df028049?hl=en). The app turned out to be a great success — I learned a ton about building conversational interfaces and Google even acknowledged my user growth with a “Gaining Traction” award. I also started contributing to open-source projects and built my personal website in vanilla JS.

I was able to accelerate my experience by networking to find a project I could work on at my company. Nearly a year ago, our company had an internal meet-up focused on React. One of the speakers, an engineering manager, presented a project they were open-sourcing, and I was interested in contributing. I sent him an email after the meet-up to see if he’d chat with me over coffee about how I could get involved (go figure, I was too timid to talk to him the night of the meet-up).

He agreed to meet me and by the time we were done chatting he offered for me to come work with their team 1–2 days per week. I was so excited — this was my first opportunity to work on a project that was already out in the world with a team!

There was still a big hurdle: I had to ask my current managers to let me spend 20–40% of my week with another team. Thankfully, I had already been transparent to my managers about my career goals, so I pitched it to them like this:

*I’m going to become a software engineer, one way or another. Working with this other team is my biggest chance at success, and I want to be able to give you — my managers, all the credit in the world that you helped me get there.*

My pitch worked, and to my surprise at the time, they said yes. I then had to balance commitments of two very different jobs, but over the next eight months I worked with that engineering team two days per week.

**8. Reach out to your connections**

Once you feel ready to start interviewing, don’t waste your time by applying to every job blindly. Your highest chance of success lies in reaching out to people you’re connected to first. Reach out to your connections on LinkedIn or talk to engineer friends and see if they can refer you to their companies. Most tech companies offer nice referral bonuses, so they might be happy to submit your name!

Groups like [TechLadies](https://www.hiretechladies.com/)or [HireClub](https://www.facebook.com/groups/hireclub/) are great places to cultivate your connections. [TechLadies](https://www.hiretechladies.com/) is a community and job aggregator that “connects women with the best jobs and opportunities in tech.” Their [Facebook group](https://www.facebook.com/groups/techladies/) is a fantastic resource. Through [HireClub](https://www.facebook.com/groups/hireclub/), I met an engineering manager for a FAANG company. He posted a role in the Facebook group for a React-focused front-end engineer, so I replied. I ended up interviewing for the position and got an offer! The likelihood that this would’ve happened if I just went to the company website and applied is virtually zero.

**9. Reach out to everyone else.**

If there’s a company you want to work for but you don’t personally know anyone there, don’t let that stop you.

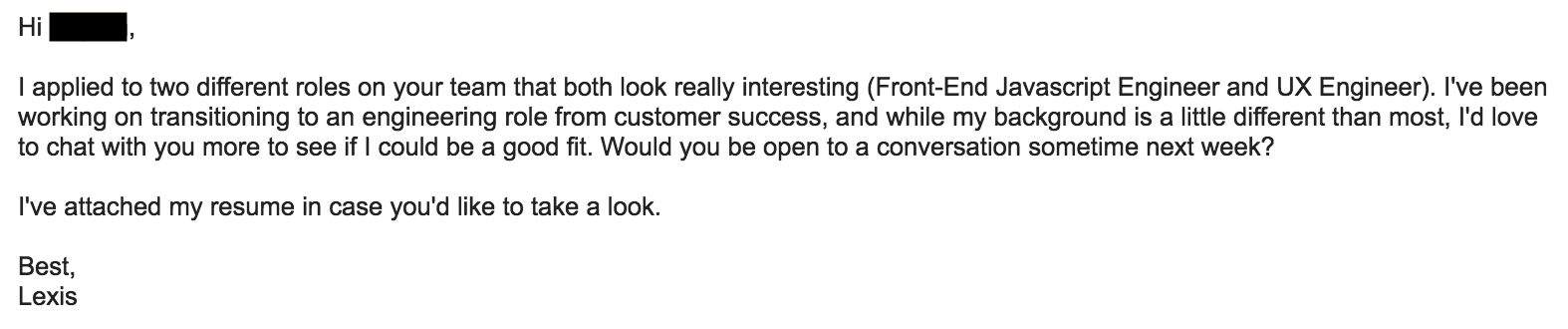
Look for people on LinkedIn or elsewhere that you can “reverse recruit.” Reverse recruiting is when you reach out to recruiters, managers, and individual contributors to make connections — good recruiters might not find you at this stage, so it’s up to you to find them.

**Include the following in your outreach messages:**

* Why you’re reaching out to said person — Sell yourself in 1–3 sentences.
* Personal connection (if possible) — “I saw you’re from a self-taught background. I am too — it’s inspiring to see your progression to [XYZ Company].”
* Ask — “Would you be willing to chat for 15 minutes next week?” / “Would you be open to referring me to [XYZ company] after learning a bit more about me?”

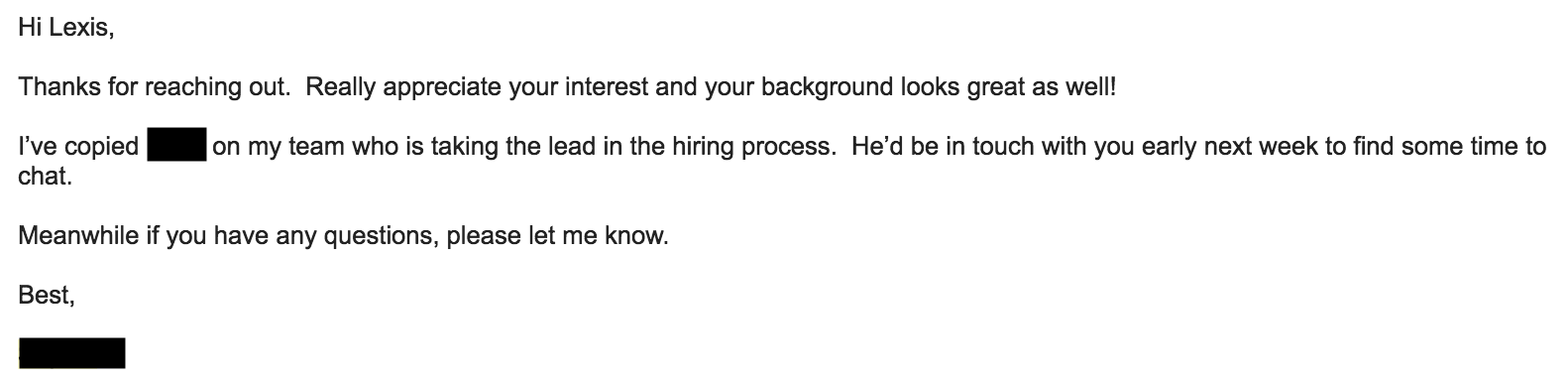
Here’s an example of an outreach email I sent, along with the reply:

Image for post



1 of 2: Outreach email I sent to an engineering manager (some content blocked for privacy)

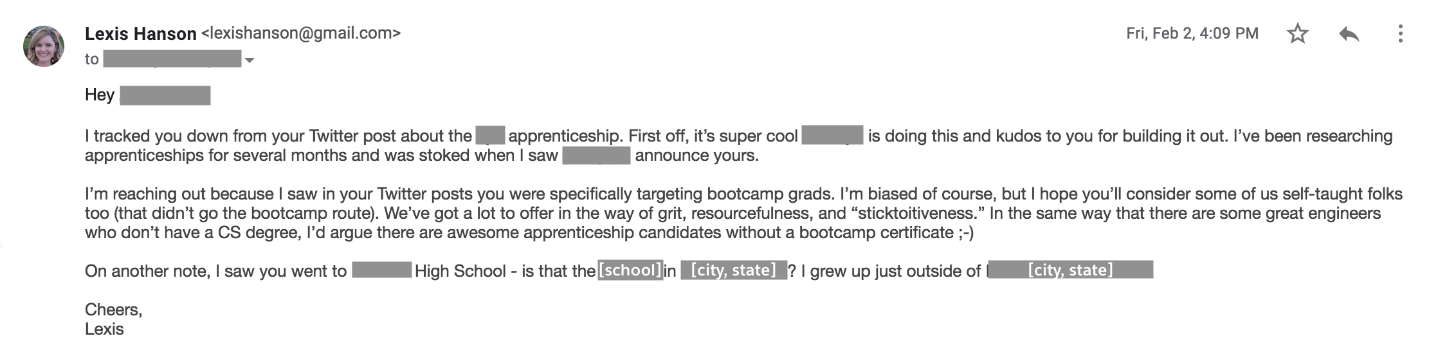
Image for post



2 of 2: Response (some content blocked for privacy)

Another example: I *really* wanted this software engineering apprenticeship role at one of my “dream” companies. After applying and hearing nothing for 2 weeks, I found the person on Twitter who started the apprenticeship. I (successfully) guessed at his email address and sent him a message. I spent an hour writing this message that was fewer than 10 sentences. It’s a delicate balance to sound like an excited go-getter and not outright desperate.

Image for post



(some content blocked for privacy)

He replied to my email almost immediately, on a Friday afternoon nonetheless. A week later, I was on-site, interviewing for the position. To my massive disappointment at the time, I didn’t receive an offer, but I was really close and knew then what I needed to improve on.

A good rule of thumb — spend 80%+ of your job-seeking time connecting with people for opportunities, and spend 20% of your time applying through typical job sites. Always see if you have a connection to a company before you apply cold.

**10. Apply, Apply, Apply**

As you’re making connections and applying to jobs, set a goal for this process too. Try to get at least 15 job applications submitted each week, with a stretch goal of 20. Use tools like [Trello](https://blog.trello.com/using-trello-for-a-job-search-less-stress-more-process) or [Jobtrack](http://jobtrack.io/), or just keep the status of your applications up-to-date in a spreadsheet.

Make sure to copy and save the job descriptions of the roles you apply to. I had an experience where a company took down their posting as soon as they started the interview process with me. I’m glad I had a back-up.

There are lots of other things to consider when writing your resume, interviewing, and negotiating job offers. Here are a few resources to start with:

* [How to Write a Great Resume for Software Engineers](https://medium.freecodecamp.org/how-to-write-a-great-resume-for-software-engineers-75d514dd8322)
* [Tech Interview Handbook](https://github.com/yangshun/tech-interview-handbook)

Also, never make a point to call yourself a “*junior* engineer.” Your resume history will show your experience level. There’s no reason to call attention to it again. Instead, focus your attention on your strengths and all the ways you’ll be able to help the company move forward with their roadmap.

The tech lead of the fellowship I was working on became a mentor of sorts to me, and he gave me a great nugget of wisdom. I was expressing disappointment that all the openings at our company seemed to be senior-level at the time. He encouraged me to apply for senior software engineer anyway, with an interesting insight:

Hiring managers will always want senior candidates to apply, even if they are open to junior or mid-level candidates. Senior-level candidates rarely apply to job postings that aren’t senior, because they don’t want to risk leveling down. In order to bring in the best pool of candidates, it makes sense to post roles at a senior level. In the end, if the team likes you, they’ll bring you on and place you in the level where you best fit.

**(bonus) Pay it forward**

Wherever you are in your career transition, you’ll inevitably run into others who are on a similar path. Try to learn from them, help them, or both. We all know how challenging making a career switch is, and it’s gratifying to know you helped another person get one step closer.

Thanks for reading! For a closer look at my timeline, check out the graphic below. Please comment with your thoughts, your own struggles, or any advice you‘d like to add. Cheers!

Image for post