YOUR FIRST YEAR
As a
SOFTWARE DEVELOPER

No BS Advice for New Devs

JONATHAN HOP

About the Author

In 2017 I made the leap from being a history teacher to a software engineer. I did not go through a coding bootcamp, I learned everything the hard way. Years later I earned the title of senior software engineer and coding boot camp instructor. I have taught 13 coding bootcamps and helped over 200 students gain the skills they needed to break into tech. I've taught bartenders, sommeliers, and machinists how to succeed and realize their dream of becoming developers.

Through my own journey and the journeys of others, I've distilled the lessons that will help you transition into a technical career—or rise if you're already in the field.

Some of what I say in this book might be blunt, but I find honesty gets you further than wishful thinking. If you're clear-eyed about your situation and act accordingly, you'll go far. Let's get to it.

Introduction

Landing a job is one thing. Thriving is another. This book is about the latter. It's possible to achieve your goal of moving up in your career with the right strategy and frame of mind. I've seen many junior developers fall into pitfalls and traps. You don't have to.

What no one tells you about your first year as a junior developer is that the industry requires skills in a variety of areas. If moving up was just about learning more JavaScript, this book wouldn't need to exist. The truth is as a developer we wear so many hats and move between different contexts. So many developers have the feeling of being overwhelmed and wondering if their careers and futures are secure.

No matter how you got here, bootcamp, self-taught, college, there's immense pressure to prove yourself in your first job. With the tech layoffs making headlines, that pressure feels heavier. This book gives you a framework to navigate the chaos, grow as a developer, and build a career that lasts.

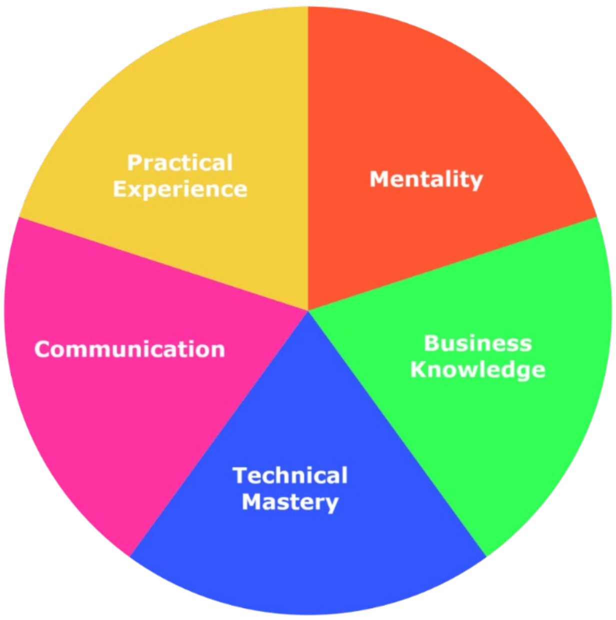
This book is short, but it contains many of the lessons I learned throughout my career. I focused those lessons on specific categories, and hopefully you can learn from my experiences and mistakes.

The Road to Improvement

If you want to move up in your career, don't focus on just one thing. Progress isn't linear. It's about leveling up across different skills. Think of it like a video game. Every project, every challenge, every mistake, it's all experience you gain. You don't just focus on one skill; you level up across multiple trees. It doesn't happen all at once, but brick by brick as you climb closer to your goal.

What skills matter? In my experience, there are five areas you must cultivate:

The Circle of Skills for Developers



Overview of the Five Mastery Skills

Rewire how you think about work. Adopt the mindset of someone who owns their success.

You must pick an area of specialty and dive into it, gaining a curiosity about how something works and mastering it.

Learn by doing. Theories matter, but knowing how to apply them in real-world projects is the most important.

You must understand how to effectively communicate with those who don't share your technical background, and work with those who do.

Software is supposed to serve a purpose, and if you do not understand the business side, then you're coding in the dark.

This book breaks down these five categories, why they're important, and how you can apply them to your work. Let's get started.

Mentality

Your Relationship with Your Job: Why You Need to See Work Differently

You got the job. Now what? Your mentality will dictate your success.

Your old job left you with baggage that won't serve you here. Maybe you took a coding bootcamp to escape a career you hated. You spent months dreaming about a better future in tech. That's human. But, if you bring the wrong mindset with you to work, you will struggle.

If you worked in a job you hated before becoming a developer, then you worked it because you had few options. Those types of jobs are not vehicles for growth.

If you graduated from a coding bootcamp then getting into tech has been your sole focus for months. You've dreamed of your first day on the job, fueled with images from social media feeds featuring bean bag chairs and colorful offices. The tech world must be Shangri-La.

The reality of your first year isn't going to match up with your imagination. Office politics, vague projects, and high workloads exist everywhere. Your boss and co-workers might be less helpful than you'd like. You will be on your own much of the time.

Let it go.

The mission of your company is to make a profit. You are there to assist them in that purpose; no more, no less. Your company's mission is not to nurture your development skills. The leadership of your company, your boss, your co-workers are the main characters of their own stories. You're a side character.

This is not cynicism, but clarity. The sooner you accept it the better, and the sooner you can focus on YOUR growth. You do not have to walk into work like it's an episode of Game of Thrones, but you need to put your head in the right space.

I Forced My Way on the Dev Team

I was happy to get my first job, but my initial position wasn't developer, it was for robotics. After a month or so I decided the development team was where I needed to be. I did not wait for someone to fulfill my needs; I took the situation into my own hands.

I pestered the folks on the development team for small tasks to accomplish. The kinds of tasks they didn't want to do. The more tasks I completed the more they saw me as a benefit to the team.

I ignored my assigned role and went after what I wanted. This was risky. My future as a developer was on the line, so I took the chance. It paid off.

That was not the end. On the new team, I was given projects that required skills beyond my ability. I did not have a CS degree or formal training.

I barely knew SQL and was asked to write stored procedures. My boss and co-workers left me to my own devices much of the time. I felt anxiety about whether I would succeed.

Many junior developers feel abandoned on their first job. Then fear creeps in.

"I'll be fired because I can't do the job."

"I knew this was going to happen."

"Why am I even here?"

Let it go.

Fear & The First Year

To move forward you must step forward. To overcome my fear of being fired, of being "found out" I said to myself:

So be it.

I did not have a CS degree nor a certificate from a bootcamp. I was on the team because I impressed my boss and proved I could solve problems. I was not the project manager, and I was not paid to ensure the success of the project. Don't shoulder someone else's responsibility. I was given my part, and I was going to use it to learn new skills and grow. Instead of thinking 'Oh god what is a SQL View?" I transformed the thought into "Today, let's learn what a SQL view is."

You will be shocked at your capacity to learn once you become **determined**.

I remember my first major project. It was a small app to allow the HR personnel to upload employee time sheets.

I stitched the project together through Google searches, trial and error, and raw stubbornness. I held my breath the first time the lady in HR clicked upload and felt pure joy when the graphs popped up on the screen as expected.

By the time I left that job a year and a half later, I wasn't the same person who had nervously asked for scraps of dev work. I had built real features. Solved real problems. I learned a tech stack on my own. That first year wasn't easy, but it made everything after it possible.

Technical Mastery

Knowledge For the Sake of Knowledge

“If I don’t have to use this on my job, it doesn’t matter.” This sentence is dangerous. It happens to be a common sentiment among my bootcamp students and society at large. We want our efforts to be effective. We don’t want to waste time on useless “theory.” If it doesn’t put food on the table, who cares?

Before I became a developer, I was a history teacher. I loved teaching. However, during the last year of teaching, the school did not pay us for two whole months, claiming they didn’t have our salary. I had enough. It was demeaning and I wanted a profession where I would be respected.

I have a friend who is a software engineer, and becoming like him was my goal. When I got home after quitting teaching, I stayed with him, and he taught me programming. It was intense. I studied six days a week, twelve hours a day.

My friend may not know it, but what impressed me about him was that he knew everything. When he taught me, it was always thorough. He was understandable but he didn’t pull punches, and he helped me understand just how deep software engineering was. He told me the point wasn’t to just learn to code, but to be good.

Even after I got my first job, he advised me. Whenever I needed help with a project at work, or was stuck on how to implement code, I’d call him up.

He would drop knowledge bombs and talk about problems from angles I had never considered.

He understood how computers worked under the hood. He was in a high position at work because of this kind of knowledge. I compared him to other senior developers I knew at work and realized that my friend attained his position because he could solve complex problems others couldn't.

Your job as a developer is to master your craft. Too many junior developers think that theory is too ivory tower. Not everything is of immediate use, but as a junior developer, you are not in a position to know what is or isn't going to be useful to you in the future.

Knowledge builds over time. Concepts that are too difficult or arcane as junior developers become relevant when we become seniors. Being able to solve the problems that no one else can is powerful.

The phrase "If my job won't require it, I don't need it" has a subtle backdrop of fear to it. After all, if you cannot meet the demands of your job, you'll be let go. Reframing is the key. Learning more about computers, understanding the framework and languages on a deeper level, will make you a more valuable employee at your current company and the next. After all, software development requires such a vast array of skills in different areas, leveling up is always going to pay off with patience.

At the start of your career, choose a language that will be your primary language. It doesn't matter if its Python or C#. You want to learn the ins and outs of how that language works. Learn a framework like .NET or Spring Boot and delve into it deeply. What does it do behind the scenes?

Learning is part of your job as a developer. It is stressful and it can feel overwhelming. You could spend years learning the details of a high-level programming language, let alone the other technologies that developers must use daily. It will require a bit of juggling, but the effort is an investment in your future.

Practical Experience

Learning By Doing

You can memorize every design pattern in the book, but if you can't work with real world code, it's useless. Practical experience is all about learning how to pick your battles. Where should your time be spent? What will be useful in the real world?

On my second job I worked at a larger company. My task? Add test coverage to a legacy codebase written when dinosaurs roamed the earth. My job was to modernize it and have it meet company standards for testing.

I opened the codebase and my stomach dropped.

"This is production code?"

I saw mounds of business logic in the controllers. I saw bulky if statements with no helper methods. Half the function names were hieroglyphs. Dependency Injection clearly wasn't on the creator's to-do list. I couldn't believe my eyes.

I was working at a larger company. In my mind, I was in the big leagues. I felt I needed to prove myself. I thought real developers wrote flawless, clean code. I obsessed over function names, reorganized the code, and tried to institute design patterns I read about in books.

In my head, this project was going to be the catalyst to show the world I was a real developer. My name, in lights!

The reality is this: **rewrites are messy**. In my zeal I didn't realize that I had introduced bugs into the code. I spent hours puzzling over failed tests like a twisted scavenger hunt. Refactoring one part of the code forced me to rewrite other parts I had no idea existed. Working with code other people wrote meant that I had to spend a good chunk of time following the thought process of someone that was no longer with the company.

It hit me: Am I working harder not smarter? I read phrases like "extensibility" and "future proof" but they were just words. Now, I was feeling what they meant.

The important question to ask was which piece of code had the greatest *impact*? I had to meet a certain level of code quality. But did that mean I had to put every line under a magnifying glass? The end user wants to take Thursday off. So long as the application did that, they wouldn't care.

I wanted to write "clean code", but spending 30 minutes on a block of code that no other dev would look at for years to come was a waste. Worse comes to worse, my predecessor could follow the flow and BAM, make the one line change they needed to make.

Practical experience is all about learning where your effort gets the most return. You have deadlines and a life outside of work. You must make trade-offs. You must identify the best places to focus your energy.

Every time I teach a bootcamp the students have their first major project. I tell them that projects are the arena for them to learn all the lessons that only come from experience.

We learn rules because those rules help guide us, but experience teaches why those rules are the way they are.

I finished the project. Parts of the code stayed messy. And that was alright. What mattered was ensuring it met testing standards. I tidied things up where I could and where it had an impact. Time was a precious resource, and I learned the importance of using it wisely.

Communication

Make People's Lives Easier – Not Just Writing Code

You are paid to write software that makes people's lives easier. Deep technical knowledge is great, but at the end of the day, users don't care about design patterns. They care that their expense form gets approved.

I once built an app that was supposed to be *the* catch-all system for the entire company. Every department would log in, get their work done, and move on. No more disconnected tools: just one unified workflow engine.

I got ambitious. Instead of writing software for a specific use case, I went fully abstract. I built a system of workflows, approvals, and transitions that could, in theory, handle *any* process. I used polymorphism, abstract classes, and even C#'s dynamic type. Every role was a generic concept: *Producer*, *Approver*, *Next Step*. Conditions, forks, backtracking—it all lived in this grand, flexible engine.

In hindsight? **Huge** mistake.

We launched. I had to be the face of the software, which meant standing in front of people and explaining how to use it. That was painful because I was *not* a good communicator. I did not realize that doing the presentation would make me the default tech support guy. If it didn't work the way they expected, my phone would ring.

I really didn’t take the time to see the situation from their perspective. I just wanted the presentation over with, for them to start using the software, and then I’d be on easy street. I could take on the next task and my flexible work engine would make it a snap.

Then, people started calling me with questions like:

"Why didn’t my travel request go to Person X?"

“Well, according to the HR data, Person X *is* your boss.”

I realized that the process was even more complex than I imagined. First off, I migrated the data from the old system. I didn’t realize it wasn’t accurate. Now the conundrum: if the travel request is already in the system, but it was sent to the wrong person, how do I change that? My almighty flexible work engine didn’t account for that.

What if there were rules like “If you travel to Mexico Bob approves, but if you’re heading to Italy then Joe approves.” My engine hadn’t taken that into account. And no wonder, I’m not a project manager, just a developer.

Then the real problems started.

Executives couldn't find the “Approve” button. I had to make it big and green because otherwise, it might as well have been invisible. Nobody uploaded supporting documents, so they just emailed them instead, completely defeating the purpose of the app. The org chart was a mess, leading to approvals going to the wrong people.

I didn’t take feedback well. It felt like a personal attack. After all, no one appreciated that I had used polymorphism and reflection to create a dynamic form of the strategy pattern to properly apply rules on the fly to certain stages of the workflow. They just cared more that the form didn't submit in Internet Explorer.

And worst of all? My "flexible" workflow engine wasn't actually flexible. Sure, it worked for travel requests *after weeks of tweaking*, but when I tried to adapt it for project management, there was so much unused code that I had no idea if I was introducing bugs.

The moral of the story is I didn't do enough testing with real people. For all future projects, I set aside time to put the software in front of as many eyes as I could. I needed their feedback.

I thought I was being clever by writing an abstract system. What I really needed to do was talk to users, figure out exactly what they needed, and build that—not some grand framework that solved problems nobody had.

The next time I wrote software, I skipped the engine entirely. Instead, I asked, "What do you need? What does the screen look like?" And I built *that.*

And guess what? It worked.

Business Knowledge

How the Sausage Gets Made

You may be noticing a theme. Software is always about the end user. And to satisfy their needs, beyond communication, you need to understand what your company does for a living.

Every company has a stated mission. Deliver high quality products. Satisfy the customer. Never be late on a promise. These are perfectly fine goals, but when you strip everything away, what does your company do that puts money in their coffers?

I was in charge of rewriting the application that handled customer service. I felt honored that my boss had enough faith in me to let me do the rewrite all by myself. I had taken the lessons I learned and worked closely with the customer service agents, even going so far as to sit on the floor with them to watch them do their job.

What hit me was I had huge holes in my knowledge of banking. What was an ACH? My only experience with mortgages was paying mine every month and as for retirement that amounted to having money deducted every week. Debits and credits, HELOC and index funds, I didn't understand these subjects beyond the superficial.

It made debugging and working with the customer service application difficult. Was the figure on their screen correct?

If I understood how depreciation works, I could’ve double checked.

I quickly realized that doing my job meant understanding the products the bank sold to its customers. I didn’t have to be an accountant, but I couldn’t get away with surface level understanding. I went home and started reading about mortgages and loans, and overnight my ability to solve problems with code changed.

This experience made my job pleasurable as well. When I worked on software I could speak in the same lingo as the person I was working with. This fostered confidence. I learned that part of the reason I had a hard time with the launch of the travel request system was that I never built trust.

This really showed in my next project where I automated expense deposits. People filled out forms for expenses, and the lady in accounting had to manually go through their forms and make sure the money fell into their account. I helped her automate the process, and it went smoothly because I took the time to learn what GL codes were and why the expense process was the way it was. She trusted me and trusted the software I built. She saw how it saved her hours upon hours of rote work every week. Not only that, she empathized with some of the things software engineers have to deal with.

“So, you mean to tell me some people don’t spell their own name right on their own expense report? Well, they’ll just have to wait for their money then.”

I realized at that moment I finally begun to do things the right way.

Final Thoughts

Being a software developer gets put on a pedestal. Social media is full of glossy videos—posh apartments overlooking the New York skyline, developers typing away in front of multi-monitor setups, snack bars, laundry services, unlimited PTO. It's an industry that attracts some of the smartest, hardest-working people in the world.

And honestly? I love this field. It's meritocratic—not perfectly, but more than most industries. Work hard, learn more, be effective, move up. I've met people from humble beginnings who climbed their way up through sheer persistence.

The salaries, the work-from-home privileges, the game rooms—they exist for a reason. This job is hard. Not just in terms of complexity, but because of the sheer scope of skills it requires. It's open-ended, constantly evolving, and mentally exhausting.

I've taught over 200 students how to become developers. They all start out starry-eyed with six-figure dreams. And then, somewhere along the way, they realize what this job really demands. Their brains must be rewired. They must embrace failure—not as a setback, but as the only way forward.

One student struggled from day one. When we ran our first program to print a simple greeting, they screamed because they thought the computer was talking to them. They constantly got lost, asked question after question, and had to fight for every bit of progress.

If you had asked me back then if they would make it as a developer, I would’ve politely said no.

But this person was relentless.

They reviewed every topic two hours before class. They took screenshots of my code to catch every syntax error. They made flashcards. They met with a tutor twice a week. They asked me questions at every turn.

Now?

That student works at Apple.

And that’s the truth about this industry: With enough grit, anyone can do this.

What’s Next?

If this book helped, then good—you’re already ahead of the game. The first year as a developer is tough, but you don’t have to figure it all out alone.

I send out real-world strategies, industry insights, and no-BS career advice straight to your inbox. No fluff, no filler—just lessons that help you **stay ahead and keep growing**.

A Quick Favor

What part of this book resonated with you the most? Just hit **reply** to the email where you got this book and let me know. I read every response.

Also, if you know someone who’s breaking into tech and could use this, feel free to **send them here**: Coding Your Career.

What’s Coming Next?

Keep an eye on your inbox for my weekly newsletter, packed with tips and advice on the world of development.

Until then, keep building, keep learning, and don’t let the chaos of this industry shake you. You’ve got this.