

SUCCEED IN SOFTWARE

Sean Cannon

Copyright © 2023 Sean Cannon.

All rights reserved. No portion of this book may be reproduced in any form without written permission from the publisher or author, except as permitted by U.S. copyright law. Cover designed by Sean Cannon.

ISBN-13: 979-8-9874976-0-9

This book is dedicated to the first person who ever inspired me.

The greatest actor of all time.

Jean-Claude Van Damme.

Acknowledgments

First and foremost, I want to express my deep gratitude to my wife, Abbey, for her unwavering support and encouragement throughout the writing process. She has always been my rock, and her love has inspired me to be the best version of myself.

I am also grateful to my children, Aiden and Maya, for reminding me of the importance of hard work and determination. They are the driving force behind everything I do, and I am so proud to be their father.

I would like to extend a special thank you to Eric Müller and Jason Monberg from Presence for their support and guidance in my technical journey. I am so fortunate to have had the opportunity to work with such a fantastic and welcoming agency.

I am also grateful to Brent Clark, Omar Abdelwahed, Jonatan Juárez, Jordan Baucke, Marco Scarlata, and Edward Philip for taking the time to proofread my book and provide valuable feedback and reviews. Your insights and suggestions have been invaluable.

I also want to thank Jack Firth for introducing me to Ramda and Eric Jordan for his ongoing inspiration in the web development realm. Their influence has been invaluable to my growth as a developer.

Finally, I want to express my heartfelt thanks to everyone who has supported me along the way. Your encouragement and belief in me have meant the world. Thank you.

Part One Traits	1
Chapter One What Makes A Senior Dev?	2
Chapter Two Experience	5
Chapter Three Intrinsic Values	12
Chapter Four Extrinsic Values	19
Chapter Five Jack of All Trades, Master of Some	23
Chapter Six Be Exceptional, Not Special	27
Chapter Seven Diversify Your Portfolio	32
Chapter Eight Proactive Mindset	35
Chapter Nine Becoming The Fixer	40
Chapter Ten When To Build It, When To Buy It	50
Chapter Eleven Failing Successfully	54
Chapter Twelve Build A Gut And Learn To Trust It	58
Chapter Thirteen Stop Multitasking	61
Chapter Fourteen Flow	66
Chapter Fifteen The Zen Road	71
Chapter Sixteen Cancerous Traits To Avoid	82
Part Two Practicals	90
Chapter Seventeen Comprehensive Analysis	91
Chapter Eighteen Security	97
Chapter Nineteen Privacy	108
Chapter Twenty Stability	111
Chapter Twenty-One Scalability	115
Chapter Twenty-Two Testing	127
Chapter Twenty-Three Documentation	140
Chapter Twenty-Four Numeronyms	150
Chapter Twenty-Five Clean Code	160
Chapter Twenty-Six Pure Programming	166
Chapter Twenty-Seven Functional Programming	170
Chapter Twenty-Eight Version Control	186
Chapter Twenty-Nine Data Persistence	197
Chapter Thirty Code Reviews	216
Chapter Thirty-One Architecture	228
Chapter Thirty-Two Technical Debt	242
Part Three Collaboration	248
Chapter Thirty-Three About The Collaboration Section	249
Chapter Thirty-Four Communication Mastery	251
Chapter Thirty-Five Pair Programming	270
Chapter Thirty-Six Mentoring	275

Chapter Thirty-Seven Conducting Interviews	284
Chapter Thirty-Eight Cross-Team Relationships	293
Chapter Thirty-Nine Vertical Relationships	311
Chapter Forty Developer Relationships	319
Part Four Growth Strategies	327
Chapter Forty-One Location Matters	328
Chapter Forty-Two Always Have A Mentor	333
Chapter Forty-Three Your Personal Brand	342
Chapter Forty-Four Goals Mastery	347
Chapter Forty-Five Compensation / Title Correlation	355
Chapter Forty-Six Continuing Education	362
Chapter Forty-Seven Interview Mastery	371
Chapter Forty-Eight Work For An Agency	383
Chapter Forty-Nine Sales	389
Chapter Fifty Networking	402
Chapter Fifty-One Fulfilling Your Dream	418

Foreword

I met Sean on a gray summer day in the SoMA District of San Francisco roughly a decade ago. I had moved to San Francisco without first landing a job, based on an adolescent over-confident attitude that my sheer will to succeed in software would overcome any obstacles.

At this point in San Francisco, the newly minted Unicorn class of software start-ups was riding high on the "Mobile First" explosion of software start-ups birthed by the introduction of the iPhone. Facebook's public debut, the seemingly overnight explosive growth of Uber, WeWork, or Airbnb, rapidly disrupted old-economy services with the new "Gig Economy." Oh, and "Bitcoin" was a thing, too - but it was just Bitcoin, not crypto, mentioned "Blockchain" outside of meet-up groups; the crowd was just as evangelical about the "wonders of the crypto religion" as they are today.

For me, San Francisco was what Hollywood is to aspiring actors. As soon as I moved to town, I figured I would audition for a role at a hot start-up company, and the glory and recognition would be all mine. Who cared if I wasn't practicing many of the valuable techniques Sean recommends in this book? Thinking back on myself at this point in my life and career, I felt I had all the answers when I had surprisingly few. Furthermore, I developed an unhealthy disdain for companies and individuals I saw as "corporate." I fancied myself an outsider and believed that successful software companies were full of virtue-signaling leaders whose successes were on the back of hard-working developers.

When I met Sean, I worked as a sub-contractor for Presence Product Group in San Francisco at our hip SoMa office, and it was replete with all the stereotypically "software start-up amenities" of the day: a ping-pong table, espresso bar, semi-regularly catered lunches, and micro-brew stocked fridge. Presence reminded me of some "Developer Commune" (very San Francisco) where folks came and went as they pleased, and all seemed to be working on a random menagerie of coding projects. At some point during the day, an announcement was

made, and a loose smattering of folks present shuffled into a small conference room to hear a presentation.

Sean was going to talk about a JavaScript library called "Ramda" (By the end of this book, you won't be surprised by that!) I don't remember much of what Sean said about Ramda, but I came away from the talk immediately impressed not only by Sean's passion for this technology but even more so by his passion for teaching about it. His enthusiasm for how this particular JavaScript library told me something about Sean (even though I couldn't describe it at the time).

Remember I came to San Francisco like an actor going to Hollywood to "make it"? Well, for me making it meant becoming a "rock-star developer." I naively assumed I just put in 12-15 hours a day coding, all day, every day, and I'd eventually find my way to the "rocket-ship" start-up on its way to becoming the next Zynga or Instagram! I didn't need to learn to interview for a job at Google or Facebook. What could they teach me I couldn't figure out for myself? I'd be respected and revered once I was part of an "acquisition" or "hot start-up."

As time went on and when one recognized my "natural coding talent" and "inhuman work ethic" (there's no such thing), I became like the actor who keeps going to auditions and never getting the part! Should I take classes? Get a coach or ready books to study my craft intensely? No. I decided: "Well, everyone doesn't understand me; I'm a one-man machine: I need to create my own company where I can just code my way to success- bend the world to my will!"

Instead of bending the world to my will, being the "Technical Co-Founder" of a venture-backed software start-up (as a first-time entrepreneur) almost broke me. Worked 100 hours this week? No one cares. Your code doesn't work, and you have a demo for a significant investor or client in an hour? No one cares. Do Your developers not understand the stories you wrote? No one cares. Or does your partner not agree about a feature? No one cares.

So what happens to the "Rock-star" who tours all the time? They get burned out. I did. I was physically and mentally exhausted; I was

overweight and emotionally drained. I had no close friends and no romantic prospects. I felt like I'd spent 4-years of my career coding my butt off for nothing.

I was so exhausted and vulnerable that I broke down and poured my heart out to an EQ and Mindfulness coach at our startup incubator like an alcoholic admitting recognizing they had hit rock bottom. I said: "I just don't know what I'm doing, but I'm in a bad place and need help."

Miraculously, she took me on as a client; she encouraged me to give up the things that were meaningful to me and leave behind the things I didn't need, to embrace my passions and lean into the things that I found most challenging and successful would follow. These weren't technical challenges either - they were emotional challenges. I needed to confront the stereotypes I had developed about what success in my career in software was all about. In a word, she set me down the path of "Mindfulness."

With time and self-reflection, I realized I had internalized a common emotion for many new developers: "imposter syndrome." And I hadn't just internalized it in my day job; I had internalized it in my entire life! And you know what else? I realized the only time I didn't feel like an imposter was when teaching and mentoring others.

I started to reflect on where I had learned this negative attitude. Did the "Fake it till you make it!", "Are you the rock-star, ninja, hacker-coder who can build an entire Uber for X app in a weekend" attitude come out of thin air? No. It's a "simple shortcut" that we all take when we see success: we believe that it was innate talent, extraordinary ability, workaholism, exploiting others, or luck. I slowly realized there was no logical basis for assuming others were born with exceptional ability! Have you ever seen someone start coding for the first time purely by intuition? Have you met someone that aced every interview or coding test they've ever had without experience? I've come to recognize that when I see or hear these sorts of anecdotes, the first thing I should be skeptical about is my belief system that wants to assume the "Simple Shortcut."

* * *

So what did all those "corporate" developers and entrepreneurs have that was so different? What helped these senior engineers, architects, and hackathon legends achieve success? In my opinion, it was simply experience and passionate mentors. And by mentors, I'm not just talking about a single individual or a teacher. These people all had colleagues, peers, websites, communities, books or papers, talks at conferences they heard or watched on YouTube that made them say to themselves: "Look at this person's success!" "Look at how trusted and knowledgeable they are!". I'd invite you to look deeper; chances are the most passionate ones will be the most authentic. Trust your intuition.

That brings me back to that talk about Ramda, way back before "We should teach Blockchain to pre-schoolers" and "WeWork is planning a moon base we'll be able to commute to an on-demand electric self-driving SpaceX rocket in 18 months". I realize now it's easy to feel like an imposter when you don't have a mentor or to be jealous of others' success when you haven't examined your belief system.

With Sean's talk, I knew there was "something there," even if I didn't recognize it then or wasn't ready to listen. Looking back, I have a better grasp on my intuition from then: Sean is a passionate mentor. I considered him a mentor, and I didn't even realize it! His passion (even more so than software development is suspect) is passing on his knowledge and experience.

Even if you don't have any mentors today, that's ok; you are reading this book. Even if half (or all) of the technical material goes right over your head the first time you read this book, that's ok. Please use the attitude and philosophy in these pages to build on that intuition to seek out the people, communities, books, and materials in your life who are passionate mentors. I'm confident you'll succeed.

Jordan Baucke*

Christmas Eve, 2022

* <https://www.linkedin.com/in/jordanbaucke/>

Preface

Hello reader!

My name is Sean Cannon. First and foremost, I'd like to thank you for putting your trust in me as a mentor and congratulate you on what I feel is the best investment you will make in your software career.

This book assumes you are either a junior or mid-level developer looking to jump 10-15 years ahead of your peers or are a senior dev who wants to start polishing your brand to shine above the rest and earn the income you deserve.

I spent the last two decades learning things the hard way through trial and error, luck, guidance, and mentorship. As I've grown and been given new opportunities, I've constantly been extending my hand behind me to help my fellow developers grow as well, so none of us are left behind.

Throughout this book, I reference some libraries, frameworks, and buzzwords which are mostly part of the JavaScript family. Please don't think that this book isn't for you if you code another language besides JavaScript. I absolutely work on projects that leverage other languages like Java, Python, Swift, PHP, Haskell, .NET... the list goes on.

In our line of work, we're often commissioned to add functionality to an existing codebase, and we don't get to choose the language. I decided to use JavaScript as the primary language for my examples and analogies because even if JS isn't your primary software language, you're probably familiar with it in some regard.

If you are brand new to JavaScript, you'll benefit in another way: I make a consistent point to mention that your **career** should not be your **syntax**. A programming language is just a tool on your belt, and you should be fluid and agnostic, using whichever tool is best for the project or the team.

So please understand that the concepts and principles I will cover in this book apply to *multiple* languages on *multiple* platforms. Most of my career and most of my commissioned work involves building software online for websites and cloud applications, so most of my examples and analogies will refer to that technology space, but please understand that I've also built mobile apps, kiosk applications, automotive center console experiences, and desktop applications. I assure you the principles work across all of the verticals. I'll do my best to mention some variances, like if something applies to statically or loosely-typed language only or a pattern that applies to functional programming and doesn't apply to object-oriented or procedural languages as much.

If you ever feel like a chapter should accommodate your situation better, please send me a message and let me know. I'll either try to help you personally or update the book in the next release to include additional content that covers your scenario if I think it can help multiple readers. You can also post in our Succeed In Software Facebook group^{*}.

Lastly, there is an accompanying instructional course for this book with several hours of video, which can be found at **SucceedInSoftware.com** if you prefer watching and listening versus reading. It also includes interviews and additional content that will be added regularly.

With that, once again, thank you, and welcome!

* <https://www.facebook.com/groups/succeedinsoftware>

Part One

Traits

CHAPTER ONE

What Makes A Senior Dev?

I want to rule out two common misconceptions right off the bat. The senior developer title has nothing to do with age, and it has nothing to do with skill. I've worked with developers twenty years older than me who decided to change careers late in life and start over. I've worked with developers twenty years younger than me that could pick up and apply new syntax remarkably fast and were always talking about bleeding-edge stuff none of us on the team had heard of yet. If a dev in either of these groups inquired about a promotion to a Senior Developer position, I would expect the engineering management to be looking for a much more specific set of qualities.

At a high level, a senior developer has to be ready to own accountability for the team and lean on experience to guide the team through architectural decisions and challenges quickly. The senior devs are the front line between product and engineering. We need to set realistic roadmap expectations and translate requirements from product and design into technical implementation tasks that the other developers on the team can easily understand, test, and objectively deliver.

In other words, the senior devs protect the junior devs from volatile product and design demands, and we protect the codebase from junior devs. As you will learn throughout this book, the skills required to reach the highest levels in the engineering vertical are much broader than simply being able to code well or being a stack expert.

* * *

Imagine for a moment that you are a sushi chef. How you learned the skill is irrelevant because you're pretty darn good at making sushi. You take orders from the waiter and make the sushi; rinse and repeat. You want to make more money and advance your career, but the problem is that you're stuck. There's no room to grow at your restaurant, and you can't buy the restaurant from the owner because you don't know anything about *running* a restaurant.

You can't easily get a job at a fancier restaurant because sushi is one of twenty items on their menu, and you're a specialist--sushi is all you know. So all you can do if you get tired of your job is move to another sushi restaurant, make sushi over there, and hope you're getting paid more. If your goal is to advance your career, more sushi is not the answer.

Now imagine that you put sushi aside for a minute. What else is on that fancy restaurant's menu? Pasta items like fettuccine, spaghetti, ravioli, linguini, barbecue ribs, steak, fondue, soups, pastries, and baked goods. These are all fantastic dishes and are making me hungry talking about them. Still, none of these allow you to leverage your sushi skills, so it may seem overwhelming even to consider applying for a position at that restaurant.

But what if you learned *just* enough to understand the underlying fundamentals? Fettuccine, spaghetti, ravioli, and linguini are all pasta; they're all essentially the same thing, prepared slightly differently, but they're the same at their core. You boil noodles and add some sauce; simple. So, instead of learning those four dishes as individual dishes, learn about *pasta* and keep your research at a very high level, like learning about the difference between the grill and the oven--frying versus baking versus boiling and which *types* of dishes need which *kind* of heat.

Now when you want to work at another restaurant, the items on the menu aren't nearly as important to you because you understand the fundamentals. Even better, you bring value to the restaurant in that they can rotate their menu more frequently because you can adapt. As you

gain more experience in this, you can hone your skills and become a master chef and come up with your own signature dishes to make a name for yourself. Once you understand what the people who receive your services consider valuable, it will open your eyes to growth opportunities all around you.

Let's bring this analogy back to software, so *pasta* becomes *ECMAScript*^{*}. Fettuccine, spaghetti, ravioli, and linguini become ES5, ES6, ActionScript and TypeScript. You can see how it doesn't make sense to try to master every language that adheres to the ECMAScript specification, regardless if the language is a deprecated syntax like ActionScript or a popular syntax like TypeScript. You don't have to know the ins and outs of each of these derivatives.

What *is* important is that you can quickly find your bearings in any of them and **fill in the gaps of what you don't know by leveraging what you do know**. Recognize the similarities in the syntax and when you lose your place, search the docs and get back to the code as quickly as possible.

Don't worry about understanding everything I mentioned at this point since we're just getting started in this book, but do understand that the overarching theme of this book will be **portability, scalability, and predictability**.

^{*} ECMAScript is a JavaScript standard intended to ensure the interoperability of web pages across different browsers

CHAPTER TWO

Experience

Experience is the most valuable trait you can grow, so I am starting this book with a bang. Before we get ahead of ourselves, let's ensure we're on the same page when I use the word experience.

Experience is simply the collection of previous *experiences* that can be recalled to make better decisions moving forward that yield *more* positive and *fewer* negative outcomes. In other words, **remember what works and what doesn't, so next time you save time.**

Notice here experience doesn't imply how skilled we are as a developer or how long we've been a developer. Have you worked with anybody who repeatedly made the same mistakes over and over, regardless of how long they have been a software professional? This is what we're trying to avoid. The good news is that we don't need to suffer through situations ourselves to learn from the outcomes personally. We can absolutely learn from someone *else's* experience and gain it as our own. The trick is to understand *why* something worked or didn't work, not just that it did or didn't.

How often we put ourselves in a particular situation determines how *deep* our experience will be--all the little subtleties and nuances that first-time participants may overlook, *we* will have in the back of our minds as the product team describes the feature they want us to build. The deeper the experience, the more comfortable we are in the situation

because we can recall memories from so many variables and will feel more prepared for all those possibilities.

For example, I ride a motorcycle. I learned how to recognize gaps in traffic and anticipate when someone would want to cut over into my lane from experiencing a distracted driver almost killing me by cutting me off a few years ago. I went down hard and totaled my bike, severely messing up my wrists to the point where it hurt even to write code for several months. Now when I ride, I can see that same situation coming from a mile away. I can feel the dangers of merging on ramps and assume that when someone hard-brakes, other drivers will likely cut over without doing a shoulder check, and it's up to me to stay out of their blind spot.

In contrast to being paranoid, experience can also build confidence. Using my motorcycle analogy, no amount of studying could help me feel comfortable taking hard corners in the rain the first few times. As I continued to ride and experienced it more and more, I learned to trust how my bike responded to the road--which surfaces were likely to be slippery and which weren't. From there, I could speed up a little more and learn when I wasn't really at risk of sliding off the road.

Experienced programmers are precisely the same. We know to put the emphasis on critical knowledge we've obtained from previous situations, so we don't repeat the same mistakes over and over.

In software development, the most valuable experiences will be associated with some form of emotion. Typically, either some great accomplishment after a lot of struggle or some significant failure after a lot of struggle. The more struggle, the stronger the experience imprint. We'll likely remember both of those situations and can recall them the next time our product team requests a feature that will require us to relive one of those experiences.

Companies typically pay more for senior engineers because the experience we carry with us can save time, reduce or even eliminate the research and development phase, and ensure any junior devs on the project get appropriate direction so the project can stay on course from

day one. This results in the company often paying *less* for the project, even though they paid *us* more.

Learning what doesn't work is often more valuable than knowing what does. What truly makes a senior developer useful to a project is being able to lean on these experiences and pitfalls from previous projects and offer genuine consulting. If a client asks us, "Which email provider do you recommend?" we can't simply say "Mailchimp" if the only email provider we've ever used is Mailchimp. We *can* do this with utility patterns but not for vendor or tool recommendations. It's just not an ethical thing to do, in my opinion.

This doesn't necessarily mean that we need to go out and try every email vendor, but it does mean one of the following should be true:

- We've used alternatives XYZ before and can offer a genuine recommendation.
- We did some comprehensive analysis research and maybe can't offer a technical opinion, but we can provide objective pricing and compatibility comparisons.
- We have colleagues in our network whom we trust and can relay their opinions of the alternatives.

Keep in mind that on this last bullet, do not consume their opinion as your own. Be honest and up-front. Say something like, "I've had success with Mailchimp, and I have trusted colleagues who've integrated Sendgrid and recommend it because of *blank*." We don't need hands-on experience to be good consultants. Still, we should have reputable and trustworthy sources to fill in the gaps, and we need to be honest and only claim expertise in things for which we want to be held accountable.

I recommend getting hands-on experience with as many tools as possible, so you get comfortable working with new tools to expedite your familiarity. The tools don't matter, but the time and place do. **I highly recommend using your own projects to try new things.** Don't use your client's projects as an excuse to propose and learn a new

library or tool you've wanted to try. If it happens, great, but it needs to be the client's choice.

One of the things that makes me valuable in our field is that I've been consistently coding for over twenty years versus migrating into a more managerial position where I'm out of touch with the implementation. I've had to part ways with a lot of technology I used every day simply because it lost market relevancy, but I'm able to keep those experiences with me and lean on them with new technology all the same. Nobody typically uses FTP, Flash, or SOAP APIs anymore on new projects, but having experienced a lot of pain points and wins with those tools; I immediately recognize when new tools try to go down a path that would recreate failure patterns.

One trick with experience I want to re-emphasize is that we don't directly have to live an experience ourselves to leverage it. We can use someone else's pain and learn from their experience. I've learned a handful of great tips from candidates when I interview them. I'll ask them for their most memorable pain point with a tool that made them hate the tool and their most notable win, where they figured out a problem after a lot of struggle. If I can recall *their* experience when it's needed on *my* project, it's valuable all the same. So with that, I want to share a story with you of a time I failed hard and learned one of my most valuable lessons so you can get an idea of gaining experience from someone else's mistake.

Very early in my career, when I was just out of college and a novice with PHP/MySQL, I opted to take on an E-Commerce website project all by myself. I had a designer friend handle the UI, but I was confident I could build the back end alone, even though I'd never done it before. This client was a parent of one of my friends, so I offered to do the site for next to nothing. She let me build it however I wanted, and two months later, I deployed the site, and we replaced her existing website with the new one. Everything worked great... or so I thought.

The client called me one day, freaking out that the website was charging other people's credit cards. She said that the previous customer was getting the bill when someone placed an order. I had no idea what had

happened. She abandoned the whole project and had me roll back to their old site. I refunded her money and studied my ass off with my tail between my legs.

Well, it turns out that being a database noob at the time, I had omitted a foreign key in the `orders` table, so it didn't point to the `customers` table at all. If you're new to database design, a foreign key means there's essentially a `column` in one table that references a `row` in another. The checkout process collected billing and shipping information, saved it, and showed a big "Confirm" button. So long as people went through that flow and clicked that button, it all worked fine--until one person decided to change their mind and abandon their cart.

The database then had, say, 50 customers and 49 orders. For a few days, everybody's order was charged to the previous customer's credit card. It was a **terribly** designed system. All I needed to have done was add a `customer_id` column in the `orders` table instead of assuming order 20 belonged to customer 20. **Essentially, I didn't know what I didn't know.**

So, as embarrassing as that mistake was, this is a happy story because for over twenty years now, I've been *completely* OCD with all my code, and I'm always the first to admit when I don't understand a tool in the stack. Had I not made that mistake, I'd hate to think how far I would have gotten before really screwing up on a big team with a big company. I carry that experience with me always--**it's so easy to mistake comfort for competence.** Never again.

Even if you don't fully understand the details of the experience I just shared, I want you to learn from this that **even the tiniest oversight can have disastrous effects on the project and our career.** Just because something seems to be working doesn't mean it is production ready.

Never trust your code, always test your code, and remember that "Those who fail to learn from the mistakes of their predecessors are destined to repeat them."

For your homework, I want you to start your own experience journal

and write down five of your own experiences and the lessons you learned. I've added a few of my personal experiences to give you an idea of how you could format your own journal. From here, populate your journal with experiences you never want to forget.

When you're finished, keep this document around and continually add to it as you work on more and more projects. You'll be amazed at how many small victories you will collect and how they can grow into a big pool of experience that you can use to justify your value on projects.

Remember, these don't have to be your own; they can absolutely be a colleague's, and that's the point. If you can tap into others' experiences when it matters, your career will grow substantially faster.

Software Experience Journal

Experience - We experienced an issue where our Redis cache was never expiring our session tokens, and users stayed logged in even though they were idle for a long time. This was hard to track down, and we spent several hours remotely monitoring the cache. It turns out Redis uses seconds for its TTL value, so while we were giving it 3600000 to represent one hour in milliseconds, Redis was interpreting that value as essentially 41.6 days in seconds. When we used 3600, it worked as expected.

Lesson - Not every service uses milliseconds, and not every type in a third-party service will align with the types you declare locally in your app. When troubleshooting, rule out the most obvious stuff - focus on verifying that our code intentions are being received as correct instructions before trying to reverse engineer stuff.

Experience - On a React project, we had some Recharts graphs on the page. The X and Y axes each had the same source data but different functions to transform it before giving it to the chart. The X-axis needed to be reversed for this particular chart, but whenever we did that, it would also reverse the data in the Y-axis. It turns out Lodash functions mutate objects instead of copying the data. They even disclose this on

their website^{*}.

Lesson - Always be aware that third-party libraries may introduce side effects. We replaced Lodash with Ramda, and the bug went away because Ramda functions clone all objects and arrays given to them.

Experience - We were getting CORS errors on an API call and spent a lot of time troubleshooting the CORS rules. It turned out the bug had nothing to do with CORS at all. In this case, the DevOps team had simply forgotten to create the Akamai route in production to match the Nginx route in the staging environment, so the request never made it to our Node layer, which contained all the CORS headers. Chrome never received the "Access-Control-Allow-Origin" header, so it was interpreted that we weren't allowing CORS.

Lesson - It turns out that most of the time, a CORS error is not a CORS header issue. It's just something with the OPTIONS call not returning. This is especially true if you've already set up CORS, and it's been working but stopped working.

^{*} <https://lodash.com/docs/4.17.15>

CHAPTER THREE

Intrinsic Values

Intrinsic values come from within and are self-motivated, rather than values that we might demonstrate if we succumb to peer pressure or to avoid reprimand.

The core intrinsic values I want to focus on in this chapter are **ethics**, **discipline**, **integrity**, and **respect**. As I cover these, I hope that by the end of this chapter, you'll say, "Yea, Sean, all this is super obvious." If not, then your journey will be much harder than it needs to be.

Let's dive in.

Ethics

As software engineers, the higher we get on the career ladder, the more we're entrusted with sensitive information. When we work on client projects or collect customer data on our own projects, it is our **duty** to treat this information with the utmost care.

I recently worked on a project where the client stored customer passwords in plaintext and had a database table full of contractor information, including addresses, birthdays, emails, and social security numbers. None of this information was encrypted and was stored along with all the other data in the same database. It was a scary sight to see.

If someone working on this project lacked ethics, they could easily export the data without anybody noticing. They could sell the data or use it for malicious intentions. The next thing you know, some poor person's identity is stolen, their credit score is ruined, and they have no way to track it back to this client or the hypothetical unethical developer on the project.

I've also worked on several projects where legacy client systems had extremely insecure authentication--multiple people using the same shared password or static FTP servers with no passwords. The client essentially had an open access system available to anybody in the inner circle, hoping nobody outside the circle would ever get the access details.

It's up to us as ethical developers to not only recommend security enhancements to our clients but also to never abuse the clients' trust and steal any of this information. It doesn't matter if the client is mean, abusive, or doesn't pay us--**there's no justification for ever stealing or exploiting sensitive data.**

While I fully expect everyone reading this book to have strong ethics, this information is still valuable to you so you can keep it on your radar that unethical devs *do* exist. We must be on the lookout for them and help go above and beyond to protect our client's sensitive information.

* * *

A much more common example of where ethics comes into play is when we sign a non-disclosure agreement (NDA) with a client. We'll have insider information that absolutely can not be used to gain a competitive advantage over our client or for personal gain like insider trading.

Discipline

The best definition of discipline I've heard is from Brian Tracy*, who defines it as "doing what you should do when you should do it, whether you feel like it or not." It's easy to do something when we feel like it; anybody can do that--doing the hard work when we *don't* feel like it is when we rise above the competition.

If you currently struggle with discipline, rest assured that it is a straightforward trait to develop, but you must *want* to develop it. The trick is to tackle the least desirable tasks first, every day, at every opportunity. When you can make a habit of doing the hard work before the easy work or even doing any work before enjoying your time, you will start to crave that feeling of accomplishment and become highly productive.

One trick to building your discipline is breaking up large tasks into small ones with measurable wins. You need to make sure your goals and tasks are measurable so that you can track your progress. When you're consistently making progress, it will drive that ambition, and you will want to make even more progress versus putting something off or suffering from that feeling of biting off more than you can chew.

Disciplined developers are reliable developers, and reliable developers get referred over and over and are offered management positions much earlier if that's something you're interested in.

* Brian Tracy is a Canadian-American motivational public speaker and self-development author. He is the author of over eighty books that have been translated into dozens of languages. His popular books are *Earn What You're Really Worth*, *Eat That Frog!*, *No Excuses!*

Integrity

Integrity can be defined as "doing the right thing, even when nobody's watching." If you toss some garbage at a waste bin and it falls onto the sidewalk instead of making it into the container, do you pick it up and put it in the bin? Even when nobody is around? When you're finished shopping, do you return your cart to the corral in the parking lot? If yes, then congratulations because you're way ahead of most people. This value will carry over to multiple areas of your career, and people will notice my friend.

For example, if I approve a pull request and it so happens that I missed some bad code and a bug was introduced, I own that and fix the bug. Even though I wasn't the author of the code that broke the app, I put my name on the code review, so it's just as much my fault as anybody else's. I'd rather be known for owning my mistakes and making things right than trying to hide my mistakes and pretending I'm infallible.

Integrity crosses over a bit with ethics and discipline. Doing the right thing, of course, includes not stealing data. Still, I chose to cover it separately because, by now, we're assuming you're an ethical developer, and you simply need to own up to situations where you weren't perfect.

Nobody is perfect, but if you can hold yourself to a higher standard than anybody else and not be afraid to hold yourself accountable for any actions you take which have a less-than-desirable outcome, you'll gain massive respect from your peers and management team. People will trust you more, believe you, and be more willing to pull you into inner circles because they know you won't talk behind their backs or share details of something you're not supposed to.

Respect

Respect is a broad and slightly vague value, but it's crucial. I've been on several projects in my career that I struggled to take seriously. The client's management teams didn't know what they were doing, or the client's dev teams were extremely junior and frustrating to work with--but one thing that helped me overcome my struggle is that I always remind myself that this is why I'm on the project!

I was hired because the junior dev team couldn't build the application. I was brought in to consult because the managers don't fully understand how all the infrastructure works. My frustrations typically come from the gap where I realize *I* know this, but I don't feel that my *client* knows. For example, if I'm consulting and offering advice and the client doesn't use it and then fails, I'll sometimes question why I'm even there in the first place. But there are ways to deal with this.

If we're ever in a situation on a project where we're working with someone and notice we're struggling to respect them as a peer, it's essential to shift our perspective and find things we *can* respect because once respect is gone, then all we are going to do is compromise our integrity.

We can't let the quality of our work suffer simply because we don't respect the person we deliver it to. If we can't respect the person, then we need to respect the position, respect the situation, respect the opportunity, or respect the experience. There is always something positive in these working situations, so we can't let ourselves get caught up in all the drama about who's contributing more, slacking off, or which team isn't doing their job and forcing us to do more work. If that happens, we can respect the chance we're being given to make ourselves look great as project saviors.

Unlike ethics, where a situation may present itself, and the ethical choice is blatantly apparent, respect is often something where we need to stop, step back, assess, and dial in with it to maintain optimism and healthy working relationships.

Summary

To summarize, before you try becoming a rockstar developer, focus first on being an ethical, integrous, disciplined, and respectful developer. These values define your character, and that character will help you succeed on *any* project using *any* stack.

CHAPTER FOUR

Extrinsic Values

Extrinsic values are the projected forms of our intrinsic values. They are perceived by our peers differently than how we perceive them. For example, integrity can be projected as honesty or trustworthiness.

Reliability

The first extrinsic value I want to cover is reliability because it encompasses all four of the intrinsic values we covered. Clients and employers rely on us to build their systems for them. They depend on us to do what is asked of us and not implement breaking changes. It doesn't matter if we're feeling lazy or prefer some other library that doesn't work well with the existing code base; if the client requires us to keep their business running while we work on stuff, then that's what we need to do. As someone who manages dev teams, reliable devs are *much* more valuable than rockstar devs. If you are a dev who can consistently deliver tickets on time that work as expected, the product team and engineering managers will want to put you in charge of other devs and give you promotions.

Let me be crystal clear: devs care about the quality of the code; clients care about features and profits. If the code isn't perfect, but the feature works, deliver it and then create a ticket to track any technical debt so the code can be cleaned up later. The code, of course, still has to work and not introduce any side effects or regressions. Still, if we need to meet a tight deadline, we need to implement shims like returning some static JSON in an API response instead of third-party service integration.

There are ways to improve reliability as a dev. Probably the most obvious way is to keep sprint commitments as light as possible, meaning don't say yes to owning a ticket if you're not 100% certain you can deliver it, or you risk both burning out and pissing off the product team.

Another way to improve reliability is to leverage iterative development versus waterfall development. Try to code every feature with a minimum viable product mindset so you can deliver the user experience before the final infrastructure. Eventually, as you get better, you can deliver your best code the first time, which won't introduce any technical debt because you'll learn to spot patterns early on in the grooming process.

Trustworthiness

Trustworthiness is more of an interpersonal relationship trait, but more often than not, we'll be exposed to things that one colleague doesn't want another colleague to know. Maybe a colleague tells us that they're looking for another job. We need to keep that information to ourselves because it's not our place to share it with anybody else.

The same goes for any corporate gossip where people share opinions of other team members and maybe their overall feelings about the project. Sometimes people need to vent, and why not? We all get stressed from time to time. We don't need to participate in the gossip to be good listeners, and our colleagues will genuinely appreciate someone they can confide in who won't go behind their backs with anything they tell us.

Trustworthiness also carries over to secure access, credentials, passwords, customer information, and other sensitive data in the system. We touched on this in the "Intrinsic Values" chapter when we talked about ethics, which is where this ties in.

That said, you can be the most ethical developer in the world, but if you're forgetful, careless, or clumsy, then SecOps is not going to trust you with production database access, for example. They're essentially extending their accountability to you and need to know that you're not going to set them up for failure by causing a data leak.

Altruism

Altruism is another way to say, "Put the well-being of your team before your own." I realize that might sound contradictory since you're reading a book on how to surpass your colleagues on the career ladder, but hear me out.

When we are on a team, we can't succeed unless our team does. We need to identify the weakest links on the team and help them succeed if we also want to. This becomes especially important when we are responsible for our team, for example, when we get placed into a leadership position like Principal Dev or Tech Lead.

When you are constantly helping your fellow devs solve problems and deliver code, you will become the dev everybody comes to for help. Your name will be mentioned repeatedly in meetings and reports, and you will gain a reputation for being an irreplaceable asset in the organization.

For example, on my daily scrum calls, it's extremely common for three to five devs to say, "Yesterday, I paired with Sean on ... whatever." If I can help get these devs over a hurdle or help unblock them so they can deliver their tickets, then not only do I feel good for helping a fellow dev, but the dev feels good for no longer being stuck. As a side-effect, my name is then mentioned more than anybody else's on the scrum call.

When it comes time for an executive to ask a product manager, "Who should we pull in from engineering to handle this?" there's an excellent chance they will pick me even if we've never met simply because they hear my name mentioned all the time.

You can immediately apply this strategy to your career. Don't practice altruism just to become popular, or it's not true altruism, and people will see your hidden motives. Help your fellow devs because that's what being a great member of the team does. Enjoy the perks later, but do it because it's the right thing to do.

CHAPTER FIVE

Jack of All Trades, Master of Some

Part of becoming a senior software engineer is learning to recognize opportunities to expand our skill set away from our core competencies so that we can consider more moving pieces when making technical decisions, delegating tasks, and so on.

For this, we use what's known as the 80/20 rule, which states that 80% of the yield is derived from 20% of the work, and 20% of the yield is derived from 80% of the work. This principle was created by Italian economist Vilfredo Pareto in 1897 when he observed that 80 percent of the land in England was owned by 20 percent of the population. He went on to observe the same thing in multiple other countries, and since then, we've seen that it carries over to practically any aspect of life where 20% of your effort will result in 80% of the results.

The trick, of course, is to do some comprehensive analysis so we can objectively determine what that 20% is and focus our energy on that.

Imagine you have a tool belt, and every skill that applies to your career is attached to it. Which skills do you keep in the front for easy access, and which do you keep in the back that you rarely use? If we follow the idea that 20% of your skills will account for 80% of your career growth potential, then we need to ensure those skills also account for 80% of your attention.

* * *

Consider these five languages from my tool belt, and let's pretend these are the only five that I know:

- XML
- ActionScript
- Perl
- JavaScript
- ColdFusion

One of these five languages, or 20% of this list, will likely account for 80% of my career growth potential; can you spot it? The answer, of course, is JavaScript, as the other four languages are a dying breed. So of this list, my "master of some" would be JavaScript mastery.

That doesn't mean I won't work on a project in the future that uses any of these other languages. We regularly get clients with legacy systems, and I need to know my way around them if we hope to migrate their applications to newer technologies. This is where the "jack of all trades" comes in.

Imagine I only coded JavaScript. I could still work on the projects I just mentioned, but I likely wouldn't be included in the kickoff meetings, and I wouldn't likely be working directly with the client. I'd probably only get tickets to write JS code for new stuff, which is fine, but **that's not career growth; that's just a career.**

Imagine we take that list of five skills and turn it into a list of 100. That implies that we should be highly competent at 20 skills and admitted novices for the remaining 80. For example, I haven't worked with Cold Fusion in over fifteen years, but I'm confident I could be assigned to a CF migration project tomorrow and get my bearings in an hour or two. That's generally where we want to be.

Our goal here should always be collecting skills to add to our tool belt, and these don't all have to be programming languages. For example, I'm highly competent with Git, and I use the CLI most of the time, but I've also used Subversion and Mercurial on previous projects, as well as straight-up FTP for projects that don't have any version control setup.

* * *

Similarly, I'm highly competent with GitHub, but I've also interfaced some with GitLab, BitBucket, SourceForge, GCS Repos, and AWS CodeCommit. I don't typically recommend those other services over GitHub for new projects because I find GitHub to be the most developer-friendly. Still, if I'm all-in on GitHub and neglect to even *speak* of those other services, I'm restricting myself as a consultant. In contrast, **being able to talk about a variety of tools and services significantly helps me build rapport with prospective clients and employers.**

So what are some examples of less-obvious skills we need to have on our tool belt?

What about something like written communication? Being able to convey intent in a written message without wasting people's time, causing confusion, or stirring up emotions is not an easy skill, and it's a fantastic skill to have on our tool belt.

Have you ever read an email or chat message from someone on your team that made you question that person's competence? Or maybe it left an impression on you that made you want to avoid working with that person? Perhaps they sound condescending or judgemental or are always complaining about something. People tend to remember how they feel other people treat them, so emails with an unfriendly tone can have lasting effects.

Another example of a less obvious skill could be working with other teams through an issue-tracking system. It doesn't matter if the tool is Trello, Pivotal, Jira, or GitHub Issues--the same skills apply. When I'm done working on a ticket, I like to attach evidence that I verified the acceptance criteria on my end. Sometimes I attach a screenshot or video capture, or sometimes I'll paste an API response or maybe a link to a URL that proves an endpoint is working--just anything that shows I did my due diligence and didn't just toss the ticket over the fence to the product or QA team.

Not only does this help the people on those teams do their job, but they also appreciate my effort. If they experience a result different from the

attached proof, then instead of a divergent perspectives scenario, they'll usually try to troubleshoot on their end for much longer before kicking the ticket back to me. Chances are the issue is on their end, and the ticket is fine.

Hopefully, you can see how much we can benefit from practicing a wide variety of skills outside our comfort zone. This 80/20 principle will follow you throughout your career, so try to get ahead of it whenever you can to use it to your advantage.

When you get to a leadership position, you'll likely observe that 20% of your dev team will do 80% of the work, and of course, the remaining 80% of the dev team will do 20% of the work. It will be up to you as a manager to recognize those high performers and pair them up with low performers to help balance the team's velocity. **As fun as it is to have rock stars on the team, as engineering managers, we need to recognize that devs come and go, and the team's competence and velocity need to survive staff transitions.**

I mention this because as you start applying these learnings to your career and becoming a high performer, you will be in that 20% bucket delivering 80% of the team's output.

Initially, you might feel like you're being punished for having to pair up with devs that don't share that drive and enthusiasm, but know that you're starting to become a mentor, and it's precisely where you want to be.

CHAPTER SIX

Be Exceptional, Not Special

When devs have been working on a project for some time, we naturally become comfortable with both the stack and the process. The ability to predict and anticipate our surroundings can allow our subconscious to take over and free up our frontal lobe to deal with more important things like solving complex problems.

If we are constantly being distracted by process changes, or if we keep having to check the docs for syntax on a new library, it isn't easy to get into flow and can noticeably take a toll on our mental health.

What often happens is that devs get used to maintaining a certain velocity and mental state. It can feel like starting over when we decide to transition to another project, maybe at another company. Velocity is at risk of significantly dropping if too many aspects of our day-to-day change simultaneously.

It's similar to driving in the city the first time if you live in the suburbs. Most people need to turn down the radio and consciously pay attention to the street names and whatnot because it takes much more mental effort to accomplish the same task they were doing for the past several minutes *getting* to the city--the only difference is the new environment.

So all this said, it can be tempting when you start a new project to make suggestions to the team that redirects them back to the stack and

process from your previous project. Suppose you used Pivotal Tracker in your former position, and the new project uses Jira. In that case, it makes sense to hint to the product team that Pivotal is superior to Jira when they're both just tools and accomplish fundamentally the same thing.

Another example may be a library. In JavaScript, if you're very comfortable with React but the new project is using Vue, you may want to complain about it and talk down Vue and talk up React.

The problem with the two examples I mentioned is that they're **parallel changes**. Imagine you did convince the product team to switch to Jira or the engineering department to switch to React. In reality, the teams are no better off than they were before--most likely, they're *worse* off, and the only person who is happy about it is you because now you have a knowledge advantage and can look like a rock star.

This mentality is typical and can be as insignificant as arguing coding style preferences like underscores are better than camelCase, four-space is better than two-space indentation, or tabs are better than spaces. At the end of the day, if we're joining an existing project with an established stack and process, we want to **let go of our personal preferences from our previous experiences and embrace the opportunity to add new tools to our belt**.

It's ok that we won't be great at these tools. It's surprisingly easy to ramp up on new tools the more we switch them. It's almost comical how often these libraries copy each other and only offer a few minor improvements over another library but fall short in other areas.

One way to expedite this process is constantly experimenting with new tools in personal side projects. Don't go crazy and build huge applications, just do some "Hello World" stuff here and there and get your bearings and you'll be fine. I recommend you choose a constant app, like a tic-tac-toe app, and rebuild it over and over in various frameworks and languages. **If you can achieve the same deliverable regardless of the tools used, you'll discover a new appreciation for how insignificant the tool choice is on any project.**

* * *

A special dev says, "At my old job, we did it this way." An exceptional dev says, "However you want to do it is fine. I've found success with a few ways, and I've found that a few other ways caused some headaches we may want to avoid."

When we can present options and show that we're bringing an unbiased experience to the table and our goal is to help the project and team and not simply make our job easier, our credibility goes through the roof. People come to us more and more for our advice on things because they know we're not going to give them an empty sales pitch, like "We should swap out Underscore with Lodash."

For context, the JavaScript utility library Lodash claims to be better than its predecessor Underscore.js because it's so much faster. In my experience, who cares? Hardware is cheap, and if we're to the point of micro-optimizing our app, then we're basically done with the app, and by that time, I'm transitioning off the project and starting another one. Any junior dev can swap out Underscore with Lodash--it doesn't need to be you or me.

That said, a functional library like Ramda^{*} is significantly better than Lodash. Lodash admits several times on their docs that their functions mutate data[†]. Mutation is terrible--it causes side effects and confuses everybody working on the project. Bugs caused by mutation are extremely hard to reproduce and track down, and the only reason libraries do it is to save a few kilobytes of memory and a few milliseconds of processing.

Ramda clones objects and arrays, so there are virtually no side effects. The trade-off is that it's a functional library, so it has a slight learning curve, and I typically don't recommend it to active projects with an established process. Instead, I'll mention the risks of mutation and point out that Lodash mutates, so we should be cautious and conscious of that risk. If it becomes a problem, I can recommend alternative libraries that

^{*} <https://ramdajs.com>

[†] <https://lodash.com/docs/4.17.15>

will eliminate the risk entirely.

I wanted to write this chapter because I see this happen all the time, and I don't think anybody is ever really aware that they're doing it. I'm guilty myself; I did this for years for things that I look back on, and I am embarrassed that I thought they were so important that I felt I needed to put my foot down. I can think of a hundred things that I used to believe firmly were "right" until I tried other ways and realized **it wasn't the syntax that I cared about keeping but rather the feeling of not having so much to think about.**

The good news is that the more we change things up and adapt to whatever environment we're thrown into, the more we can code in flow just by using our peripherals and matching the surrounding syntax. Tools like a linter or static analysis can assist us by formalizing the syntax and offering genuine consistency in the files so our eyes can scan more quickly.

The underlying takeaway here is that we don't want to make our employer accommodate us. Avoid suggesting that they do things our way because if we have a way, we're just wearing our *inexperience* on our shoulders. If we're going to make any suggestions, they need to help the project grow, not shift. A lateral change can make us appear selfish and incompetent. A forward change can make us appear valuable.

Whatever changes we recommend, our primary goal is not to break anything. **We don't want to be known for breaking things. We want to be known for fixing things.** So if we see any problems the team faces that our previous team didn't, we can mention that they may find success by adopting that previous process for this one thing.

This becomes even more critical on client projects versus employer projects. I currently do most of my work at an agency in San Francisco called Presence[‡], and while some projects are greenfield, where we get to decide the stack, we often have to work on existing and legacy client

[‡] <https://presencepg.com>

systems, which may be using twenty-year-old code.

There's no *way* I'm going to tell a client to change *anything* about their decades-old code base. Legacy code is fragile, and if it works, it works; leave it alone. Create integration points and separate your new code entirely when possible.

To recap all of this:

- Every codebase is different, and every team is different.
- Don't make teams accommodate you. Instead, go out of your way to accommodate their needs and help them rise as a group.
- Every new project is a unique opportunity to be a leader vs. a complainer.
- Match existing patterns before suggesting new ones.
- Prove growth opportunities, not change requests.

CHAPTER SEVEN

Diversify Your Portfolio

One of the most important things we can do to grow our career is diversify our portfolio. When that phrase is used in investing, it doesn't just mean investing in several tech companies instead of just one. It means don't just invest in tech companies - include some index funds, some gold, some medicinal companies, etc.

That concept is essentially the same for our projects. When I say diversify your portfolio, I don't mean that if we make websites, we should vary our languages from Node to PHP to JSP to Python. I literally mean, don't just build websites. Build games, desktop apps, mobile apps, and IoT projects. Work on large enterprise systems and startups. Work on 20-year-old legacy systems, and work on bleeding-edge R&D systems.

For example, if you're a C# developer, it makes sense to pick up Mono and build something in Unity. You're already way ahead of the curve, and you'll likely not only be able to pick it up and get something built much faster than you realize, but you can sell these games for easy passive income, learning on your own time and having fun with code.

The sooner you **learn to be comfortable with being uncomfortable**, the sooner you will naturally take on projects and responsibilities that you wouldn't usually have had you stayed in your comfort zone.

You don't need to shift your career unless you want to. If you're a website developer, I'm not suggesting that you start accepting projects to build games or mobile apps, for example. What I'm suggesting is that when you build something you don't usually work on, you expose yourself to patterns and processes that you may not ever see building websites, and not only do a lot of those patterns carry over from application to application, but you may even find that you enjoy building games much more than building websites. We spend thousands of hours of our lives writing code; shouldn't we enjoy our work? If you're coding boring stuff all the time, you're going to burn out and resent your life, my friend. Trust me.

One of my side projects is a geo-proximity startup where my system tracks people in health clubs while they work out so they can earn and redeem points at the club. In exchange for the rewards, the club gets analytics data on how members use the facility. This helps with marketing and equipment layout efforts. I built the prototypes on Arduinos using Bluetooth low-energy (BLE) beacons.

Figuring out triangulation and struggling through ideas that didn't work out well was a fun and educational experience. I learned that RFID and radio waves are unreliable in a gym setting where everything in the room is metal and people constantly move in weird positions while exercising. BLE worked out much better, and now I can keep that tidbit of information in my back pocket.

Parsing packets from binary streams and dealing with RFID and BLE technologies was not only super fun and new, but it also carried over to a client project. We were initially tasked with building the website, but I was able to assist with the integration of an electrical engineering component and increase my value with the client.

Another cool project I did that was outside of my comfort zone was integrating Minecraft into the real world with my son. We coded a small plugin that talked to a *code block* connected to some *red stone*--the plugin communicated with a Raspberry Pi, which had a button and an LED. When we pushed the button, it lit up the LED and a torch in the game, and when we stepped on a *pressure plate* in the game, it lit up the torch

and the LED on my Raspberry Pi.

It was a relatively small project that took us an afternoon to write the code and solder the LED. It was super fun, and now I know it's possible, which is half of what being a good consultant is all about; just having the ability to recall previous experiences and provide options to clients.

So right now, you have some homework. I challenge you to find something you don't typically work on and make something simple. It can be anything, but it should make you uncomfortable, and I ask that you share it in the Facebook group^{*}.

For example, if you typically only code stuff, try doing something visual and artistic. Go to Daz3d.com and download their app to see how 3d modeling and rendering work. Or, keep it simple, download a trial of Adobe Illustrator, and design a vector logo. These skills carry over to our workday and can help build cohesion with the design teams. If you want to keep it related to coding, try making a "Hello World" app with Expo using React Native, or try native Swift or Android to see what you can do. Even better, try to see how you could deploy your app both as a native desktop and a mobile app and see where you can leverage shared code and where you need to branch logic.

Don't spend more than an hour or two. The goal here is to get comfortable with being uncomfortable. And then repeat the process every so often until you start craving that discomfort. It won't take long before you realize that discomfort equals growth; after all, that's why you're here reading this book.

I'm excited to see what you all create and share in the Facebook group. Post a photo or a link to a GitHub repo, or even describe what you made if you cannot digitally share it. There's really no pressure here; just have fun.

* <https://www.facebook.com/groups/succeedinsoftware>

CHAPTER EIGHT

Proactive Mindset

There's a difference between being proactive and having a proactive mindset when it comes to being a software professional; both are important in their respect. For example, in my day-to-day, I have a lot of small maintenance tasks I need to do to help other people do their jobs. I need to ensure my hours are entered every day, both in our project tracking system and itemized on each ticket in our issue tracking system.

I need to ensure all my tickets are in the correct swim lanes and see if anybody has commented on them requiring action from me. I also need to review and close out tickets from other devs if they assign me as a dev tester. This is important because the product and project management teams are constantly looking at the kanban board to gauge the health of the current sprint, and if tickets are in the wrong place, they can't do their jobs.

I also need to do code reviews on pull requests and approve or reject them accordingly. Sometimes devs can work on other things while they wait to merge their code, but sometimes they're blocked until their PR is approved and will ask the team to do a quick review to unblock them. When there is a sense of urgency and pressure on a code review, I find that devs will do blind approvals to keep their peers unblocked, but that introduces bugs, so I need to get in there and review code before that happens to make sure the code quality stays high.

* * *

And, of course, the meetings--always with the meetings, right?

So everything I just listed is merely a subset of responsibilities and *none* of what I just listed is writing code or architecting a system. Being proactive with those responsibilities means I try to do them before someone pokes me with a reminder that they're waiting.

This ultimately comes down to responsible scheduling, where I try to spend the first half of my day keeping people unblocked and the second half of my day getting into flow state writing code. Not every day works out like this, but it's still a goal, and if I have too many meetings or foreseen interruptions in a day, I don't even bother trying to write code that day.

Now, what's the worst thing that can happen if I choose to be reactive instead of proactive with these responsibilities and decide to prioritize my coding tickets? Well, I'll likely be constantly interrupted with chat notifications asking for code reviews, moving my tickets, or checking in with another dev to see how they're doing, or that I'm late to a meeting, and so on.

It's not the end of the world, but it's certainly annoying and distracting, even for the people waiting on me. Being proactive with all this keeps me in control of my schedule so that when I *do* code, I can focus and write *great* code.

What does it mean, then, to have a **proactive mindset**? When we architect systems, regardless if we're at the network level building microservices or the UI level building components, we need to have a proactive mindset when implementing anything.

Before writing any code, we need to stop and step back a bit and consider all the moving pieces. Does our new code play nicely with the rest of the system, or is it a bolt-on solution that will require other areas of the codebase to know about it and accommodate it, so things don't break?

* * *

For example, on a recent web project, we had a media platform microservice whose primary job was to save and retrieve media assets into the static asset store, which was Amazon S3. On the UI, we had a file uploader; when the user selected a file to upload, React made an API call to the media platform, which provided a pre-signed S3 URL that React could then use to upload the file directly to S3. This was a great solution because it played nicely with our auth tokens, and we didn't have to pipe binary through our microservice unnecessarily.

The problem was that one of the customers had a firewall that blocked AWS URLs, so they couldn't upload the files. DevOps implemented a solution to proxy the S3 host through Akamai so the customer's firewall would allow it since it had the same domain name as the rest of the app.

The problem was that everywhere else in our microservice suite dealing with S3 went through Akamai, which is really expensive. So I got a ticket that read, "Use Akamai for customer-facing applications and bypass the proxy for internal-facing applications."

Having a proactive mindset told me a few things:

- We needed a way to manage the list of internal and customer-facing applications efficiently.
- Our code needed a way to identify if a request matched one of the applications on the list.
- The code shouldn't care about S3 or Akamai because, again, the proactive mindset told me that my client may want to change from S3 to GCS or switch from Akamai to Cloudflare, for example.

So I spent a couple of hours just *thinking* about all the moving pieces. It would have been super easy to write some conditional logic in the media platform and be done with it, but then I'd have been coding us into a corner. The code wouldn't scale because we'd have to return to the code when we wanted to change something. So my first approach was to leverage the data, not the code.

I made a dictionary of supported hostnames and mapped them to their respective pre-signed URL hostname. This dictionary was a simple JSON object that was then stored in AWS ParamStore and could be fully automated and populated by the DevOps team. It could differ on each deployment tier in the infrastructure, and my code wouldn't know or care. My code interpolated the pre-signed URL hostname with the value assigned to the inbound origin hostname. The code didn't care if the request came from a browser, another microservice, or a developer tool like Postman.

```
{
  "localhost:3001" : "http://127.0.0.1:8001",
  "localhost:3002" : "http://127.0.0.1:8001",
  "localhost:3003" : "http://127.0.0.1:8001",
  "localhost:3004" : "http://127.0.0.1:8001",
  "localhost:3005" : "http://127.0.0.1:8001",
  "localhost:3006" : "http://127.0.0.1:8001",
  "localhost:3007" : "http://127.0.0.1:8001",
  "localhost:3008" : "http://127.0.0.1:8001",
  "dev.mysite.com" : "https://dev.mysite.com",
  "int.mysite.com" : "https://int.mysite.com",
  "test.mysite.com" : "https://test.mysite.com",
  "mysite.com" : "https://mysite.com",
  "intra.dev.mysite.com" : "https://s3.amazonaws.com",
  "intra.int.mysite.com" : "https://s3.amazonaws.com",
  "intra.test.mysite.com" : "https://s3.amazonaws.com",
  "intra.mysite.com" : "https://s3.amazonaws.com",
  "green.intra.dev.mysite.com" : "https://s3.amazonaws.com",
  "green.intra.int.mysite.com" : "https://s3.amazonaws.com",
  "green.intra.test.mysite.com" : "https://s3.amazonaws.com",
  "green.intra.mysite.com" : "https://s3.amazonaws.com",
  "green.dev.mysite.com" : "https://dev.mysite.com",
  "green.int.mysite.com" : "https://int.mysite.com",
  "green.test.mysite.com" : "https://test.mysite.com",
  "green.mysite.com" : "https://mysite.com"
}
```

It was easy to test both with a unit test and an integration test. As we built new internal and customer-facing applications, DevOps only needed to add their hostnames to the dictionary, and the proxy switch worked precisely the same. The client can change from Akamai to Cloudflare, and the code will work the same, and the client can change from S3 to GCS, and the code will work the same.

So while you can see that having a proactive mindset can save thousands of hours and dollars on projects, two aspects of this mindset can get you in trouble, so you need to be careful: **pre-optimization** and **over-engineering**.

As great as it is to write future-proof and blazing-fast code, sometimes the project just isn't ready for it, and we need to get something out the door so that the project can earn money. Suppose we have two options that solve the same problem, where one is a quick and dirty option, and the other is robust but will push the roadmap back. In that case, it's often better to build the quick and dirty one and immediately create a technical debt ticket to refactor it to a more robust solution later.

Our clients or employers need to make money from the code that we deliver. We must be able to understand that and communicate a balanced stance to the product team that we want the project to succeed, but that ultimately it's their responsibility to include some technical debt tickets in every sprint. They need to understand that every compromise puts future sprints at risk of screeching to a halt while we rebuild some infrastructure to support a new feature or put all hands on deck to fix a flaw in the system.

Our ability to be proactive aligns with our ability to foresee and anticipate issues. The best way to improve this is to step back from what we're currently working on and consider as many other factors as possible.

If any of those external factors will be affected by our task or *affect* our task, then we must consider them when we work on it. Deal with them or not, do them now or do them later, but track the work so the people around us can understand the moving pieces as well as we do. The more you apply this principle, the better you will get, and it'll start to happen naturally.

Catching and considering potential issues early on will certainly increase your value on the project. You'll most likely be invited to more architectural meetings and feature-planning sessions. The key stakeholders will turn to you as an authority to help them sign off the roadmap and help them avoid making unrealistic deadline promises to customers or corporate execs that would put the project at risk.

CHAPTER NINE

Becoming The Fixer

Generally, when people come to *you* to help solve *their* problems, that's a good thing. It's not really the same thing as getting assigned a bug ticket; it's more like when someone *else* is assigned the bug ticket, and they struggle for a while, come to you, and you're able to help solve the problem quickly. They remember that, and that reputation will follow you.

What typically ends up happening is we start spending more time helping other people than working on our own stuff. Initially, this may seem overwhelming, and you may feel like you're getting distracted all the time, but the payoff comes rather quickly.

For example, if you're on an agile project and have a daily scrum, you should start noticing that when everybody gives their reports of what they worked on yesterday, more and more of the team will start mentioning your name in their report.

The management teams absolutely recognize this, and you'll likely start to be pulled into meetings above your pay grade. The managers see you helping everybody else, so maybe you can help them as well. You become known as the problem solver, and that's absolutely the title you want.

Of course, this is in direct contrast with the title you want to avoid,

which is the problem *creator*. So how do you get to a point where you can solve problems faster than everybody else? Well, there are a few ways we're going to cover, and none of them entail being psychic or being that dev who created the problems in the first place, so you know exactly what is breaking the system.

Local-First Development

It's a good practice to develop with an offline-first approach so we can catch and reproduce issues locally without having to deploy and wait for CI builds. Developing offline forces us to compartmentalize our services, which aligns perfectly with test-driven development, something we will go into much deeper in a later chapter.

The first thing we want to do is define the interaction points between our code and the services. For example, we can't run MailChimp on our laptops. MailChimp is a cloud-based email service. If our code is using the MailChimp API, we can only use it locally if we're making those API calls directly to the MailChimp servers from our computer.

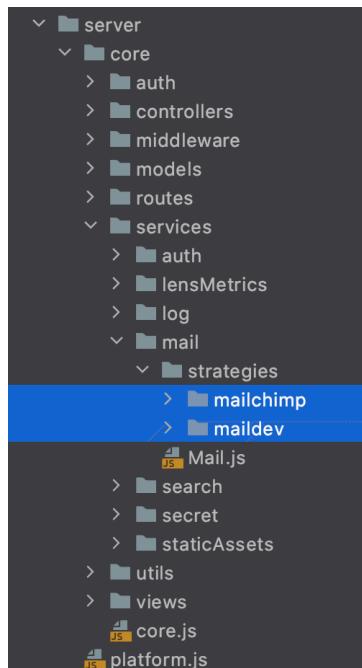
This is typically a bad practice, as we want SecOps to have the ability to control and track API traffic. If we make API calls to MailChimp from our personal computers, we're essentially bypassing our project's network.

Even if we're on a VPN, this is a bad idea because our localhost server isn't part of a controlled subnet or API gateway. Even a tool like ngrok can introduce security concerns, so if you're currently jumping through all those hoops, I'd recommend making a complete 180-degree turn and simplifying your entire infrastructure.

Imagine on our local machine, instead of making API calls to MailChimp, we spin up a simple SMTP server on a Docker container and make API calls to that instead. The one I use regularly is MailDev, which can be found with a quick search on Docker Hub. As with any tool I mention throughout this book, it's just a tool that can be replaced with one of a hundred others.

If it's not Docker, maybe it's Vagrant or a port forwarded locally-run server on our computer. Ultimately, abstraction and compartmentalization are essential here, not *how* we achieve them. I like Docker because it's easy and well-adopted, but please populate your own tool belt with the tools you prefer using.

So, let's use a design pattern. My favorite for situations like this is called Strategy^{*}, which is a pattern that allows us to choose an algorithm at runtime. Instead of forcing our application code to know about MailChimp, we create a general `Mail` service with two strategies: `mailchimp` and `maildev`. Imagine our `Mail` service is very simple and only exports a single function called `send`.



When we import our `Mail` service into the file that needs to send an email, we will give it a strategy (most likely from an environment variable) so it knows which functions to expose. Then we would invoke `Mail.send` instead of `MailChimp.send`.

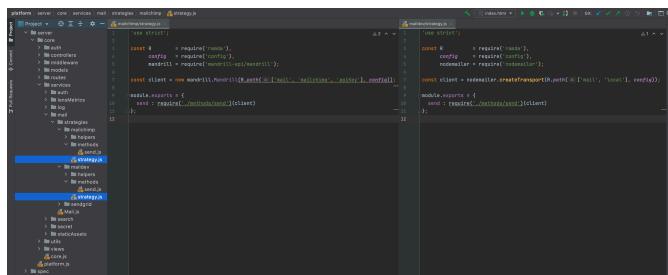
^{*} https://en.wikipedia.org/wiki/Strategy_pattern

```
const MailSvc = require('../../../services/mail/Mail')(R.pathOr([ 'mail', 'strategy'], config));

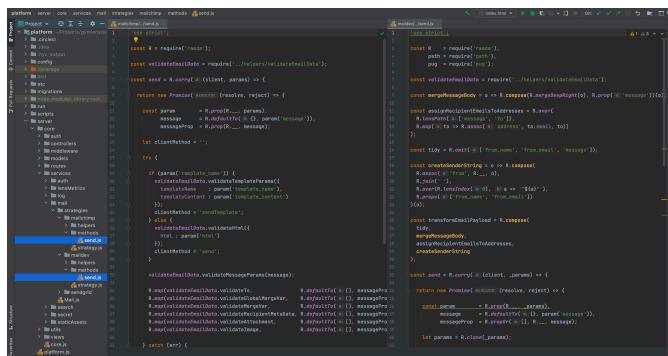
return MailSvc.send({
  template_name : emailTemplate,
  from_email : R.pathOr([ 'noreply@gymlens.com', b: ['mail', 'platform', 'from', 'email']](config),
  from_name : R.pathOr([ 'GymLens', b: ['mail', 'platform', 'from', 'name']](config),
  message : {
    subject : 'GymLens Password Reset',
    to : [
      {
        email : data('email'),
        name : inferFullName(resetData),
        type : 'to'
      }
    ],
    global_merge_vars : [

```

The signature of the `send` function would be identical in both strategies...



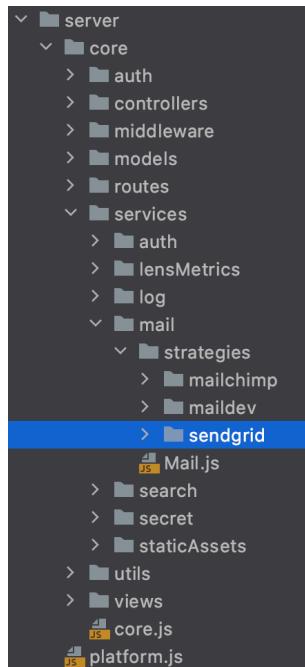
...but the guts of the functions would do slightly different things based on what the underlying tools care about in the payload.



We may need to rename some properties or transform some data a bit so that our `mailchimp` strategy can directly call the MailChimp `send` API, and our `maildev` strategy can call the MailDev `send` API on our Docker container. For the strategy identifiers, our local dev environment would

expose the environment variable "maildev," and DevOps can expose an environment variable "mailchimp" on the appropriate deployment tier. You can see how the separation of concerns and the leveraging of the strategy pattern can make life really easy for you and allow the product team to try out new vendors quickly.

What if they want to switch to Sendgrid? Simply copy and paste the `mailchimp` strategy, rename it to `sendgrid`, and change the guts of the new strategy methods to play nicely with the Sendgrid API. Now, if we have a blue/green deployment setup, we can send blue emails through MailChimp and green through Sendgrid.



The underlying point is that we don't want our application code to know or care about vendor code. We need to separate application logic from utility logic. Our unit tests will cover our service and all the strategies. Thanks to the strategy pattern syntax, we can mock out each of those third-party libraries or API calls quite easily to get 100% test coverage in our project.

* * *

This was one really deep example, but the key here is that we want to repeat this with all the interaction points and compartmentalize services that are *out* of our control so that our application is entirely *in* our control. That way, **when we need to work on a bug, we can first try to reproduce it locally and either rule in or rule out huge portions of the app instantly.**

If we use this email example, imagine the bug ticket is titled, "Customers are not receiving thank you emails." Well, suppose we try to reproduce this locally, and the MailDev Docker container sends an email just fine. In that case, we know that it likely has nothing to do with our application code unless there's some conditional logic surrounding that send call, in which case we'd need to ensure the conditions are met in the same way locally and remotely.

We know then to run our unit tests, and if the `mailchimp` tests pass, then it's really only one of three things:

- Our `mailchimp` test has a logical assertion bug.
- The issue is at the account level; maybe someone forgot to pay the MailChimp bill or the template names changed. Perhaps we've reached our send limit for the month. There are lots of things to rule out in this space, but logging into the vendor tool will likely display some messaging as to what's up with the account.
- The issue is in a DevOps context; maybe the environment variable is wrong, and the production app isn't pointing to the correct strategy. Perhaps it's using the wrong API key and sending emails to the sandbox account.

Regardless of the actual bug or solution, I want you to understand that how we structure our code can make a huge difference in how quickly we can troubleshoot issues and rule out possible causes. **Instead of blindly playing guess-and-check, we can systematically narrow down the culprits and present solid options to whoever asks us for help.**

* * *

I realize this might sound both super specific and super vague at the same time, but trust me when I say it gets creepy how quickly you'll start to develop hunches of what could be the cause of any given bug at any given moment literally because you can bubble sort your way down to a few interaction points in a matter of seconds.

I will close this point by saying that if you're working on a project with a ton of bugs, no unit tests, no local dev, and regular hotfix deployments, you may want to get out of that environment altogether. Find another job or move to another team. I'll cover this topic a bit deeper in the "Growth Strategies" section of this book.

So, if we can't reproduce the bug locally and suspect it's an issue with the remote environment, how can we still be the hero here? It may seem enticing to kick the ticket over to DevOps or another team, but that's not going to help us grow our career, and there's always a risk that our hunch was wrong; the bug gets kicked back to us, and the other team resents the wasted time.

Build Rapport

Early in every project, I request as much access to as many systems as possible to offer as much value as I can to the team. Not all teams want to hand out access to just anybody, so work with them and compromise where it makes sense, like getting a read-only account.

There is considerable value in being able to SSH into a remote AWS EC2 instance to see what's going on or remotely connect to a staging database to run some queries in the CLI and verify if some data exists or is missing.

If administrative access to pre-production systems isn't allowed, try to get permission to deploy a branch to a pre-production tier so you can rule out third-party integration issues. Suppose you depend on other team members to do these things for you. In that case, I highly recommend setting up a pair-programming session with them on your video conferencing app to solve the problem together.

It's not necessarily about who gets credit for the fix; rather, it's extremely valuable to projects when multiple contributors from multiple teams share knowledge of system intricacies and nuances. **If you are the one driving this effort, you will look *really* good to the executives who typically want cross-team collaboration more than anybody on the project.** I'll cover this more in a later chapter.

Learn The Logs

Lastly, learn the logs: learn where they exist, learn how they're grouped, learn how often they're updated, and learn which systems are logging and which systems are muted. For example, on one of my recent projects in AWS, I was troubleshooting an API issue. I discovered that none of our Lambdas had permission to log to Cloudwatch, which explained why certain data was missing from the log stream and confused the dev team.

Another issue I ran into was when Splunk grouped a bunch of log events that shouldn't have been. As it turned out, AWS was sending over Cloudwatch logs in chunks, and Splunk thought they were all related. This caused a lot of confusion when trying to follow a round trip with a request header because the log grouping contained multiple users that happened to be using the system in the same timeframe.

Try to learn the log systems as well as you can, as early as you can, because, with remote systems, you'll start to come up with theories and then use the logs to go one by one and prove or disprove them. The faster you can do this, the better you'll look and the happier the rest of your team will be.

CHAPTER TEN

When To Build It, When To Buy It

Twenty years ago, SAAS wasn't nearly as popular as today. There were a few obvious ones, like Salesforce, some payment gateways, merchant accounts, and whatnot, but software as a service didn't really have a place because APIs were terrible. We needed to use SOAP and XML, and the languages and frameworks driving most server-side development had modules and plugins to handle a lot of the stuff, like sending emails.

In PHP or Java, we'd just set up a monolith server that would run the web server, the email server, the database, and whatever else we needed. Sometimes our servers would be running thirty websites to distribute hosting costs.

When APIs got more popular, primarily thanks to JSON gaining traction, we started seeing more SAAS providers pop up; external image or video hosting, email, etc. Eventually, companies started offloading huge portions of their business to other companies, like Okta or Auth0, which handle user authentication, or Twilio, which addresses MFA or SMS.

So when should we build it and when should we buy it? At a very high level, I typically recommend that my clients build it if it's part of the core business offering and buy it if it's not. If you are a video game company, code your games and offload user registration, payments,

marketing, HR, Legal, etc.

For example, my company Alien Creations, has an extremely simple website. I build websites all day, so I'm confident I could build a payroll system to pay myself every two weeks, but why? It's so much easier to make an account on Gusto and let them deal with it.

I don't even need to integrate it into my website. It's just a huge responsibility I don't have to deal with anymore, and I can rest assured they will get it right because payroll is only one tiny slice of my business, whereas payroll *is* their business.

Another example, I built a website several years ago called Playlist.com. We primarily only focused on writing code to handle the UI and track user preferences and saved playlists. All the music recommendations and taste profiles came from The Echo Nest API, and all the MP3 files were streamed from a MediaNet CDN.

We didn't have to do any work pulling in MP3 files and saving them in S3 or something. We didn't have to try to reinvent song recommendations based on likes or listens because that was the core business offering from The Echo Nest, which Spotify later acquired.

Using these third-party services saved us likely a few months of work and several thousand dollars in hosting and development costs. Realistically, any solution we'd have created would have likely been sub-par to The Echo Nest and MediaNet because that's *all* they offer. Their entire business effort is making that one thing extraordinary.

Deciding whether to build it or buy it is ultimately up to the client since they not only would have to pay for the service but also pay you to implement it or code it from scratch. That said, this is how I typically weigh my options when communicating my recommendations to the client:

Use third-party when...

- It's cheap, easy, and you've done it before.

- It's something you can build later if you want to, but you want to get your product out the door quickly.
- It's something completely out of your league.
- There are other complimentary services already in play. For example, if you're using Hashicorp for one or two things, use the rest of their stack when possible. If you're using AWS for hosting, use S3 for static assets, Cloudwatch for logs, Elasticache, RDS, CloudSearch, and so on, so everything integrates nicely.
- It's not the core competency of the project owners--they will have to maintain whatever you deliver to them. The less they have to maintain, the better.

Roll your own when...

- The client wants something very custom, and you don't want to try to customize or override a third-party offering.
- You've already built a strategy for something on a previous project and can plug it in to save the client money.
- It's the client's core business offering.

While this may seem obvious for prominent features like email, search, SMS, and so on, what about third-party libraries? Should we make our own carousel or use a third-party component? What about a file uploader? Or form validation? I typically recommend using a framework or component library whenever possible for this integration layer to get 80% of the feature for free and then to polyfill anything missing when you cross that bridge. Ideally, **pick a stable library that's widely used**. Don't choose something brand new because if the creators abandon the project, you'll be stuck with someone else's unmaintained side project as the underlying infrastructure for your code base.

As I will likely repeat several more times in this book, the tools themselves are not important. Just weigh the risks and rewards for the ideal candidates and decide which ones your team wants to adopt, and move on to use the tools to build your project.

CHAPTER ELEVEN

Failing Successfully

Before we talk about anything else, I want to ensure we're on the same page with the word failure. Failure is something not only that we need to embrace but also something we should look forward to. It's a hard concept to grasp, especially since this book is titled *Succeed In Software*, not *Fail In Software*.

Failure is how we grow. Earlier in the "Experience" chapter, we talked about how we can learn from other people's mistakes, so we don't have to suffer through them ourselves. That also applies here--we don't need to be the ones failing. We can be on a team or part of a company that fails.

The underlying moment of realization that the outcome did not align with the intent is all we need to recall. The level of emotion associated with that moment will directly coincide with how vividly and quickly we can recall it and grow from it. If a coworker casually tells us over lunch to avoid using a library because it sucks, there's a good chance we won't remember that in six months if we're in a position to choose the library among others. We may vaguely recall the library being mentioned, but we may not remember the context and end up repeating our coworker's mistake.

Knowing that success will help us validate something and failure will help us grow, we want to apply this mindset to our career, where we

fail behind the scenes and succeed in public. Let's use the opposite scenario as an analogy to make it more obvious. Imagine when we worked on our side projects, everything worked just fine the first time we tried it, and in our actual job on our paying project, we made disastrous choices, and nothing worked how we wanted it to. We wouldn't last very long at our company. It would be embarrassing, and we would be the dev our colleagues would reference when telling stories of terrible devs in their network.

So flipping back to our goals, we want to deliver as much success as possible in our job and fail as much as we can on our side projects so that we can still learn and grow. Our employers and clients pay us to deliver working code and to offer sound advice. This has to come from a place of experience. We can't use our client projects to try new things unless our client hands us the new thing and we're upfront with them that this will be our first time working with it.

Earlier in the "Jack of all Trades, Master of Some" chapter, we discussed creating some "hello world" stuff with tools we don't usually use. This is precisely why we want to do that; so that when it comes time to be paid to implement it, we know *just* enough to tell our client, "Okay, that's fine, we can work with this," or "I've had some bad experiences with this in the past; we may want to consider other options."

A much more common example of failure in our line of work is a simple mistake. A typo in the code, a botched conflict merge, or if we unintentionally broke something which created some pain for another team.

When that happens, we own it, apologize, fix it, and communicate with the other team, so they know that we care about their time. When it's fixed, verify it for the other team and prove to them it's fixed, and odds are they'll remember us working with them to resolve the issue much more than we were the ones who broke it in the first place.

When things break on the project, don't be a public complainer, and don't throw people under the bus for breaking things. It can create an image that we're not only infallible and can do no wrong but also that

we're somehow entitled to have a perfect system in place for us to do our job. Nobody wants to work with that person. If we ever accidentally broke something, we'd look much worse to the team if we'd been the ones complaining about imperfections. We'd be seen as part of the problem, which is the opposite of our objective.

When things break, point out that the problem is likely the process, not the person. Say something like, "Can we look into getting our test coverage up to 100% so we can catch these issues before they get deployed?" or "Should we look into requiring an additional reviewer on pull requests?" Just make it known that we're there to help, not judge.

So what about success? In our last chapter, we talked about helping to find and fix bugs quickly. Well, what if another dev causes the bug? I don't even bring it up if it's just a typo and an honest mistake. Refer to everything I just talked about over the past few minutes.

If it was a logical bug, I tend to comment in the devs' shared channel of our chat application--something like, "Hey team, just a heads up, when working with the user data on the profile page, we can't assume the user has these particular fields in their profile because those are populated through another flow. The user may not have done that yet, so we just need to account for an incomplete profile, or we'll get bugs like ticket 123."

Notice I didn't tag the author because I didn't want to embarrass the dev, and I didn't send a private message to the author because being aware of nuances in the code and data is a shared value that all the devs on the team should have.

So it's an excellent coaching opportunity for the dev team, nobody was thrown under the bus, and most importantly, I didn't say something like, "I fixed this bug" or "I found the issue". Those claims will be obvious when I move the ticket to the QA column in our issue tracker.

Ultimately, what really matters is that we're balanced and honest, so if we're going to take credit when we succeed, then we must also take responsibility when we fail. I prefer to help others succeed so we can

succeed and fail as a team. As long as we grow and stay humble, our relationships and careers will blossom.

That said, I'm writing this book to help you grow and succeed, so please, when you fail hard and learn from a big mistake, let us know on the Facebook group so we can all learn from it. And when you get your pay raises and promotions, let us know as well so we can all celebrate together.

CHAPTER TWELVE

Build A Gut And Learn To Trust It

One of the coolest things that come from experience is being presented with a problem and having an almost supernatural feeling about it, like knowing the answer before even hearing the full context. One example I used earlier is when we get CORS errors. Often a bug ticket will mention a CORS error on a page, and the action item will be updating the CORS rule, but most of the time, my gut tells me that the error has nothing to do *with* CORS.

To give you some context, if you're not a web developer, CORS stands for Cross-Origin Resource Sharing. It's essentially a ruleset that you set up on your server when exposing an API that basically says, "I only want my API to work for domains that match XYZ," or "I only want my API to accept these headers," and so on. These rules are sent to the browser in pre-flight requests called OPTIONS calls.

An OPTIONS request precedes each API call from the front-end to fetch the initial headers. One of the headers is called Access-Control-Allow-Origin. If the client origin doesn't match the value in the header, the browser throws a CORS error, preventing the website from making the subsequent API call. That error is triggered either by an Access-Control-Allow-Origin header mismatch *or* if the header is missing entirely. If the pre-flight request never makes it to the API endpoint, it'll never receive the header, and the browser will think we're simply blocking it with a CORS rule.

* * *

Most of the time, this is not intentional; it's a typo in the route, Nginx is misconfigured, or something equally unrelated to CORS configuration. My gut tells me it has nothing to *do* with CORS because we hardly ever do anything *with cors*. The people who create the bug tickets don't know that. They see the error, log it in the ticket, and assume the remedy is to fix the CORS rule, which, 99% of the time, is an incorrect assumption.

The more you grow and learn from your experiences and mistakes, the deeper the experience imprints on you. The faster you'll be able to recall your experience and leverage your subconscious mind to search through it all in real-time as problems are being described to you. The nice thing about being able to develop a gut instinct is that you don't have to make grand claims and hope you're right. You simply need to bring up your gut feeling as something you'd like to rule out before diving deep into the problem.

These days I'll do this a lot with bug tickets where we need to fix it fast, and it takes some experience and a senior mindset to find and identify the bug. The actual work can most likely be assigned to any of the devs on the team--I don't need to fix the bug to receive credit for finding the bug.

I often use my gut when doing comparative analysis for tools and services that the client is considering. Reading a few snippets on a library's main page and gauging whether it feels like a dependable library or a waste of time is usually all that's needed.

I'll also use my gut instinct with new-hire candidates. A quick look at a GitHub profile and a few snippets can tell me all I need to know to decide if a dev will solve or create problems. Building a gut takes time, and learning to trust it takes even more time, so I recommend starting today.

Lastly, *never* go against your gut instinct. Your gut instinct is your subconscious mind trying to tell your superconscious mind something important. The information just isn't loaded into your conscience yet. Always bet on yourself. If you're not comfortable putting your gut

feeling out there, you're welcome to keep it to yourself, but remember, **your initial gut instinct isn't something you have to commit to; it's just the first thing you want to rule out.** If you're right, you look *outstanding*, and if you're wrong, who cares? It's part of the troubleshooting effort.

Your homework for this chapter: before starting any work on a bug, quickly write down what you think is the possible cause. Write down two or three possibilities if the bug is vague. The more you do this, the more you're going to start guessing correctly before doing any investigation.

CHAPTER THIRTEEN

Stop Multitasking

What do you do when you're working on a file, and your calendar reminds you that you must attend a weekly meeting? If you're in the office, do you bring your laptop and continue working during the meeting? If you're working remotely, do you just put the meeting on the conferencing software in the background and listen in while you continue to code?

What you're essentially doing is splitting your focus between two things at the same time. What if the meeting entails a slide show or screen share? Do you attempt to code *and* listen to people in the meeting *and* look at the visual reference being discussed?

We all work with people who work through meetings, and if you ask most of them about something covered in the meeting several days later, they won't be able to recall it. If they were engaged in the meeting, it would have taken them just as long to finish the code had they just waited until after the meeting to focus on it, and the odds of introducing bugs in their commit are *much* higher because of the distractions.

Almost every state in the US has already made it illegal to text while

driving^{*}, and some won't even let you wear headphones while driving[†]. These laws are based on the fact that accidents are more likely to happen if your focus is divided.

Recent studies have also shown that valuable time is lost when people are forced to change gears in the middle of a task. The more complex the task, the more time is lost.[‡] The lost time is the time that's usually spent by the frontal lobe of your brain making decisions and establishing priorities.

In our line of work, these typically come as scheduled distractions like meetings or team events, where we're not expected to interact necessarily, but we're expected to consume the knowledge and understand the decisions that were made in the meeting, so we have appropriate direction moving forward on our projects.

If our focus is on our laptop, we're not receiving the information, and our code quality is at risk if our focus is on the meeting. These situations demand our **time** and **presence** but not necessarily our **focus**, which is why it's tempting to want to continue to work through them.

If it's not a meeting, maybe you're starting another ticket while waiting for a build to finish. If the build fails, you're now fifteen minutes into a new file and have to return to the old one. For these situations, I typically recommend taking the opportunity to do the busy work like checking emails or doing code reviews. Something that you can drop on a dime regardless if the build passes or fails.

Situations that demand our focus make it nearly impossible to multitask; for example, if we're working on a file and another dev pings us in chat asking for help solving a bug, or if we're asked to jump on a call and explain something or approve something in a small group. We

^{*} <https://www.ncsl.org/research/transportation/cellular-phone-use-and-texting-while-driving-laws.aspx>

[†] <https://www.findlaw.com/legalblogs/law-and-life/illegal-to-drive-with-headphones/>

[‡] <https://www.apa.org/topics/research/multitasking>

can't continue working on other items during those situations because we must focus on *and* engage with the new task.

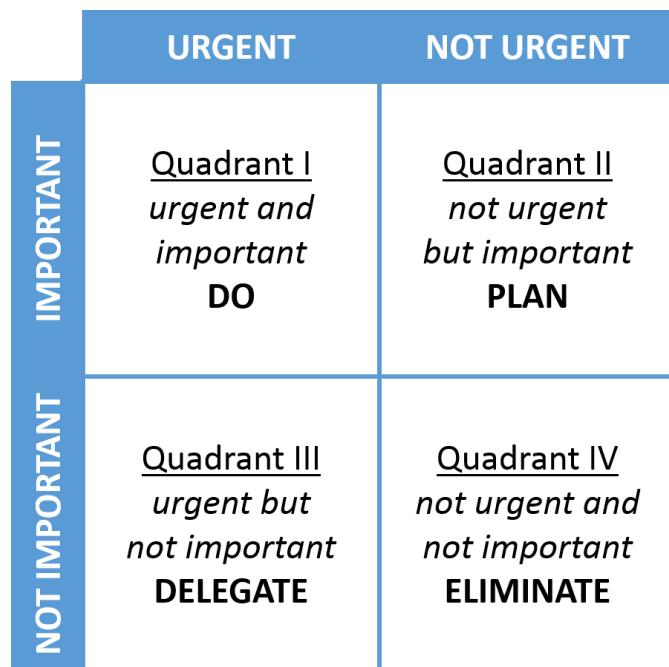
Most often, when we're put in those situations, our ability to problem solve and participate is very high because we're giving all our focus to the other dev and their issue. We can't do anything else because they depend on us in real time. It likely wouldn't even occur to us to check our emails or catch up on Facebook. This is called single-handling. It's the opposite of multitasking, and it's how we're going to learn how to finish an 8-hour day in 3 hours instead of dragging out 3 hours of work over 8 hours. We want to address the two problems we have with the previously mentioned situations.

The first problem is that others treat our work as a second-class deliverable. The meeting or pairing session *feels* more important because there is urgency attached, but us getting our work done is likely just as essential as our peers getting their work done. The more of a mentor and manager you become, the fewer technical tasks will be asked of you, and you'll have much more time to assist your team. If this doesn't happen, your role and responsibilities will be misaligned, and you should consider moving onward and upward to a new employer. We'll cover this later on in another chapter.

The second problem is that when we come back to our work after being distracted by another task, we have to get back into flow and pick up where we left off. That takes both time and mental energy. **We want to avoid starting and stopping as much as possible.** Pretend we're trying to get the best gas mileage possible from our brain, and we want highway miles, not city miles. Planned stops vs. rush hour traffic.

Several years ago, Steven Covey wrote a famous book called "The 7 Habits of Highly Effective People." We were forced to read the book and write reports on it in high school. One thing that stuck with me from that book was the four quadrants of any task. The task is either not important and not urgent, not important but urgent, important but not urgent, or important and urgent.

* * *



The goal, of course, is to try to do important and urgent things before doing unimportant and not urgent things. The other two can be prioritized based on our time blocks and agenda for the day. For example, if we have twenty unread emails, we don't have to read all twenty.

We can spend 5-10 seconds on each email and decide if it's trash or something that needs to be dealt with later. If it is important and urgent, we deal with it immediately, and if it's urgent but not important, then we weigh how long it will take. If we think it will take more than five minutes to complete, we do it later; otherwise, we do it immediately. Find a system that works for you, but the principle is the same for all tasks like this.

A system like this can drastically help us reduce our daily distraction potential. Most distractions come from other people thinking their task is more important than ours, and we need to decide whose career is more important, ours or theirs. **I'm not suggesting we don't help our**

peers. I'm saying schedule our availability. Please understand that people who distract us are, for the most part, not trying to be inconsiderate of our time.

My kids will randomly come into my office and say hi, and as much as I want to tell them that I need to focus on a ticket, I put my family above my work, so it's a welcomed distraction. I do have signals that I use to let the people around me know when I really need to focus. At home, I'll close my office door, telling my family that I'm either on a work call or that I need to focus for a while, and then in exchange, I'll take an extended break and hang out with them.

In my work environment, when I start work on a ticket, I will set my chat status to "Do Not Disturb" so that my colleagues know that I fully intend to ignore them. They're welcome to leave me messages, but I have no intention of reading them until my status is set back to "Available". The trick is to communicate up front with your supervisors and peers so they don't see you as an isolate just working in a cave. As I've said in previous chapters, I firmly believe communication and collaboration are massive in this field. Find your balance, and instead of just shutting out your team when you need to focus, communicate to them that you're shutting them out so you can concentrate. It may sound silly, but it's essential.

Lastly, when you feel the urge to multitask, and you're in the middle of a file, I recommend doing what I do. I write a `TODO` comment that reads "LEFT OFF HERE," with a quick one-liner stating what I was doing and planned to do next. It's all I need to restore my focus after an undetermined period of doing another task.

CHAPTER FOURTEEN

Flow

Have you ever been so immersed in a task that you lost track of time? Your focus was so intense that the world around you faded, and all that mattered was the task in front of you. Your productivity skyrocketed because you were focused on one thing and only one thing for an extended period, and because your focus was so intense, you made massive progress on that task--much more than you would typically make.

That state is called "flow." To achieve maximum productivity, you *must* understand it because it sets the highest achievers apart from everyone else. When you're in flow, your mind is 100% engaged. You're entirely focused on the work in front of you. Rather than emptying your mind, you are trying to fill it completely. You are so focused that nothing can distract you from the job in front of you. You've got laser focus.

Flow requires the practice of single-handling, which we covered in the previous chapter. When you have only one task in front of you, it's possible to zero in on this one task so profoundly that everything around you becomes a blur. When you're in flow, you'll experience several significant things:

- **Total immersion in the task at hand:** You're so immersed in the task at hand that everything else fades to the background. Being in flow is highly enjoyable and can turn even the most

challenging jobs into pleasant experiences.

- **Increase in productivity:** When you're in flow, your productivity goes through the roof. Why? Because you're only focused on a single task. You're not trying to multitask, and you make significant progress because your energy is focused so intensely on one thing.
- **No distractions:** When you enter flow, your brain shuts out everything except what's in front of you. Nothing else seems to matter. You are utterly oblivious to the distractions flying around you and focused on achieving something that matters to you.
- **Lose track of time:** Because you're so immersed in your task, you may lose track of time. This is also why you tend to be more productive when in flow. You can dedicate more focused time to a task than usual and get significantly more done.
- **Increased enjoyment of the task:** When you're in flow, you're not thinking about what you're missing out on. You're not thinking about the problems in your life. All your attention is in a single spot, not scattered about by a thousand different thoughts and worries. The result is hyper-productive bliss.
- **You're stretching yourself:** Entering flow requires pushing yourself and tackling worthy, challenging goals. It's difficult to get into flow when you're working on mundane, tedious tasks.
- **You live a more meaningful life:** The primary benefit of flow is that it allows you to live a more meaningful life. How? Flow ensures that you dedicate a significant portion of your time to meaningful tasks and will enable you to experience joy while performing those tasks.

Now that you understand what flow is and can identify *when* you're in flow, how do we get there on our own terms?

* * *

Step #1: Choose a challenging task. It's not easy to get into flow if you're working on something you don't enjoy or on something that feels beneath you. You can do it, but it takes more practice. I get into flow when I'm coding unit tests and don't enjoy writing them at all. The trick is to have *respect* for the task. Either respect that it challenges you or respect that the project benefits from it and you're the best person for the job.

Step #2: Set clear goals on what you're trying to achieve. When seeking to enter flow, be specific about exactly what you're trying to accomplish during a particular period. Set a time limit for yourself, and then choose what you will aim to achieve. Putting boundaries on yourself forces you to concentrate quickly so you don't waste time. Do one thing and only one thing during your period of intense concentration.

Consider using the Pomodoro method to help you enter flow. Using the Pomodoro method, you force yourself to focus intensely for 25 minutes without distractions. After 25 minutes, you take a 5-minute break, and after 4 Pomodoro cycles, you take a more extended break, like 15-30 minutes. It's important to note that you give yourself enough time to get into flow. For example, if you have a meeting scheduled in 15 minutes, that's probably not enough time to get fully immersed in the task.

Step #3: Cut out all distractions. Put your phone on airplane mode. Close your email and silence all notifications. Block social media sites if necessary. Listen to background music that will drown out distracting conversations. Headphones are a huge help as they will also cut out background noise. Remember, your goal is to give 100% of your attention to a single task. If anything cuts into that attention, it needs to be eliminated. Be extremely vigilant about this. Once we're *in* flow, distractions won't be much of an issue, but while trying to achieve the state, distractions can derail us quite easily.

Step #4: Eliminate multitasking. If you want to enter flow, you must eliminate all multitasking. Trying to work on multiple things at once can be tempting, but doing so will absolutely kill your flow. We covered this in our last chapter. You must give *all* your focus, drive, and energy to **one** thing. Becoming so immersed in one thing allows all other things to

fade away. This simply can't happen if you're simultaneously trying to listen in on a meeting, respond to chat notifications, and so on. When you choose a task, as I mentioned earlier, you should respect it enough to give it your 100% undivided attention. Your brain simply doesn't have enough power to focus on multiple things simultaneously.

Step #5: Strengthen your concentration. If you want to enter flow, you absolutely must be able to concentrate fully. If your attention wanders, you'll struggle to give 100% of yourself to what's in front of you. How can you ensure that your concentration is at peak capacity? One of the primary ways is to ensure that you get enough sleep. High-performers know that sleep is essential to peak performance. They go to bed at a reasonable hour and ensure that they have good sleep habits. If you need an extra jolt of concentration during the day, consider drinking a cup of coffee or tea a short time before you need to enter flow. Caffeine can help your brain concentrate. It won't give you energy, but it'll block the adenosine receptors in the brain that make us feel fatigued.

Step #6: Monitor your emotional state. If you still find it challenging to enter flow, take a step back and look at your emotions. Are you feeling frustrated, worried, or overwhelmed? If this is the case, you should relax before you enter flow. Put on some chill music or take five minutes and watch a funny video on YouTube to boost your mood and take your mind off your anxieties.

One thing that can be extremely helpful when you need to enter flow is creating a ritual or series of actions you perform every time you're about to work. The ceremony is a trigger – an indicator to your body and brain that something important is about to happen. The more you perform the ritual, the more your body interprets it as a sign to enter flow, and the easier it is to get into flow. It creates a deep neurological link and helps you get the tunnel vision to shut out all your other responsibilities while focusing on the task in front of you.

For example, your ritual might look like the following:

1. Make a cup of coffee.

2. Take ten deep breaths with the cadence of inhaling for four seconds through your nose, holding for four seconds, and exhaling for eight seconds through your mouth. This will slow your heart rate down and help saturate your bloodstream with oxygen, helping to get rid of brain fog and make you more alert.
3. Close your door.
4. Close your email.
5. Turn your phone on airplane mode.
6. Mute all notifications on chat applications.
7. Put on your headphones.
8. Turn on Brain.fm or a Deep Focus playlist on Spotify to keep you from being distracted by sound. I recommend finding a playlist you enjoy and re-using it indefinitely.
9. Begin.

Every time you want to enter flow, follow this exact series of steps to help train your body and brain, and you'll be able to get into flow *much* more effortlessly.

The last thing I want to mention is to remember to be mindful of your time. You likely work on a team, and no matter how efficient you are at your task while in flow, the reality is you're shutting yourself out and away from the team when you do this, so reserve this for blocks of time when you're pre-communicated to your colleagues that you will not be available, and in turn go the extra mile when you're *not* working on your own tasks to be proactively available to assist the rest of the team.

CHAPTER FIFTEEN

The Zen Road

In this chapter, we'll discuss how to achieve zen in our workday using these five principles: **iteration**, **delegation**, **deferral**, **compromise**, and **balance**. Our goal is to get as much fulfillment from our careers as possible. We want to avoid burnout and be in control of our day, so we enjoy doing what we do. As we grow in our careers, we'll likely be moving further and further away from the things that drew us to this field in the first place.

Maybe you love tinkering with new code to "figure stuff out," or you love building applications from start to finish to showcase them to the world like works of art. You'll likely have less ownership of low-level code implementation as you get promoted. You'll probably be part of larger teams and won't be building anything by yourself, let alone anything you personally envisioned. This reality can be a wake-up call for some of us; one day, we're sitting at our desks overworked and feeling miserable.

So let's talk about ways we can bring order to the chaos of our day-to-day so we'll be happier and have energy left over at the end of the day to work on our own projects that bring us personal joy.

Iteration

Iterative development is relatively new but has proven invaluable in an agile environment. The idea with iterative development is to do the bare minimum necessary to deliver base functionality and move on. By doing this, we're developing horizontally instead of vertically and can get a minimum viable product out the door and into customers' hands much faster.

From there, customer feedback can drive the next iteration of development. Do they want more features, or do they want one of the features to be enhanced? Iterative development puts our customers in charge of product direction, so all our work meets a specific customer demand.

The contrast of iterative development is waterfall development, where we finish building one thing before moving on to the next. It implies the product team already knows exactly what customers want, which is rarely the case. Most of the time, the product teams have ideas they think are great, but ultimately they're created in a biased sandbox, and they won't know if the ideas are good until the product is released to customers.

Let's use a real-world example. Imagine we need to implement authentication into a new product. We have *so* many options--custom registration, Social login, SAML, OIDC, etc. The iterative development mindset would tell us that we should implement the most basic registration possible to get customers into the system. We can add a "Login with Facebook" button and have users in a database table in a couple of hours tops, and then we should move on.

I realize not everyone has a Facebook account. However, we can plan an initial marketing campaign to use Facebook ads so our prospects will not only have Facebook accounts, but they'll be *on* Facebook when they click our ad and can register with our product quite seamlessly.

Our login doesn't need to support signing in with Google or LinkedIn

or a custom email and password combo because no users have asked for it, and we can add those on the next iteration. Instead of spending an entire sprint loading up support for the ultimate login page, we spent two hours and moved on. Imagine how much we can cover in a sprint with this approach.

Of course, the code we commit should be thoroughly tested and up to our highest standards. We want it to scale, so we should leverage the strategy pattern and compartmentalize our services so that future iterations have an obvious plug-and-play feel.

Delegation

I used to think that the way to get noticed and promoted was to be the rock star on my team. I thought if I said no when someone asked me to do a task, someone else would say yes, and *they* would get the promotion. What always ended up happening was I was always working, and the managers were always leaving early and enjoying their evenings and weekends.

I remember one director on a former project leaving at 5 pm one day while another dev and I stayed literally all night until 8 AM the next morning. We were stuck working with an offshore IT department that was refusing to deploy our work in the middle of the night because they never got the approval to deploy it from the director, who left.

The director came in the following day at 8 AM. He asked us why we were still there and explained that since we missed the deadline for deploying our work the night before, we should plan to deploy it the following month instead. He made us work all night for him and set us up for failure by forgetting to tell the offshore IT team that we'd be needing them. Not only did he never apologize or give us any credit or compliments for pulling an all-nighter, but he also blamed us for pushing the deadline back, and was promoted to Senior Director that same month.

That was the moment in my career when I realized that **working more than everybody else in the room does not get you promoted**. It just makes us look overwhelmed and incompetent.

The reality is that most tasks don't need to get completed when we think they do. Product likes to put a false sense of urgency on tasks, but that's typically only because they made promises to people above them, and they don't want to be put in a situation of explaining why they missed some promised dates.

Usually, though, the product team promises an entire roadmap made up of smaller promises. During the sprint planning phase, telling the

product team that we're at our limit for how much work we can accept that sprint is perfectly ok. They typically don't care if we can do more. They're often more focused on trying to be accurate with the sprint velocity so their promises won't be broken.

It's up to us to underpromise and over-deliver so that *they* can underpromise and over-deliver, meaning we should accept less work during sprint planning. We want to complete all our tasks and have time left to help the other devs and pull some tickets in from the backlog if we run out of work. We will look so much better to everybody around us when we have a light "To do" list by being proactive with our time vs. having a full "To do" list and making people worry and wonder if we'll be able to finish all the work.

The more tasks we own, the less we can help and collaborate. When this happens, we need to take action immediately and ask other devs if they can help by taking some of our tickets. Asking another dev to help us is a great way to build rapport with that dev--much deeper than when we help them. It feels like reverse psychology, but that's how our brains are wired.

To summarize, we should delegate our work when we feel overwhelmed and learn to say no to people when they want us to stretch ourselves too thin. We don't need to downplay our skills or abilities when refusing additional work. Instead of saying, "I can't; I'm already overwhelmed," we can say, "I'm at sprint capacity, but I'm happy to do it next sprint."

Keep it brief and professional, and we'll come across as having our shit together. Being reliable and consistent with our sprint velocity is typically much more appreciated by the product team than being able to take on extra tasks to help cover for someone who didn't plan their roadmap well. Remember, doing that favor for them will only affect how we see them, not how they see us.

Deferral

We covered deferral a bit in the delegation principle but let's zoom in on this one. Deferral is basically delegation to your future self. During sprint planning meetings, I'm very conscious about keeping my sprint light, so I punt low-priority tickets to future sprints but keep ownership of them.

This does a few things. It keeps me on the client's radar for several weeks or months when they're looking at the roadmap, so when the execs are renewing the SoW, there is no question that I'm a valuable asset to the project.

If I locked myself in my office and worked nights and weekends to complete all the tasks assigned to me in the backlog, I'd have no more work, and they'd roll me off the project. Part of being a good consultant is maintaining a presence on the project to help steer the ship in the right direction when it starts to shift off course.

On the surface, it might appear that my value to the project is calculated in Jira story points, but any dev can write code and move those tickets through the swim lanes. I need the clients to see that my value to the project (what makes me **irreplaceable**) is all the other stuff you're learning in this book: how to see the forest from the trees, how to communicate across teams, and translate product subjective to technical objective, how to help junior devs become senior devs, and so on.

You must believe you are that person before anybody else will treat you like that person. This is all part of taking charge of your software career. You start deciding how many tickets you accept in a sprint instead of letting the product owner or dev manager decide.

As long as your tickets are done, and you're working on something to help move the team and project forward and upward, you will be perceived as valuable. Nobody should give you a hard time for having no tickets in your queue if you're pairing with other devs, helping them grow and deliver their tickets. If anybody does, it's their problem, not

yours, and is a direct reflection of their own insecurities. Don't change for them--you're leading by example. Some people fear change regardless if it's a positive change.

One of the most common things I'll defer is big under-scooped features. The product teams on any number of projects I've worked on over the years love to pitch huge features that don't fit with the existing infrastructure. They'll often add the tickets to the sprint planning out of nowhere, and we need to catch them and kick them back to grooming.

I talked earlier about developing a gut. This is when my gut almost always tells me to stop the feature in its tracks. I'll say, "This feels like a much bigger feature than what's being presented in the ticket. It doesn't feel like it was groomed properly. I recommend kicking this out a couple of sprints and adding a spike to the next sprint instead. We can assess all the integration points and see how much extra work will be needed to support the ticket."

Fifteen or even ten years ago, I'd have just dived right in on a new feature ticket and worried about the integration points *after* building the feature. Of course, I learned that this is a messy approach. When you don't have integration contracts at least identified, let alone specced out, you're just asking for a bolt-on solution that will regularly break as the system evolves around it.

Compromise

Early in my career, I used to be extremely passionate about my way of programming. I would resist lateral change because I had convinced myself that my velocity would tank if I switched. When I say lateral change, I mean switching from a two-space indent to a four-space indent, or from underscore to camelCase, or being told to use charting library "B" when I'd been using charting library "A" on the past three projects. None of these changes are better or worse; they're just changes, so I resisted.

The problem, of course, is that while I was extremely comfortable with underscores, another dev on the project was highly comfortable with camelCase. To this dev, I came off as a stubborn asshole. The manager at the time sided with the other dev, and we went with camelCase. It didn't take me long to get used to it, and I've stuck with it ever since.

I've used all of these lateral comparables throughout my career. Having been forced to switch countless times, I can assure you that **none** of these preferences are worth standing your ground.

You can vote for them if you're starting a new project, but I highly recommend you save your battles for things that matter, like object-oriented vs. functional patterns, for example. That's a big deal because you don't want to deal with mutation if you don't need to. For example, where OOP may work well for a video game, it introduces unnecessary complexity and overhead on a web app.

By showing your team that you're willing to let go of your preferences to them, they'll most likely reciprocate that when you insist on something you're passionate about.

One strategy I've used in the past on a team is when we had a meeting to redefine our coding standards. We had roughly twenty items on the agenda, similar to the few I mentioned above. Each dev in the forum got twenty "votes." They could vote on any item with as many votes as they had in their hand. If an item they voted for wins, they lose those votes

for future items. It made it really fair and democratic.

For example, if I participated in this approach in the future, I'd save all my votes for the JavaScript library Ramda. It's helped to improve my JavaScript projects so drastically that I can't even comprehend working on a project without it or some alternative functional programming library like Sanctuary.

Let go of little biases and allow the linter to do its thing. Set up your IDE so that when you hit the tab key, it adds however many indentation spaces the team decides on, and your workflow really doesn't change all that much. It allows you to expand and grow every time you don't get your way, so take it with a grain of salt and compromise whenever possible.

People will like working with you much more when you make it clear to them that their opinions matter and that your working relationship with them is more important to you than some syntax that helps you type faster.

Balance

The last principle I want to cover to help you achieve zen in your work day is learning to appreciate work/life balance.

We've already covered delegation and deferral to help free up your workload, but taking this a step further, you need to be able to clock out at the end of your work day. If you're passionate about coding and software development, this doesn't mean you need to unplug. It would be best if you unplugged at some point every day, but I'm talking about setting a hard boundary of when you stop helping your employer live their life and start living your own.

Your work day is your employer's dream. They dreamed of getting to the point of owning a business and hiring staff to build stuff for them. You realize that dream for them from 8-5, Monday through Friday, week after week, or whatever your current schedule is. If you already work for yourself and have paying clients, set your workday boundaries in your SoW when you sign your paperwork.

To be able to give your 100% on any project, you need to be able to recharge and spend time away from the project. I wake up every morning around 5:30-6:30 AM and take 60-90 minutes to work on whatever side project I've assigned myself, like this book. At 7:45 AM, I'll let my dogs out, have a quick breakfast, and work from 8-5, taking a lunch break from noon to one to hang out with my wife and kids.

I relax for a half hour after work and then work out by going to jiu-jitsu for 2 hours or exercising in my garage and running around the neighborhood. I'll have dinner around 8 PM and hang out with my family again, watching TV or whatever they want to do. On the weekends, I have more jiu-jitsu, and then I rarely touch my computer unless my family has their own plans, and then I'll tinker with some Arduinos or something fun, maybe play some video games or read. Come Monday, I've had two full days to recharge, and I'm ready to give my 100% again. Imagine, had I been working on my full-time project all weekend, how much I would dread Monday. I'd feel like I never get a

break like I'm overworked or burning out.

Burnout is a very real thing--you need your breaks. Again, **you don't need a break from *your* passion; you need a break from your employer's passion.** If you're a full-time dev building your own stuff and doing client work, and you love what you do, by all means, keep doing your thing, friend! Just don't neglect the other people in your life, or you will likely lose what you love the most.

When my first son was born, my instincts kicked in to be a provider. I was young, and the only way I knew how to provide was to earn money writing software. I'd work and work and work. I was in my office for 12 to 14 hours every day, just coding my life away, trying to earn as much money as possible so my wife wouldn't have to work.

My wife started resenting me for working all the time, and I started resenting her for not appreciating how hard I worked for her and our son. We weren't seeing eye to eye at all. Thankfully, she was able to help me understand that I was missing out on precious early moments with my son, and she was feeling alone and overwhelmed having to take care of him all day by herself. We ultimately decided on a work schedule that made us both feel happier, our stress levels went way down, and our relationship has been stronger ever since.

It was at that moment when I realized the importance of work/life balance. I promised myself that if I ever had to work an all-nighter again, I would take the next two days off to make up for it. Balance is everything, my friends.

CHAPTER SIXTEEN

Cancerous Traits To Avoid

Imagine two years from now, we've moved on, and we need to interview some dev candidates at our new workplace. Imagine one of those candidates is a former colleague, maybe someone we're working with right now.

If we had a positive working relationship with this person, odds are we will be extremely lax in the interview and sing their praises to the hiring manager or HR. On the other hand, what if we hated working with this person, and we were able to tell the hiring manager to skip the interview entirely and focus on the other candidates?

This happens more than you'd think, and it really helps prove the saying, "It's not *what* you know; it's *who* you know." The list we're going over in this chapter is collected from actual experiences with real colleagues. Not everything on this list bothers me personally. My skin is pretty thick, but if the dev represents our agency and the behavior makes us look bad to the client, then the dev is gone.

With that, let's dive in.

Complaining In A Shared Channel

It's no mystery that this is a stressful career, and sometimes it's healthy to vent and let off some steam. When we internalize our frustrations, it can affect our mental state and our work performance, so being able to communicate frustrations and concerns is a healthy part of the job.

However, it is not ok to whine and complain to our coworkers on a chat app or email. If we are not accompanying our frustrations and concerns with proposed solutions that will help the project, we're giving everybody around us a reason to tune us out. They will stop listening to us and taking us seriously, and we will lose respect and credibility.

If we have frustrations or concerns and don't have a proposed solution, we need to write them down and save them for the next retrospective meeting, where we can add them to the "stop doing" column and let other people propose solutions. People will understand and respect our thoughts as long as the discussion is collaborative and the goal is to improve. The moment we come off as entitled and unappreciative, we will lose the crowd, and people will want to mute us.

Deflecting Blame

People love to point fingers and pretend they're infallible. When something goes wrong, and we're blamed for it, it's one thing to refute if we had nothing to do with the issue. If, however, we did have something to do with the issue, we need to own it fully.

Deflecting the blame is when devs say things like, "Sure, I broke the build, but if DevOps hadn't made the build so fragile, then my bug wouldn't have broken it," or like, "Yea, that's my bug, but I was waiting on Dave all sprint and was blocked by his ticket, so I only had a few hours to code it instead of all day."

Bringing other people down with us is highly cowardly, and it guarantees that those other people will remember what we did. Instead, **we should push them out of the line of fire and take all the blame ourselves**. That way, we allow everyone around us to respect our integrity and honorable intentions so they can remember us for that instead.

Everybody makes mistakes. Show your team that you're not afraid to recognize that **growth comes from failure**. Don't pass around blame; it's just a really amateur thing to do.

Public Shaming

If we combine deflecting blame and complaining, we get public shaming. When bringing up concerns to key stakeholders, **never single out a member of the team**. These should be private conversations kept internally to your close peers with the intention of either passing along a direct concern to that person's manager or to uncover a coaching and growth opportunity for the person in question.

If one person on the design team is being extremely difficult to work with for example, maybe by changing attached designs after devs begin work, we may have the urge to call that person out in a shared channel--don't do it. Keep it private. Send a direct message to the person expressing your concern, or even better leave a comment about your concern with the process on the ticket itself. **Don't make it personal, make it procedural.**

I don't have major issues with public accountability, but I believe that what goes around comes around. If you don't want to be called out publicly when you make mistakes, don't do it to other people and you won't plant the idea in their heads to return the favor.

Committing Sloppy Code

If you regularly force the rest of the dev team to find your errors and typos for you in your code reviews, you will most certainly notice your PRs sit unreviewed longer and longer because your team grew sick of doing your job for you.

Automated tools like linters, builds, and static analysis can help a ton but misspelled variables, inconsistent naming, logical bugs, or missing unit tests likely won't be caught by those tools. They'll instead be caught, *hopefully*, by the devs reviewing your code. Again, we all make mistakes, but a habit of them is called incompetence. Being junior and struggling with the code is one thing, but being lazy and sloppy is inconsiderate, and there's no excuse.

You're here, reading this book, so I know you want to be the best dev you can be. For that to happen, you need to hold yourself to a higher standard than anybody else in every area of your job. Of all the traits I'm listing in this chapter, committing sloppy code is the most common way devs get kicked off projects. The rest of us didn't have time to do our job *and* their job, so we replaced them with another dev who could do the job right the first time.

Making People Wait For You

This one is short and self-explanatory. People like to feel important, and making them wait for you does the opposite. Arrive on time to meetings, deliver tickets early, and understand that being on time is the bare minimum required not to lose respect.

Discussing Identity Politics

In our world, especially these days, identity politics is everywhere. Everybody has a group; if you say anything online, you make one group happy and another group angry. In a work setting, you're just asking for trouble by bringing up *any* identity politics.

I've personally witnessed a conversation at an office holiday party where some folks were joking about a colleague who came into the office a few weeks earlier wearing a wild and comical 2016 presidential candidate shirt, and his supervisor was in earshot and got legit mad and said "I don't think he can work with us anymore. I don't think I can have him on my team if he supports that candidate."

Crazy right?

Now, I won't say which presidential candidate was on the shirt because it honestly doesn't matter, and that's my point. Some of you are visualizing a Trump shirt, some of you are visualizing a Bernie shirt, some are visualizing a Hillary shirt, and so on. Whichever presidential candidate you have in your head directly coincides with whether or not you agree with the employee for wearing it or agree with his boss for wanting to let him go.

Identity politics puts people in these echo chambers where they only want to be around other people who share the exact same opinions. If you challenge any of their views with facts, they will argue about the source of the facts instead of considering the facts at face value. All this does is divide people and ruin relationships.

If you are serious about growing your career, you must draw a hard, non-negotiable line between your work and personal life. I'm not suggesting that any of you reading this book would flip out if you discovered your boss or coworker was part of another group. Still, it's a very real risk that you could jeopardize your career growth potential if your identity politics conflict with those around you.

Outside work, live your best life, and support whatever groups bring you joy. Our society may one day be mature enough to have rational conversations across identity lines, realizing that belonging to any particular group most likely won't affect your job performance. Still, what people don't know, they can't use against you.

Part Two

Practicals

CHAPTER SEVENTEEN

Comprehensive Analysis

When presented with a new project, we often want to understand what we're getting ourselves into and get a clear picture of the scope and state of things before putting pen to paper.

Typically this scenario pops up either when we are freelancing and need to bid on a project which has an existing presence or if we work for an agency and need to help the sales team understand the scope of the work so *they* can bid and eventually get an SoW in front of the prospect.

Bidding on greenfield projects is a very different approach; we won't cover that in this chapter. We typically have much more control over the stack and infrastructure and need to work with the client much more at the product level.

Bidding on projects with some established infrastructure presents many questions that need to be answered before you can even think about writing code.

- Are we replacing the existing product entirely?
- Are we extending the existing product?
- Are we making a new product that depends on the existing product?

- Are we simply upgrading the presentation layer of the existing product?
- Are we being asked to fix or improve part of the existing product's core infrastructure, like improving security or performance, for example?

These are some examples of questions that I would be asking which would lead to follow-up questions like whether or not we need to work on legacy code or if we're going to leverage APIs that talk to the legacy system, and which team would be responsible for developing those endpoints and payload contracts.

Ultimately we want to find out the underlying deliverable for which we would be assuming responsibility. We don't have to get this perfect. Still, we *do* need our SoW to be written in such a way that if the client suddenly expects us to take responsibility for legacy support, the scope and budget are adjusted so we are compensated for our time and materials.

When I do a comprehensive analysis for a new project, **I always assume the worst possible scenario**. A missed detail could end up costing us more money than we earn from the project. Things I want to understand first are typically:

- Which programming language or languages are we dealing with?
- Which frameworks or libraries are currently implemented? Have they been maintained, deprecated, or in need of updates?

We don't need to necessarily worry about upgrading these tools, and often it's better if we don't and leave them alone because there's a significant risk of introducing regressions. We don't want to deal with that if we don't have to.

- Is the code under version control? Does it use one of the

common ones like Git or Subversion, or is it some custom proprietary thing that will throw a giant learning curve wrench in the ramp-up? Is the code deployed to a self-hosted repo or in the cloud?

This is a big deal when it comes to developer provisioning. How quickly can we onboard new devs to the project and handle access management? Self-hosted repos often have to be provisioned internally from the client, so we must be aware of process bottlenecks.

- Is there a content management system (CMS) in place? How is it deployed and managed? Does it have an edit history, or is there a risk of devs clobbering data while we test and build? Is there a sandbox tier, or can we run the CMS locally even? Does the CMS offer an API or generate static files on disk?

A giant monolith CMS can create development nightmares because replicating the local dev environment becomes extremely difficult. Often we're stuck working with what the client has, so knowing this upfront can help set realistic expectations with prospects.

- Does the CMS provider offer a docker container that can be run locally? Does it need a license, or can we run it for free?
- Regardless of whether there is a CMS, can we run the existing product locally and offline? If not, do we need to be on a VPN to work?

This can create challenges when we have offshore devs on our team because often, client SecOps put restrictions on onshore IP access only. This may require offshore devs to daisy-chain VPNs and can really slow down deliverables.

- What databases are in play? Are they SQL? NoSQL? Graph? Is there a data warehouse, and do we need to care about it, or are the backups automated? Is there an ORM being used or raw queries?

* * *

- Are devs allowed to write queries, or do we need to go through a DBA? Is the existing product using database migrations, or how are devs expected to get database table alterations and schema changes deployed?
- Is the database using any replication? Like CouchDB for a mobile app? Is the existing product a big data app with hundreds of billions of records, and can we work locally with a significantly smaller subset of the data?
- Is there a cache layer? What kind of cache is being used? An in-memory database like Memcached or Redis? A file cache? Waterfall cache? Are devs able to purge cache, and do we have insight on cache expiration?

A cache is excellent for end-user experience and keeping database costs down, but it creates development nightmares when we want to troubleshoot things rapidly. We must understand what cache is in play and how much control we have.

- Which keys are being used in the system? SSH keys, API keys, and so on. Would each developer need their own keys, or would we share keys? How are keys managed so they remain secret? For example, would our code need to make calls to AWS Secrets Manager or something, or can we blindly expect keys to be provided from DevOps through environment variables? Are keys rotated or are they persistent? How should we fetch and refresh keys? Can we subscribe to a refresh event or do we need to poll the secrets store?
- Are devs able to remotely connect to deployed instances of servers or databases using SSH or other credentials? If not, how are we expected to troubleshoot issues remotely?
- What kind of logging is being used? Are we allowed to connect to a remote logging service like Splunk or an ELK stack? Does the existing product do trace logging and round-

trip logging? Is there any instrumentation in place like New Relic that would complement the logs?

- How does the existing product handle permissions? Does it use RBAC or another approach? How are roles and permissions created and managed? Are they managed externally through an admin interface, or is it all baked into the code and managed by devs?
- How does the existing product handle authentication? Does it use JWT or Cookies for sessions? Does it use OAuth or basic auth? Does it use SAML or OICD for single sign-on?

We always want to understand authentication for apps to assess if our local dev environment can leverage the same authentication strategy if possible. Often I'll see projects where the auth gate only exists on prod--devs and internal staff get a free pass to a non-authenticated experience. This is ok if the auth gate is its own deployed app and is managed by DevOps and SecOps. They can just put the gate in front of whatever URLs they want to and use middleware to check to see whether deployed apps have been gated. However, we still want to understand what's currently in place and whether or not we need to accommodate it.

The sample comprehensive analysis I shared above is a guide. Every project is different, so the research for your project may vary. This type of analysis works well for most web projects, but projects in other tech verticals like games or IoT will require additional inquiry. We will want to know if the game supports multiple platforms. We will need to determine if we're expected to build or help finish a new game, or work on DLC or patch releases for an existing game, for example. An IoT project opens up even further inquiries, like if we will be issued a developer's kit or if we're expected to ship hardware back and forth as firmware requirements change.

Not all of this knowledge needs to be obtained before bidding on a project, but the analysis can help us decide how to structure our SoW. Typically for freelance projects, I'll sell an initial 8-hour comprehensive

analysis SoW--one day to deep dive into their system and ask all these questions I just mentioned. The more unknowns that come from the analysis, the more I'll lean toward time and materials (T&M). If the analysis proves that the project is very straightforward, then I'll consider a fixed bid with a clause that if the scope increases, so will the invoices. I've worked on a project where a competitor charged \$200,000 for a three-week comprehensive analysis. Two hundred grand just to get a quote! **Always charge for the time you spend on someone else's project.** You don't have to be writing code to be spending time.

If they don't want to pay me for my time to do the comprehensive analysis, I typically move on because my time is precious. For huge clients like Salesforce or Google, we may want to do a comprehensive analysis on credit because we want to get our foot in the door as quickly as possible. I don't typically recommend ever doing any assessment work for free. Still, depending on the prospect, it may be wise to take a loss if we can use them on our portfolio to gain additional work in the future.

If you work for an agency, you'll likely do these kinds of deep dives regularly, and who cares if the prospect pays your agency or not because the agency will pay you either way. There's a lot more pressure, though, because the agency will depend on you to expose as much potential profit leak as possible before they sign an SoW.

Your homework for this chapter is to take the list of questions I just covered and answer them for your current project. Don't just answer them from memory. Pretend you were just granted access--where would you need to look to find the answers? Who would you need to ask? Who would you need to ask to learn who *else* you would need to ask?

Don't spend an entire day on this, obviously but feel out the steps involved in doing a deep dive on your own project and see if there's anything I listed that you don't yet know. This could be an excellent opportunity for you to improve your overall understanding of your project.

CHAPTER EIGHTEEN

Security

Security is the most crucial aspect of any project we work on. I want to list three laws I live by when securing my projects. Live by these three laws, and you will give security the attention it deserves.

Law 1

Always assume your users are smarter than you.

I see this time and time again on projects. The application will have a security exploit, and the dev team will brush it off as a low risk because the only way the exploit can be achieved is if a user knows how to do something clever. Some of the best hackers in the world are among the most intelligent people in the world. The moment you think your site is "secure enough" is when you should obtain a security consulting firm to run penetration tests on your system.

Law 2

If your project has a back door,
assume everybody knows about it.

Obfuscation is not security. Just because your application has a big sign that reads "enter here" doesn't mean everybody will do that. If your website has a query string parameter `?admin=true`, which will bypass the login, then you need to act as if that is in the public-issued docs and everybody knows about it.

People leave companies, people talk, and people get hacked. One example might be, imagine some contractor working on your project upgraded their hardware and reissued their laptop to another contractor, *not* on your project. There's a risk now that the new contractor will have access to the browser history on that laptop and see the back door. You can't assume this new contractor won't snoop or care. You can't assume the contracting firm will wipe their laptop disks when reassigning them. You need to consider the worst scenario.

Law 3

Always assume if your application is breached,
that you will lose everything.

Seemingly innocent oversights have brought down entire companies before. Even if there was a breach and no data was compromised, the breach still needs to be disclosed to customers. That could result in millions of dollars lost simply because customers lose trust and decide to put their money elsewhere.

Live By These Laws

So I want to reassure you of something now. Not only do we *not* need to be security experts to be able to secure our project, but we should stay somewhat ignorant or remain humble as we learn and improve. There are entire companies specializing in security, and it's their job to keep up with all the latest exploits. It's almost impossible to keep up with every security precaution and best practice if we're full-time developers.

So while we don't have to know about all the exploits personally, we *do* have to do everything in our power to ensure our system can be audited over and over for security leaks, and we *do* need to have security on our mind, front-and-center, throughout the entirety of building and running our project. If a breach happens, we need to show that we covered all our bases and followed all the recommended guidelines and best practices as directed by SecOps and any external security firms doing the audits. We do **not** want to reassure the executives that all our security precautions were enough and that their system is secure unless we are *on* the SecOps team and our job is literally securing the application and the network. Even still, why own that risk? Obtain a security consulting firm to run audits and let them make the claim.

We as software professionals should be humble and untrusting of any security precautions in our system. We want that executive all-clear to come from a multi-million dollar security firm, so if the project gets breached, the blame stays high at the executive level. Ironically, having said all that, I've been on *several* projects where the SecOps team really didn't know what they were doing, and the pen testers weren't really penetrating anything exposed to the public.

For example, one complained about a plaintext password in my spec files. The culprit was nothing more than a literal string to create a temporary test user that was immediately deleted when the subsequent spec ran. I'll cover this example deeper later in the book.

Not all security analysis software is intelligent, and often you'll feel more competent than the security auditor. Still, it's not the ones and

zeros that matter here. It's corporate and customer data, intellectual property, and system credentials. Losing control of any of it simply can not happen.

So, while I recommend avoiding *confidence* in the security space, I still want you to have *competence*. So, what are some precautions we know we always need to take?

- **Short-lived tokens:** Tokens get passed around, and the longer they remain valid, the more they risk being intercepted and used by someone posing as you.
- **One-way hashes:** I highly recommend using one-way hashes that need to be recreated for validity confirmation instead of encrypted tokens that can be decrypted with a key. If anybody gets the key, our whole system is at risk. One typical example of this is user passwords. We should create and store a one-way hash for the password. The user will then provide plaintext to log in, which is then hashed, and we compare the hashes. If they match, the user is legit. We should also salt our hashes with a relatively high exponent factor to slow this process down. We want this step to be noticeably slow to help mitigate brute force attack risk.
- **Multi-factor authentication:** Stored one-way hash passwords are great but brute force attacks previously mentioned are extremely common. If the hacker can guess a password simply by trying hundreds of millions of strings, then how secure is our system really? Even if the speed of our hashes is curbed by salting, we can take additional steps to enhance security even further. MFA means that even if a hacker guesses a password, the owner of the password still needs to confirm both the login intent and a one-time short-lived token from a separate physical device.
- **Long-complicated passwords:** Make sure the app demands long, complex passwords from users. An 8-character password can be easily cracked, no matter if it contains special characters

and uppercase letters. A 60-character password suggests there might not be enough time in the universe to crack it. Last year (2021) I learned that the Wells Fargo home page login treated all passwords as lowercase and restricted the password length. Hence, my login was case-insensitive and easy for a hacker to guess eventually. A colleague told me, and I didn't believe them. Sure enough, I confirmed it with my own eyes. They seem to have fixed this bug in recent weeks, but never forget that even the most prominent companies you think would be ultra-secure are terrible with security. Be better.

- **Principle of Least Privilege:** In your system and your company, entities should only hold keys for locks that apply to them. For example, if you have twenty microservices, but only one connects to Amazon S3, then that's the **only** one that should have access to Amazon S3.

For engineering, the UI team does not need the production database password. Every entity that holds a key risks losing it, abusing it, or having it taken from them by an attacker. When we're dealing with other people's information, we need to assume the worst-case scenario--only give people access to what they need to do their jobs. Keep track of and reassess that access regularly, and only expose credentials and keys in environment variables to systems that require the keys to function correctly.

I'd like to talk about **authentication** vs. **authorization**. Authentication is essentially identity, and authorization is essentially permission. While they both live under the security umbrella, I often hear developers conflating the two, even though they are *very* different.

Setting your project up for authentication is essentially putting an identity gate between the user and the application. The user or service provides some form of identifier and credentials for proof. It could be a username and password combination or an SSH key. There are several ways to verify identity. Our system either looks up the agent based on the identifier and validates the proof or delegates that step to another

identity provider using OAuth or a similar approach.

Authentication is essentially going to a concert, showing your ticket to the bouncer, and in turn, they put a green wristband on you and let you in the door.

Authorization is essentially whether or not your wristband lets you go anywhere in the venue or only in some places. Maybe you try to get backstage with your green wristband, but a separate bouncer says, "Sorry, you need a VIP badge," for example.

So if an identity gate covers the basics for authentication, how do we handle authorization? Can't the login gate handle both? Yes, it can, but only if everybody using the application has the same permission.

This is a common pattern when you compartmentalize apps through subdomains or routes where each app has its own login. Maybe your website presents www.myapp.com to customers and an admin.myapp.com subdomain for employees. If you're developing a mobile app, you might have one dedicated to customers and another entirely dedicated to internal staff. Each of those apps can then only know about the APIs that apply to their respective users.

This follows the separation of concerns principle and makes authorization handling quite simple because you essentially only have to do one security check at the front gate.

If, on the other hand, you have an application that presents a common UI to users with varying permissions, then **you need to adopt roles and permissions into your identity layer so you can check them at the permission layer.**

In our concert analogy, maybe some folks have a green wristband, some have a Media Pass to take photos on the main floor, some have a VIP badge to go backstage, and so on. That venue can't just have one bouncer at the front door; it needs one at each designated area. These bouncers on the inside only care about your badge, not your ticket. Only the front-door bouncer cares about your ticket.

* * *

If you had to carry your ticket around the venue and prove its validity to each bouncer, it would not only make security extremely slow but also increase the likelihood that you drop or lose your ticket or that someone else steals it from you. The principle of least privilege tells us only the authentication gate knows about the password, not the authorization gates. In my projects, I typically use RBAC, which stands for role-based access control. RBAC is divided into four main data points: **roles**, **resources**, **role assignments**, and **permissions**.

Roles are essentially just a name for a typical position. I recommend avoiding using "admin" because, in RBAC, different roles can get permission to various resources, and permissions can cross over each other, so admin becomes a very ambiguous term; admin of *what*, for example?

I prefer using actual positions or a shared department role, like "Tech Lead," "Customer Support Manager," or "SecOps," for example. It makes it much easier to correlate who is most likely to need access to what based on their daily responsibilities.

Resources are entities in your application that need to be gated. Most of my project work is web development, so I consider a resource to be a combination of an HTTP request type and action. For example, if we have a RESTful API endpoint for `/user`, I typically define four resources, one for each `POST`, `PUT`, `GET`, and `DELETE`.

Identity and ownership can make RBAC challenging when used at the route level for APIs because they require additional middleware lookups. Still, it beats the alternative of adding overhead by including database records as resources. In other words, we may want to permit users to update their own profile but not everybody else's. Or maybe we want to give a user permission to edit *any* user's profile.

Defining the resource at the API route becomes slightly ambiguous unless we split the routes into well-defined administrative routes where it's clear that the route's intention is for the token holder to act on behalf of another user. For example, a `PUT` request to `/api/v1/user/:userId`

should only be allowed if the `userId` in the JWT matches the `:userId` URI param *or* if one of the roles in the JWT allows administrative permission.

We can simplify this by having two routes:

```
// 'user' role required, userId pulled from JWT. This means users can  
only ever update their own profile.  
PUT /api/v1/user  
  
// 'customer-service' role required, userId pulled from URI param. This  
means customer service can update any user's profile.  
PUT /api/v1/user/:userId
```

Role assignments are essentially just mappings of roles to users. This is many-to-many. Many users can be assigned a role, and a user can be given many roles. For example, maybe a user works in multiple departments, or perhaps numerous departments use the same endpoint in their workflow.

Permissions are essentially mappings of roles to resources. This is also many-to-many. Many resources can be assigned a role, and a resource can be given many roles.

I mentioned the need for middleware^{*} on some of the routes briefly. This is just a pattern preference and is ultimately up to you. For the concert venue analogy, this is a matter of deciding whether that backstage bouncer is outside at the back door checking tickets or inside the venue near the side of the stage checking wristbands.

Most of my examples come from a bias toward web development, but behind the scenes, it's really all the same when we consider desktop applications, game dev, mobile dev, and so on. Define and expose the entry point for users, validate them, and assign them a badge with some roles attached to it. Then, various actions in your application that should only be exposed to those roles should do those role checks there. Try not to reference specific roles throughout your application logic, or

* Middleware is a programming pattern that allows for the incorporation of various cross-cutting concerns (e.g. logging, authentication, compression) into a codebase with minimal modification to code points.

you will clutter your code with security checks. This is why I use middleware. If you are using object-oriented programming, you can use a superclass that does some security checks behind the scenes, for example.

Ultimately, security starts with us, so lead by example. Most of the time, computers are hacked not by some elaborate spyware but because people write their password on a post-it note next to their computer. I personally use 1Password to store all my online credentials. Most of my passwords are 50-60 characters of random gibberish, and it's fun watching my kids spend fifteen minutes entering the PSN password with a Playstation controller when they try to buy a game.

I also use multi-factor auth for *everything*. MFA is a no-brainer, folks. It's the difference between a hacker stealing my identity from a website and a hacker telling the website to send me a notification telling me to change my password because it's been compromised.

Junior devs on our team will notice when we put security first in our lives and our projects, and it's likely that our good habits will rub off on them. **Leading by example is a much better approach to most things than simply lecturing and nagging.**

To summarize this chapter, always be thinking twenty steps ahead when we code. Assume every user is trying to break our application and steal what's behind the curtain. If that happens, assume our entire career is finished, and we must apologize to everybody whose identity was stolen personally. With that paranoia in our heads, we can be smart and defer responsibility to big security firms and the SecOps team. Their job is literally to sign off on security. Let them handle it so we can do everything we can to *support* them and go above and beyond their bare minimum requirements. Remember, *we* still have to do the work, but *they* need to grade it before it goes live.

CHAPTER NINETEEN

Privacy

While most customer privacy is at the mercy of the product and marketing teams (like whether or not to sell data or how much personal information we need to capture in forms), the engineering team can take extra precautions to protect privacy in the event of a data breach. The two precautions I want to cover in this chapter are **redaction** and the **principle of least privilege**.

We already briefly covered the principle of least privilege in the "Security" chapter, so how does it apply to privacy? In the same way that we don't want to hand out database passwords to devs who don't ever need to run database queries, we also don't want to expose customer information to people internally who don't need the information.

If our application logs user records as they are created, then the logs are potentially a data gold mine. If anybody ever got their hands on the logs, they could scrape and collect thousands of email addresses, phone numbers, and whatever else is being logged.

I don't want to imply that the people internal to your project would steal customer data, but if **they have the customer data, they could lose it**. I've witnessed people working on the train have their laptops stolen from them in a flash robbery. Someone grabbed a laptop and ran off the train as the doors closed. If the victim was a software engineer and had

a local production data dump or log dump on the laptop, just imagine how much customer information is instantly at risk. I don't remember if the robber even closed the laptop, so the OS password was already bypassed. You could have all the security in the world on your production app, but if it's taken from a dev, it's gone all the same.

As senior engineers, we must actively keep data safe and secure at all times. Simply put, **if it's not our personal data, we're in no position to put it at risk**. We need to be voicing our concerns and protecting the data. Junior devs don't understand the risks yet, and executives don't understand the technology.

Sometimes, we have to log stuff or expose data publicly. For example, personal profile information stored in a JSON web token (JWT) is not secure. We have to assume it will get intercepted and logged by our ISP or DNS servers or at any point along the request's journey. Even if we're on HTTPS, JWTs are often attached to query strings, so they're not in the encrypted payload. They're encoded, not encrypted, and decoding them is no more complicated than pasting them on jwt.io and seeing the contents.

Knowing this, we need to **redact all the data not critical to identity**. It may be convenient to store a user's address and phone number in their JWT profile because that information may sit nicely next to their name and email address in the database. But if the address and phone number are never checked in the auth gates, redact them--omit them entirely from the JWT.

In most of my web applications, I use an MVC mindset, so I typically do the redaction at the controller level because I want the models to be raw behind the scenes. I have a function called `omitPrivateFields` which I put as the last step of every controller because this is the response that will be sent back to my route.

I don't do the redaction in the route itself because if my controller happens to be used for multiple routes, I don't want to burden each of them with the same responsibility. If a new route is created and the dev forgets to redact, then we have a security hole. One could argue a route

could bypass a controller and leverage the model directly, but that's a major divergence from the established paradigm and we'd likely notice that in our code reviews and integration test reports.

In my application's primary config file, I typically create a dictionary of all the private fields associated with the project overall, and then a dictionary for each model. My `omitPrivateFields` function will concatenate these and augment model-specific private fields onto the default list at runtime. Typical private fields for a user might be "password," "ssn," or "token." Standard private fields for an order might be "card-number," where you might want to return the last four digits used but not the whole number. You can see how this would scale.

It's important to appreciate that these patterns shouldn't be limited to any stack or application. Almost all applications have data and UI layers, even if the UI is a CLI. Which data gets presented to which user can absolutely be controlled by you, regardless if you're building a website, a video game, or a promotional kiosk.

Lastly, I'd be remiss not to emphasize once more that we *never* want to log plaintext passwords or store them in a token. It's bad enough that I still see databases storing plaintext passwords, but exposing them in the logs or tokens is even worse. You're just asking to get hacked.

CHAPTER TWENTY

Stability

Facebook's former mantra, "Move fast and break things," is easily some of the most selfish and irresponsible advice I've heard in this industry, but it does have a place where we can apply it to our benefit; it's an effective way to learn a new language or technology. I do it all the time with new languages and frameworks. I'll intentionally veer from the happy path so I get comfortable with error states and failures and then try to fix them from memory, reversing the steps I took to break them.

The moment I use that technology on a project with people besides myself, I'm no longer in a position to learn through intentional failure. It puts the entire project at risk, causes frustration and confusion, and wastes the time of colleagues and customers. **Stability in our application equates to predictability in our application.** Being able to predict how the app is supposed to work makes it easier to recognize when the application is not working as intended. Suppose sometimes the application throws an error, and other times it works fine. In that case, it becomes a quirk in the system--people don't know how to reproduce it, so they can't fix it. Entire days can be lost trying to recreate a scenario when someone experiences a quirk.

Stability comes from a variety of factors, including, but not limited to:

- Unit test coverage.

- Developer discipline, meaning using best practices and not cutting corners with hacks and patches.
- Thoughtful implementation, such as leveraging strategy-based development versus condition-based development.

Predictability in our projects allows us to refactor large portions of our codebase because we should have unit test coverage to tell us if we broke something. It also makes it easier to A/B test. Stability in our project can also be less literal and more meta, like our APIs, for example. Versioning and documenting our APIs allows us to create payload contracts, which, in turn, support integration testing efforts and cross-team collaboration. Any time we break our APIs, we want it to be intended, documented, and ideally scheduled for a few versions after a deprecation notice has been added to the docs.

The opposite of stability is volatility, which creates uncertainty. Nobody enjoys using a volatile application. One that crashes all the time or throws errors occasionally during typical use. Nobody enjoys working with an API that changes its payload or response schemas on a whim. All of this causes disruption, which causes frustration. Our objective as senior developers is to help alleviate frustration on our projects. We're supposed to be dependable and help deliver clean, consistent code so that our clients or employers can spend more time focusing on their business growth and less time on their business maintenance.

Keep this in mind when you are working with a legacy system. Clients and employers depend on legacy systems to run exactly as they are, and if you fix some bugs, you'll be a hero. The moment you change or break one thing, though, it'll feel like everybody on the team will notice. If you are tasked with replacing a legacy system with a new one, **prioritize stability over velocity**.

Often, these clients do not rush to switch to the new system. If the new system has any bugs or quirks that the old system doesn't have, you will feel immediate pushback on the new system. Suppose you break the old system while building the new one, even worse. Trust me, stability, stability, stability.

Often, the product team overloads the dev team and insists that deadlines are met without tests and QA. If you have to cut corners to deliver something on time, make a technical debt ticket to refactor or add test coverage at the earliest opportunity. If you are on a team that never pulls tech debt tickets into the sprint and regularly forces devs to cut corners to deliver, I recommend looking in the mirror and asking yourself if this is the team you want to be on. Ask yourself, "If I lead this engineering team, will I have any authority over the product team to handle technical debt or reduce sprint load to improve the engineering process?" If the answers are no, I recommend you move on to another project where the engineering team is more appreciated and respected.

Depending on your project, you may be using third-party dependencies. Often on websites, 80-90% or more of the application's code is third-party source code. Given this, even if we have 100% unit test coverage on our own code, if it ultimately only makes up 10% of the code that is being executed at runtime, then we need to do some serious due diligence when selecting and maintaining our dependencies.

I almost *always* pick the LTS release of whatever library or framework I'm considering. For example, **odd versions** of NodeJS are volatile, and **even versions** are stable. So when I was running Node 10 in my app, not only did I wait for Node 12 to upgrade, but I went so far as to wait until Node released its deprecation notice of 10, so by then, Node 12 was like 12.15.

Most of the time, the latest and greatest features in a dependency are nice-to-have and won't offer any real value to your application. This can easily be proven by rebuilding a simple app in multiple languages and frameworks. I'm confident that I can rebuild my current project in PHP 5 with patterns I used 15 years ago.

The back-end developers usually understand the risks better and are more in line with the slow-and-steady approach to development. In contrast, front-end devs often want to use the latest bells and whistles to keep up with browser updates or mobile UI updates. This is an extremely risky mindset because not only are we essentially using our

customers as guinea pigs to try out new libraries, but the view layer on most applications is most often the most variable. While we can control the version and state of our backend layer, different users have different versions of browsers and operating systems installed. Different screen sizes, varying devices being used to view our application, and so on.

When we roll out new code, we need to assess whether backward compatibility needs to be considered. We must have conversations with the product teams before leveraging technology that will only be supported by a fragment of our user pool. Always remember that we don't have to make all these decisions - it's often best to present all the options, consult the pros and cons of each, and let the key stakeholders on the projects make the decisions.

Each team has its purpose, and engineering's goal should be to implement the product team's vision as accurately as possible, where the highest majority of work is on features, not bugs.

CHAPTER TWENTY-ONE

Scalability

First off, what is scalability? Well, it's essentially how well our application can accommodate growth. There are several areas of growth to consider as we are architecting and engineering our projects. It's important to understand our market cap and load potential early on, so we don't waste hours unnecessarily over-engineering or over-architecting.

So what are some examples of growth?

Records

Imagine you are building an internal application for companies that will deploy your app instance inside their own data center. You can infer that the user cap will likely be the number of employees in the company--past, future, and present potentially.

Walmart is on record for having the most employees worldwide, at 2,200,000 as of this book writing, which is technically not that bad. Any mid-tier database like MySQL or PostgreSQL can handle a few million rows in a table with no problem, so your bottleneck will be the data transfer and display.

We wouldn't want to select all 2 million users at once, and we certainly wouldn't want to *display* all 2 million users at once on the web UI because we'd crash the browser.

If we knew from day one that we were building our project to support 2 million users, we'd most likely accommodate all this *from day one*. Simply adding pagination and ensuring the `users` table has the appropriate indexes solves this problem. We could even use the database for fuzzy text search and get near real-time autocomplete on 2 million rows with MySQL.

Let's imagine for a minute that we had built our app for local businesses with 50 to 100 employees and didn't need to add any pagination on the user management screen. The app works fine for months, and we start growing our business and securing larger clients, such as a health club chain with 30,000 employees. Our user management screen freezes up now and offers a terrible user experience, so we have to rebuild that page to support things we discussed, like pagination and indexing.

This is a very common scenario--if it's not users, it's orders or a ledger. Any time we collect data over time and display a list of data, our app is growing, and it should accommodate this growth by defining the UI and API capacity early on. If those capacities exceed our projected stored data capacity, we shouldn't need to adjust for growth.

In our list scenario, pretend we're making a music library application. A safe growth assumption might be that our users could eventually have millions of songs in their library but only a handful of playlists. We could decide if we want to keep the playlist feature simple or if we want to leverage patterns and consistency and enable pagination for even small data sets.

Neither is a better approach, though I'd recommend being consistent because redundant consistency rarely hurts the app. Typically, the product team will own these decisions on the UI and UX level, but we need to think ahead at the API and database levels.

Let's shift our perspective now and imagine that this application isn't deployed to corporate data centers but is a multi-tenancy application, so it needs to accommodate *all* company employees in the same database. Our growth potential just skyrocketed.

We don't need to look any further than LinkedIn, which has over 750,000,000 users at the time of this book writing. The game changes dramatically when we start getting close to or exceeding billions of records. If our application grew from even 2,200,000 users to 750,000,000 users, our scaling decisions aren't narrowed down to things like pagination. We must start considering data migration to an enterprise database or another database engine.

Migrating from PostgreSQL to Oracle isn't the end of the world, and I wouldn't ever recommend Oracle on smaller systems anyhow. Still, we need to have it in the back of our heads whether or not the migration is inevitable so that we can make decisions early on to support this migration.

For example, if we are using an ORM that doesn't offer Oracle support, we'll need to refactor a large portion of our code when we migrate, which adds *considerable* risk to the stability of our project. If we are using raw SQL for our queries, we'll most likely need to refactor those queries to change a few keywords and syntax patterns to be Oracle-compliant.

* * *

It's often ok and best to punt these decisions; rest assured, we won't be alone in the room making them. Migrations and refactors like this take months and involve multiple teams. Just make sure that the multi-team, several-month migration effort isn't required because of something we missed.

If we went down some rabbit hole with our code and architected the project into a corner to support a record cap of 15% of the project's growth potential, then our product and executive teams would not be happy with us.

Network Load

The next area of growth we need to account for in our projects is network load. I don't mean load at our ISP or in DNS, but once the packet lands on our server, does it need to wait in line, or can it be processed immediately?

Ultimately we're talking about user concurrency--how many users can engage with our application at the same time? This is a typical challenge for websites and online games, and the simplest way to support this is load balancing.

It's funny how long it took websites to catch on to this concept because even vintage games in the late 1990s, like Ultima Online and EverQuest, had multiple servers. Each server had a player limit. Once a server started reaching that limit, the game companies would deploy a new server with a new name and encourage players to either migrate their characters over to the new server or encourage new players to sign up on the new server.

In the web world, it's much simpler for the end-user because the load balancing happens behind the scenes. Most of you reading this book likely work for a company with a product deployed somewhere in the cloud. AWS, Heroku, DigitalOcean, Azure, GCP, etc. Even with these services, it's possible to bottleneck your application, so it's important to understand best practices and the challenges you're trying to solve.

You could spin up a DigitalOcean droplet or an AWS EC2 instance and run your entire website on it, and for smaller projects or static corporate landing pages, this is often all you need. If you have a SaaS application, you must assume that too many requests will overload one instance. Even servers like Nginx will bottleneck with too many requests, and often your database will bottleneck with too many connections.

It's important to define early on how many concurrent requests are allowed and how we will accommodate several times that. AWS offers a

service called ELB, which stands for "elastic load balancer." We can create an auto-scaling group so that as our website receives more traffic throughout the day, AWS will automatically spin up more instances of our app, and as traffic decreases, it'll tear them down to save us money. This mindset is fantastic; all we need to do on our end is write our code so that it can be part of a cluster.

For example, a monolith website on a single server can use sessions in memory and have an internal pub-sub mechanism because everything is tightly coupled. When we spread our code across multiple servers, we can no longer have state persistence on any one instance. A typical user making two requests might hit two instances of the API, and instance **two** won't know that the previous request hit instance **one**.

This is important to consider when handling authentication, authorization, and state. I typically solve this by putting state outside of the cluster, for example, in ElastiCache or on a separate Redis instance for smaller projects. Even working locally, I always assume my code can't own the state and might need to be deployed as part of a cluster behind a load balancer. Deferring ownership of the state allows me to scale my application and my state mechanism independently.

If this all sounds complicated or overwhelming, I recommend reaching out to someone on your DevOps team and chatting with them. Ask them how you could make their lives easier with your code, or frame it another way and ask them how the application is currently making their job difficult, and there's an excellent chance they will list off some things in line with what we're discussing now.

System Load

The next area of growth is system load. A lot of this will be solved by the load balancing approach from the previous topic, but it's common for data computation to slow down over time as data accumulates and your server is doing more things. Generally speaking, it's better to scale horizontally than vertically, spreading the load out among more processors instead of making one faster.

It's not always reasonable to keep spinning up EC2 or ECS instances when you max out your CPU. Sometimes we want to identify our CPU bottlenecks and see if we can extract the processing for this one thing entirely out of our server, for example, in a Lambda. Suppose we can defer this particular step to a third-party service completely; even better.

A typical example of this could be a virus scan. If every time people upload files, we need to scan them for viruses before we store them, that can create a bottleneck. We want this step to be separate from the uploader, typically as part of a pipeline. If your project's core competency is not virus scanning, find some service that can pipe the file through an OPSWAT instance. Whichever way we abstract this computational load, we want to use a non-blocking flow. Instead of forcing the user to sit there watching a spinner, we return some messaging to the user immediately, letting them know the process is happening, and we will notify them when we're done.

Features

As projects grow, the product team will start wanting to add new features, and it might be up to you to decide where to put them in the code. If we use our previous example, imagine you had created an "image uploader" feature, and then the product team tells you they want it to support video upload as well.

Suppose the uploader was coded very specifically to images. In that case, you'll need to have a bunch of redundant code also for videos, or you'll need to refactor the image uploader to be a general media uploader where the type and properties are passed to it based on the file being selected for upload.

Maybe once it's uploaded, some code generates a thumbnail. Well, that code won't work the same on a video. You probably need to first find a frame in the video or allow the user to choose a frame, then your code will extract it and potentially resize it the same way you resize image thumbnails.

There will be a utility logic fork at this step. You can see how adding a second media type caused a lot of change requirements. Imagine two months from now, the product also wants to support uploading PDF files. How will your code accommodate the new capability?

If you keep having to refactor this code, then it's not scalable. In OOP, you should make an interface and create separate classes for each type of file that needs to be uniquely handled. In FP, you'd want to leverage the strategy design pattern (refer to the "Becoming The Fixer" chapter).

Assets

Static assets are files that exist on the system regardless if anybody requests them and are delivered the same every time. Because they don't require any code or processing when fetched, they don't need to live alongside your codebase, and we can use this to our advantage.

The most common static assets are typically the images that make up your user interface, be it a website, a game, a mobile app, etc. Some other examples are javascript files to be run on the browser, a background music sound file, an intro video, legal documents, or a pdf of your press kit.

For remotely-fetched static assets, like images on a website, we want to consider a couple of things when deciding where they should reside and how they should be delivered. Ideally, we want the assets to live off the application server so the server can focus all of its power on processing code and handling network traffic to internal subsystems like a database, a message bus, various microservices, or a cache. We also want to keep the overall size of our application servers as trim and tidy as possible so that managing deploys and containers is faster and simpler. For example, we may need fifteen instances of our application server to handle high traffic at some point, but we likely don't need fifteen copies of our logo.

A common approach is to put all the assets in a CDN. Use a storage service like Amazon S3 and then put some access rules in front of it through something like Cloudfront. Alternatives like Cloudflare also offer static uploads for images and video, which can be accessed through an API. Wherever we put the assets, we want to leverage a highly trusted domain that doesn't send cookies along with the binary. In our system, then, we simply store a reference to the remote asset.

I highly recommend you keep proprietary terminology out of the references. Store just the file name instead of the fully qualified URL to its location in Amazon S3, for example, so that the files can be uploaded anywhere with the same name, and we can infer its location through

config or environment variables and strategies. For assets, try to land where if you change your mind and want to move your assets, it won't require you to change thousands of records in a database or thousands of lines of code.

Engineering

As projects grow in features and complexity, so typically does the need to expand the dev team to accommodate all the new work. I've seen teams grow from three devs to fifteen devs in a matter of a month, and it was basically up to those three devs to wrangle in the other twelve to make sure it didn't turn into the wild west. As senior devs, we *need* to make sure the codebase is set up in such a way that it tells us exactly how it wants to look and behave. We must set a crystal clear expectation with new devs that the ultimate goal for the project should be that **we can open up any random file and have no idea who authored it**. If you find yourself saying, "This looks like Dale's code; he's the only one using `for` loops," then Dale is hurting the project.

Let me give you a loaded example. Imagine you're pulled into a project as a new dev. You notice two files. One is `get_users.js`, and the other is `getOrders.js`.

Imagine `get_users.js` uses a promise chain for its asynchronous data retrieval, which happens to be a SQL query to a PostgreSQL database. The promise chain includes a `catch` handler and returns a system-specified static record-not-found error referenced from a constant.

The other file, `getOrders.js`, uses `async/await` instead of a promise chain and uses an ORM called Sequelize to fetch data from the PostgreSQL database. There is no `try/catch` block, so any error returned is the Sequelize error.

You're given a ticket that reads, "Make an API to get products from Postgres." Well, what do you do? Do you copy the style of `get_users.js` or `getOrders.js`? Do you ignore both and make your own, which could be some hybrid of the two or entirely original? Do you embrace the chaos and name your file `get-products.js`?

My point here is that the convention hasn't been established, so there is no clear direction for the implementation details. Tools like linters and static analysis can take care of a significant portion of your coding

conventions, so if you don't want to carry the burden of enforcing the rules, try to automate as much as possible as early as possible. Establish the rules, patterns, and standards before building any application logic. The project will be able to scale, grow, pivot, and typically out-pace the product roadmap, thereby boosting dev morale and reducing dev stress.

CHAPTER TWENTY-TWO

Testing

In this chapter, we'll talk about different ways to test your application and some patterns that can both accommodate 100% code coverage efforts and genuinely make the task easier for all the devs on the project.

Before we get into the weeds here, there's a fundamental mindset that I want you to adopt, which is *testing embraces the separation of concerns principle*. If you want your project to be easily testable, I suggest you support dependency injection at the unit level. Parameterizing your dependencies allows you to mock them out for tests *and* offer variability in production--cue the strategy pattern.

There are multiple ways to test your systems, and different types of tests are more appropriate than others at different levels of your infrastructure. Ultimately, **tests are there to solidify the bond between intent and execution**. If you look at a random file in your project, the code may be working exactly as it's written, but a test will assert that the code is working strictly as it's *intended* to work.

Unit Tests

Let's start with unit tests because they're the lowest level of tests I will cover, and we need to build a solid foundation. I first started writing unit tests for my PHP projects in 2003 using a tool called PHPUnit on a Zend MVC application.

I had been avoiding unit tests for a few years mainly because I was the sole PHP developer on most of my projects and had an impression in my mind that unit tests typically only offered value for shipped software like video games or desktop applications, where anything printed to an installation CD or DVD needed to work for everyone who installed the software.

Websites seemed more fluid, and I had adopted the belief that as long as I was willing and able to respond quickly to bugs, I could fix them on the fly to keep customers and users happy.

I joined a project using Hudson for continuous integration, which was still a relatively new concept in web development. One of the steps in the pipeline was being added to run unit tests on some of the core, cross-vertical code. The code was intended to be leveraged by several web applications in the company, so we all agreed this particular suite of code had substantial pressure to work as expected. Any bugs could have a catastrophic ripple effect throughout the business.

It didn't take me long to realize that writing tests for existing code, especially code authored by another developer, was clumsy, stressful, and irritating. It became evident that the code really needed to be considerate of runners like PHPUnit. When it wasn't, it was just a huge pain in the ass for all the engineers because we would have to refactor seemingly working logic simply to get the code to adhere to the CI rules.

I feel blessed that, for me, the natural progression of optimization here was to write tests alongside the logic--sometimes even beforehand. Write a simple test, then write an expression in my application file that

would satisfy the test. At this stage in my career, I still wasn't doing stuff like this as a "best practice." I was doing it because it made my life easy, and none of the bug tickets I got ever seemed to apply to anything I authored or had corresponding tests. I don't think I was substantially a better developer than my peers. I just caught all my own bugs offline while I was writing tests. The tests caught my bugs, not QA, and not the users.

This was huge for me because I typically need to know something will not break or set somebody up for failure before I feel comfortable putting my name on it. I want to be known for helping people, not causing problems. Doing due diligence through these tests gave me a major confidence boost. I knew my code worked as intended, and I could point out risks and reassurances for new features; tested code was robust and safe, and untested code was uncertain and risky.

Uncertainty might be the number one underlying motivation for our projects needing unit tests. Tested code offers certainty, providing a whole suite of positive side effects--boosted confidence, velocity, morale, reduced stress, predictable roadmaps, and more.

Fast forward a couple of years, and I'm at a new company still working on PHP, but instead of Zend, we're using CodeIgniter. This was right around the time CodeIgniter 2.0 was released, and full disclosure, I haven't used the framework since this version, so hopefully, it's improved since my time with it. That said, it did a ton of stuff behind the scenes. So much so that if we wanted to test a controller, for example, we had to run through like 10,000 lines of core framework bootstrapping and boilerplate code to get the controller into a state where it could be tested.

So even if we did that and even if it worked, it significantly slowed down the test runner and made our tests look disgusting. I wished there was an elegant way to test these controllers outside CodeIgniter. Still, the way these frameworks were built makes it so you can't just copypasta a controller from one framework to another. They reference core framework constants and classes, and if you need to leverage those constants in your tests, you need to bootstrap the framework entirely *in*

the test.

This experience opened my eyes to the reality that frameworks can help or hurt you. They can cut 80% of your dev hours off the budget, or they can defer those hours into an unfortunate "framework accommodation" effort where you're ultimately trying to customize and configure to the point that you might as well roll your own framework that satisfies the business and call it a day. Frameworks don't offer the same wins that unit tests do. All they offer is perceived consistency and the idea that the dev team can focus entirely on application logic and get all the utility logic for free.

All tools have their pros and cons. The variability and volatility of any framework should be funneled through the certainty and reassurance that unit tests provide. In other words, if my code is thoroughly tested, I don't need to care what framework or language is behind it. I have my reassurance, and I can move on.

Unit tests should be as exhaustive as possible. It's common for any of my API platforms to have upwards of 2000 assertions. I want to assert as many likely failure scenarios as possible, typically with the database models. I often use runtime validation over static types because most of the data that runs through my applications comes from sources outside my control--users, third-party services, and so on. I assume the worst and most incompatible data will be thrown at my APIs, and I want my tests to offer me a level of certainty that all my validation is doing precisely what I want.

These tests aren't here to prove my validation works. The library I regularly use for my NodeJS projects has its own tests, so I don't need to re-test it. My tests essentially create a snapshot of the rules so that when the product team decides one day that they want to change what fields a user is going to send to our API, we can run the change through the tests. If the DevOps team wants us to upgrade our version of a particular tool or library, it's the same thing. We will see what things will break through the tests and can have a rational conversation with the product or DevOps team.

* * *

We can see if the regression is isolated to one model or if it will affect multiple places in the application and can provide a realistic set of expectations from the engineering perspective. Is this a **2-point** story or a **13-point** story that needs multiple devs on it? It may require a feature flag, documentation, a deprecation notice, etc.

In the "Functional Programming" chapter, we'll walk through an example of how to leverage dependency injection in a unit test.

Integration Tests

Integration tests are an additional level of tests that typically either hook into service connectors or leverage a network. For example, if we have an API, we may have a suite of unit tests that prove the handler code works as expected, but those tests live in a bubble and don't consider each other or any other API in a particular flow.

If we want assurance that our APIs are behaving as expected in the context of their respective role and residence in the architecture, we would also want a suite of integration tests that prove that we can interact with the API on the network as expected--considerate of headers, middleware, cookies, query strings, and so on.

These types of tests can range in complexity and scope. A quick start might be a simple health check endpoint that responds with a build number. Maybe it makes a simple query to the database and returns some expected data so we can verify that the code can both access the DB and also handle HTTP traffic. This could prove that our API server is up and running without having to test each endpoint separately.

The next level of testing might be a set of smoke screen tests that hit each API and only check for a 200 HTTP response. If the response payload is JSON, we can additionally confirm it is semantically valid. If all the JSON responses contain a consistent property, like "timestamp", we can test that the property is present. You can see how we can add layers of smoke screens. I recommend aligning your integration test suite with any documented schemas and behaviors attached to the release version.

Essentially the published API docs describe the intent, and the tests prove it. Using a tool like Postman to build out the API documentation, we can attach tests to the endpoints. Then with our CI pipeline, we can run through all the Postman tests to ensure nothing broke before deploying code changes to any APIs.

Most of my database model unit tests typically run SQL through an

actual database. I'll expose a local database server in a Docker container which my unit tests can use. I'll spin up, populate, test, and destroy the database on each spec. In the past, colleagues pointed out that those unit tests are better labeled as integration tests, but I see the database as just another engine that allows me to prove the query works.

I'm not testing that my code can connect to a database; I'm using the database to help my tests assert the queries are working. So even though the unit tests integrate with a different engine, it's good to differentiate whether you're leveraging an integration to test a unit or leveraging an integration to test the integration itself.

Functional / End-To-End Tests

Let's talk about functional testing. Unit tests give us confidence in our utility and application logic, and functional tests give us confidence in our user experience. Functional tests are often referred to as end-to-end (or E2E tests). However, many larger projects will split up those efforts, put functional tests inside "feature testing" terminology, and have separate teams handling end-to-end "acceptance testing." Testing the functionality of something becomes very subjective because of how inclusive or exclusive the acceptance criteria are for the test. In the previous example, where we might add a step to our CI pipeline to run through a bunch of API endpoints and test that they all work as expected, functional tests require another level of interactivity.

In the web development space, tools like Selenium can spin up a headless browser and, for each test, can click buttons, input data, and simulate a user going through a predetermined UX flow. These flows can be as simple as asserting that a user can load the homepage and see the login button. We can also create UX flows, like verifying the user can log in through a form, view and verify profile information, and successfully log out.

We can even get as granular as walking through an entire sales funnel--adding products to a shopping cart as a guest or a returning customer and completing a purchase using a fake credit card with a merchant sandbox. We can even verify that an order confirmation email was sent and can be viewed as expected.

We could test that products the system recognizes as "on-sale" are presented as such and, on the checkout page, are also calculated accordingly. We could test that different users from different zip codes are given accurate shipping and tax calculations depending on which products are purchased and which shipping carrier and shipping option are selected.

As our example e-commerce site evolves and new sales funnels are introduced and supported, we could add new functional tests to prove

each new funnel works. More importantly, we can prove that when we add new funnels, the existing ones continue to work as expected.

The takeaway here is that while adding this level of test coverage may sound intimidating or even overwhelming, the alternative is allowing the customers to experience the pain of bugs or a quirky UX. Even if the customer reaches out to you to let you know, you'll be in a terrible position by then. The customer will likely be upset and have lost some trust in your company, possibly even to the point that they won't want to do business with you anymore and will tell others to shop with a competitor instead of you.

You'll be put in a position where you'll have to try to reproduce the customer's issue manually. If you *can* reproduce it, you'll be playing catch-up and apologizing, potentially offering credits or refunds. If you *can't* reproduce it, you'll be in an awkward situation where you'll have to choose whether to treat your customer as an extended QA staff or pretend it's a one-off issue and hope no other customers experience it.

It is absolutely imperative that we catch as many of these issues as we can before the customer does. As software professionals, it will likely be up to *us* to insist on the test coverage for our employer's or client's project to help *them* save face and keep their business successful.

Functional tests also play a critical role because of the level at which they reside. We can prove that our application supports variability at the client layer. In the web space, that is the browser. We want to run through our functional test suite in various browsers and versions of each of those browsers, and even on different operating systems. This is a fantastic way to automate QA and generate bug reports for edge case scenarios in older or bleeding edge browsers. This process is typically very time-consuming for a person to do manually. In a game development space, you might run your tests on various gaming platforms like Steam or native OS with multiple OS versions or different consoles, mobile devices, and so on.

The fundamental values here are **reassurance** and **certainty** as we build and evolve our codebase and software offerings.

A/B Tests

The next type of testing I'd like to cover quickly is A/B testing. If you're new to A/B testing, it's a way to try out new features and let the data objectively decide which features become permanent.

A straightforward example of this could be a "buy now" button--should it be yellow or green? Well, let's A/B test it. We'll include both in our code, and half the users will see a yellow button, and the other half will see a green button. We will track which buttons are rendered and which buttons are clicked, and then at the end of a certain amount of time, we will look at the analytics data and objectively see which color performed better. Then, the better-performing color becomes the permanent color.

Excellent idea, so how do we do this? Well, you're in luck because we've solved this problem in an earlier chapter: **the strategy pattern!** Instead of making a `BuyNowButton` component that returns a fixed UI, imagine we can have the component accept a "strategy" param and provide it with "a" or "b" or whichever variables you use in your approach. The component pulls a child component from a nested strategies folder that matches the strategy and renders the correct UI to represent "a" or "b," respectively.

Your components can all inherit from a higher-order component* that does all the A/B analytics logic for you, so you don't need to code the interaction event monitoring every time. Maybe your components get some params for free like "`ab-interact-type`," where you could provide values such as `onClick` or `onBlur`, or something to that effect.

The objective here is that we don't want to litter our codebase with a bunch of A/B testing logic. Instead, set up patterns very early, so this all happens in the background. One approach is an API that provides a dictionary of all the component A/B test states. The product team can then enable A/B variability or force one strategy at the data level. Once

* A higher-order component is a function that takes a component and returns a new component

the product team decides, we won't have to go back through our codebase to hard-code "b" or delete the A/B strategy. We can put that effort into a technical debt cleanup ticket and do it later.

It would be best if we didn't have to deploy our project simply because the product team says, "The yellow button is performing better; let's show yellow for everybody." We should be able to update a value in the database, so `BuyNowButton` is set to "b," and the app renders it that way. We can get very granular with this, of course, and can infer things with class names, tags, ids, or however we want.

Maybe the A/B test is for all the "buy now" buttons, or perhaps it's only for the sale items. Maybe only the home page featured product button is eligible for the A/B test. Every project will have different needs, so the constant here is to get the pattern in place early so that when the product team makes a decision, it doesn't have to involve us.

Load Tests

The next type of testing I want to cover briefly is load testing or stress testing. Tools like Jmeter can spam our APIs with payloads to simulate traffic spikes to determine if our system can handle it or if we need additional consideration for those times. Load testing can help lure out race conditions or can help unveil pipelines that cause data loss when one step of the REST transaction fails. Typically these days, we deploy to the cloud in a cluster and trust the load balancer to do its job. Still, once we get past serving up static data and start handling thousands of concurrent requests, we want to see what will inevitably break and start formulating a plan.

For example, imagine the simulated traffic loads are reasonable in the near term. In that case, we likely want to refactor our models or tune our database. It may be as simple as adding some indexes or creating a view. Or we may find out that our choice of database is too small or slow for our product, and we need to change it completely. We could change the storage type from RBDMS to NoSQL or add a cache layer in front of it like Redis and serve up reads from that. If our simulated traffic loads are much greater than our marketing team is projecting for the near term, we can likely note this as a risk and deal with it later when we get closer to that potential reality.

The goal here is to experience the pain before the customers do so we can adjust or pivot without needing to apologize or ask the customers to accommodate our efforts.

Pen Tests

The last type of testing I want to cover is pen testing. I prefer to delegate all penetrative tests to SecOps or a hired security agency. Our goal as software engineers isn't to be masters of security because that's an entire field separate from ours. Our goal is to understand the most common types of attacks and vulnerabilities in our medium, whether mobile device operating systems, web browsers, game consoles, or IoT.

Different clients will interact with our software differently and present a range of vulnerabilities. While we can't reasonably keep up with every vulnerability in every medium, we *can* architect our projects in ways that make pen testing easy and convenient for SecOps. We want them to hammer our application with as many crazy edge cases as possible. We want to pass with flying colors and get an A+ grade for security, and we want our code to be structured in a way that allows us to quickly detect and patch or refactor vulnerable code without breaking the rest of the system.

We want the ability to swap out any vendor that may have a vulnerability in their API or library with an alternative that may be more stable or secure on a whim, with the least amount of pain for the developer. We want to adopt that mindset internally in our applications as well so that we can swap out vulnerable expressions with secure expressions, and the rest of the app isn't drastically affected.

This is easier said than done, but patterns evolve as we experience the pain of failing. We start remembering more about what *not* to code while we're coding. Eventually, getting it right the first time becomes natural, but keeping up with exploits is arduous. So much of our code these days is third-party library code. While our own internal logic can be as secure as Fort Knox, the charting library we used to display the pretty graph may not pass, and we need to be able to swap it quickly with another one.

CHAPTER TWENTY-THREE

Documentation

In this chapter, we'll talk about the process of documenting our project and how to avoid writing endless pages of docs nobody will ever read. I'm breaking this lesson up into two main types of documentation: **coupled**, which is deployed with the code, and **decoupled**, which is deployed separately from the code.

Coupled Documentation

I can recall two very distinct points in my career involving documentation, both when I decided to go overboard with it and then when I decided to abandon it altogether.

Nearly twenty years ago, Yahoo created a front-end JavaScript library called YUI. This was before jQuery gained popularity, and it was fascinating stuff to me at the time. I remember opening the source files and learning and understanding the library's guts mainly because the in-code documentation was so prominent--paragraphs of comments describing every function, variable, and expression.

As a relative novice JS dev at the time, this was fantastic for me. I didn't have to read the ugly code syntax; I could just read the human-friendly words above the code to understand it all. Every block of documentation was wrapped with over 50 asterisks and perfectly indented. I decided to adopt this approach for all my projects regardless of the language. I spent several months strictly adhering to the perfectly OCD commenting process. Every line of code had at least one accompanying comment.

Of course, I eventually realized that I was spending more time commenting than coding and who else but Sean was ever going to be looking at this code in the future?

* * *

Fast forward a few years, I read the book *Clean Code* by Robert C. Martin, which completely changed my outlook on commenting. He proposed that code should describe its intentions and not just execute its logic. Wrapping bits of logic with intuitively-named functions meant I could start composing those functions to tell a story of intent in my code. There's no need for documentation at that point because the application logic tells a human-readable story, and the utility logic is abstracted away inside those functions.

* * *

```
'use strict';

const R      = require('ramda'),
      config = require('config');

const PROTOCOL_SUFFIX = '://',
      PORT_PREFIX   = ':',
      DEFAULT_PROTOCOL = 'https',
      DEFAULT_PORT    = 80;

const authProps     = R.props( [ 'user', 'password' ] ),
      notMissing     = R.complement(R.either(R.isNil, R.isEmpty)),
      allHaveAValue = R.all(notMissing);

const hasAuthProps = R.compose(allHaveAValue, authProps);
const formatAuthString = R.compose(R.concat(R._, B '@'), R.join(':'), authProps);
const maybeUseAuth = R.ifElse(hasAuthProps, formatAuthString, R.always(''));

const hasSchema     = R.compose(notMissing, R.prop( 'schema'));
const formatSchema = R.compose(R.join('/'), R.prepend( '/'), R.prop( 'schema'));
const maybeAppendSchema = R.ifElse(hasSchema, formatSchema, R.always(''));

const throwIfMissingRequiredHost = credentials => {...};

const makeConnectionUrl = credentials => {
  const credentialOr = R.propOr(R._, R._, credentials);

  throwIfMissingRequiredHost(credentials);

  return R.join('', [
    credentialOr(DEFAULT_PROTOCOL, 'protocol'),
    PROTOCOL_SUFFIX,
    maybeUseAuth(credentials),
    credentialOr(undefined, 'host'),
    PORT_PREFIX,
    credentialOr(DEFAULT_PORT, 'port'),
    maybeAppendSchema(credentials)
  ]);
};

module.exports = makeConnectionUrl;
```

The book recommended that we save comments for describing *why* our code is doing something instead of describing *what* it's doing. If we need to explain the why, then the logic is likely unintuitive by design, and we should name our function something self-descriptive and refactor it later.

For example, if we allow users to upload photos to a web app, the result will most likely be an RGB format for the screen. If we need to convert CMYK to RGB manually because our remote image service doesn't do that yet (but ultimately, we would prefer it if they did), we might want to wrap our conversion logic in a function named something like `ensureRgbUntilMediaServiceCanEnsureForUs`. Sure it's a long-winded function name, but there's no question about what it's intended to do or why it's there. No comment is needed. If the media service ever

supports this feature on their end, we can likely delete this function and the guts inside it and call it a day. My favorite long and descriptive function name I saw in a recent project:

```
maybeUpdateMediaAssetConversionJobStatusWithErrorAndRethrow(...)
```

Is there any question about what that function is going to do? It has "maybe" in the name, so we know the update may or may not happen. It even has "rethrow" in the name, so we can infer this function is likely part of an error handler. We can also infer it receives an error as a parameter that will be rethrown when the function is finished trying to update the media asset conversion job status.

I want to zoom in more on the importance of describing **intent**. Imagine we receive a bug ticket that new users aren't being thanked and welcomed when they register. Let's also imagine we work at a place that doesn't have a solid grooming process for user stories. The ticket came from the customer support team relaying some end-user frustration, and we don't have any context whatsoever. Essentially, we reach out to the reporter of the ticket, and they say, "I don't know, the customer just said it would have been nice to thank them for registering and gave us an attitude."

Another day, another dollar. Ok, so we look in the code and see that the final step of the registration flow is calling the `Mail` service to send an email with the template `userRegistrationEmail1`. We look inside that template and see that the contents of it are simply advertising the newsletter.

So what is the actual bug here, and what is the fix? As it stands:

- We don't know if users should receive a newsletter *and* a thank you email and that we're just missing the thank you email `send` call.
- We don't know if the `Mail` service is receiving the wrong template, and the user should be receiving the thank you email *instead* of the newsletter email.

- We don't know if the `userRegistrationEmail1` template is missing verbiage thanking the user for registering.

This process might sound overly analytical and silly if you work in a healthy agile setting with grooming, sprint planning, and regular cross-team communication. The reality, though, is a ton of companies just aren't there; especially older enterprise companies. The registration code may not have been touched for five years, and the product manager and dev who authored it could have left the company. All you really have in front of you is the code, and you need to figure out what's going on and what your options are.

Imagine now the email `send` call was wrapped in a function called `sendNewsletterIntroEmail`, and the template was called `newsletterIntroEmailTemplate`. Now we have context; the code is explicit in its intentions. If we add "thank you" verbiage to this email template, it won't comply with the naming convention anymore, and we will introduce confusion. It makes sense then that we can add a new function called `sendRegistrationThankYouEmail`, ensure a supporting email template, add the logic, and invoke the function right after the newsletter function.

We can go a step further by creating a composite function called `sendNewUserEmails` which invokes the other two functions. That conveys the intention that we want them to be sent at essentially the same time.

Now, when we author our unit tests, our assertions are clear as day. We know precisely if the tests are passing because the functions describe what they expect to happen, and that's the next level of self-documenting code. Our unit tests assert what we *expect* by testing happy paths and what we *expect to avoid* by testing error handling and UX accommodations on sad paths.

Once this syntax is in place, there's no further need for comments. Updates to the code will inherently deliver updates to the documentation as well. Code reviews will be significantly more reliable because devs rarely read surrounding commentary when doing code reviews to see if the comments also need to be updated. Diff tools often

collapse surrounding lines anyhow, so ideally, a diff shouldn't just be displaying a change from A to B; it should also make crystal clear sense *why* the dev is changing from A to B.

You can apply this mindset to other areas of your codebase as well. We talked about wrapping logic with an intuitively named function, but what about code that's not an expression? For error codes and other constants, simply provide a name for the value. Instead of just throwing error 1234 and having a published document that explains all your error codes and meanings behind them, keep it in the code and create a dictionary where `ERROR_CODE_ACCOUNT_EXPIRED` is assigned the value 1234, and then don't ever worry about the value of that error again. The next level up from there might be to create static error types and then throw specific errors instead of intuitively named general errors with a particular code. Depending on your coding language, your options may vary, but the idea is to start making life easier for your future self and the other devs on your team.

```
SNIPPET_ERRORS = {
    system : {
        UNCAUGHT : makeError({name: 'SYSTEM.UNCAUGHT', code: 9999, _statusCode: 501, _message: 'System hiccup! We have been alerted and will be resolving the issue as soon as we can.'}),
        NETWORK : makeError({name: 'SYSTEM.NETWORK', code: 9801, _statusCode: 501, _message: 'System hiccup! We have been alerted and will be resolving the issue as soon as we can.'}),
        CONFIGURATION : makeError({name: 'SYSTEM.CONFIGURATION', code: 9802, _statusCode: 501, _message: 'System hiccup! We have been alerted and will be resolving the issue as soon as we can.'}),
    },
    auth : {
        AUTHORIZED_API_ACCESS : makeError({name: 'AUTHAUTHORIZED_API_ACCESS', code: 8000, _statusCode: 401, _message: 'Unauthorized API access.'}),
        FLAGGED_ITEM : makeError({name: 'AUTH.FLAGGED_ITEM', code: 8001, _statusCode: 200, _message: 'Resource has been flagged for administration.'}),
        TOKEN_EXPIRED : makeError({name: 'AUTH.TOKEN_EXPIRED', code: 8002, _statusCode: 401, _message: 'Permission denied.'}),
        UNAUTHORIZED_API_ACCESS : makeError({name: 'AUTH.UNAUTHORIZED_API_ACCESS', code: 8003, _statusCode: 401, _message: 'Permission denied.'}),
        MISSING_REFRESH_TOKEN_PAYLOAD : makeError({name: 'AUTH.MISSING_REFRESH_TOKEN_PAYLOAD', code: 8004, _statusCode: 401, _message: 'Permission denied.'}),
        MISSING_REFRESH_TOKEN_SECRET : makeError({name: 'AUTH.MISSING_REFRESH_TOKEN_SECRET', code: 8005, _statusCode: 401, _message: 'Permission denied.'}),
        MISSING_REFRESH_TOKEN : makeError({name: 'AUTH.MISSING_REFRESH_TOKEN', code: 8006, _statusCode: 401, _message: 'Permission denied.'}),
        CANNOT_SIGN_TOKEN : makeError({name: 'AUTH.CANNOT_SIGN_TOKEN', code: 8007, _statusCode: 401, _message: 'Permission denied.'}),
        INVALID_CREDENTIALS : makeError({name: 'AUTH.INVALID_CREDENTIALS', code: 8008, _statusCode: 200, _message: 'Invalid credentials.'}),
        USER_ACCOUNT_EXPIRED : makeError({name: 'AUTH.USER_ACCOUNT_EXPIRED', code: 8009, _statusCode: 200, _message: 'Permission denied.'}),
        USER_NEEDS_MFA : makeError({name: 'AUTH.USER_NEEDS_MFA', code: 8010, _statusCode: 200, _message: 'Permission denied.'}),
        USER_MFA_TOKEN_INVALID : makeError({name: 'AUTH.USER_MFA_TOKEN_INVALID', code: 8011, _statusCode: 401, _message: 'Permission denied.'}),
        REFRESH_TOKEN_PAYLOAD : makeError({name: 'AUTH.REFRESH_TOKEN_PAYLOAD', code: 8012, _statusCode: 401, _message: 'Permission denied.'}),
        REFRESH_TOKEN_EXPIRED : makeError({name: 'AUTH.REFRESH_TOKEN_EXPIRED', code: 8013, _statusCode: 401, _message: 'Permission denied.'}),
        TRANSFER_TOKEN_INVALID : makeError({name: 'AUTH.TRANSFER_TOKEN_INVALID', code: 8014, _statusCode: 401, _message: 'Permission denied.'}),
        UNSUPPORTED_STRATEGY : makeError({name: 'AUTH.UNSUPPORTED_STRATEGY', code: 8015, _statusCode: 401, _message: 'Permission denied.'}),
        UNAUTHORIZED_IP_ADDRESS : makeError({name: 'AUTH.UNAUTHORIZED_IP_ADDRESS', code: 8020, _statusCode: 401, _message: 'Permission denied.'}),
        SAML_ERROR : makeError({name: 'AUTH.SAML_ERROR', code: 8021, _statusCode: 401, _message: 'System error.'}),
        LOGIN_ERROR : makeError({name: 'AUTH.LOGIN_ERROR', code: 8022, _statusCode: 401, _message: 'System error.'}),
        UNKNOWN_DIDC_ISSUER : makeError({name: 'AUTH.UNKNOWN_DIDC_ISSUER', code: 8023, _statusCode: 501, _message: 'System error.'}),
    },
    db : {
        NO_QUERY_RESULTS : makeError({name: 'DB.NO_QUERY_RESULTS', code: 6000, _statusCode: 200, _message: 'No db query results.'}),
        NO_DB_CONNECTION : makeError({name: 'DB.NO_DB_CONNECTION', code: 6001, _statusCode: 501, _message: 'System error, please check back later.'}),
        DUPLICATE : makeError({name: 'DB.DUPLICATE', code: 6002, _statusCode: 501, _message: 'System error, please check back later.'}),
        ACCESS_DENIED : makeError({name: 'DB.ACCESS_DENIED', code: 6003, _statusCode: 501, _message: 'Access denied.'}),
        UNKNOWN : makeError({name: 'DB.UNKNOWN', code: 6999, _statusCode: 501, _message: 'System error.'}),
    }
}
```

So given how much pain can come from comments in your code, or even worse, commented-out code that's been committed into the `master` branch, is there a scenario where comments are helpful or necessary? My experience tells me, yes, but there's a caveat.

The comments should ideally only be used to generate public-facing documentation, like API payload schemas. Static analysis should be set

up to compare JavaDoc or JSDoc style param identifiers with the function signatures and fail if they are not aligned. In React, this is similar to defining PropTypes and then relying on Webpack to check the type signature against the component usage. This type of documentation is typically for users who need to integrate your software with their project but may not have access or time to read your software's source code.

```
/**
 * Method sets our slide up for view-ability and triggers the focus() event for our
 * section model.
 * @param slideViewOptions
 * @return MEP.Controller
 */
bootstrapSlide : function(slideViewOptions) {...},

/**
 * Method does some much needed garbage collection for our slideListView. We have to
 * unrender slides out of focus to prevent orphan event listeners from being
 * inadvertently triggered.
 * @param slideListViewOptions
 * @return MEP.Controller
 */
bootstrapSlideList : function(slideListViewOptions) {...},

/**
 * Method keeps to our singleton view container logic.
 * @param slideBulletsViewOptions
 * @return MEP.Controller
 */
bootstrapSlideBullets : function(slideBulletsViewOptions) {...},

/**
 * Method sets our modal up for viewability and triggers the focus() event for our
 * section model.
 * @param branchedContentModalViewOptions
 * @return MEP.Controller
 */
bootstrapBranchedContentModal : function(branchedContentModalViewOptions) {...},

/**
 * Method does some much needed garbage collection for our branchedContentListView. We have to
 * unrender slides out of focus to prevent orphan event listeners from being
 * inadvertently triggered.
 * @param branchedContentListViewOptions
 * @return MEP.Controller
 */
bootstrapBranchedContentList : function(branchedContentListViewOptions) {...},

/**
 * Method does some much needed garbage collection for our branchedContentListView. We have to
 * unrender slides out of focus to prevent orphan event listeners from being
 * inadvertently triggered.
 * @param branchedContentCarouselViewOptions
 */
bootstrapBranchedContentCarousel : function(branchedContentCarouselViewOptions) {...},
```

The takeaway here is any comments in the code should be evaluated by some automated process, so our project isn't relying on human eyeballs. Any published documentation should be generated *from* the code files to ensure a tight coupling and high accuracy and reliability.

Decoupled Documentation

So what if published documentation can't be generated from the code, or what if another team manages it, like a technical writer? At this level, the documentation's purpose and audience are likely different. This person reading published documentation is likely a decision maker who needs to determine whether your library is appropriate for their project. They probably don't need all the low-level details of how each function works once the library is approved.

I like to focus on *process instruction*, not *use instruction*. Detail how to install the library and set up any environment variables or config parameters. Fundamental and concise documentation to get the code running as quickly and painlessly as possible. Once it's running, the developers have a frame of reference that they can use when looking at the low-level generated documentation we mentioned earlier.

The nice thing about README files is that they reside with the code. A pull request can include new instructions when new features are deployed and can be published simultaneously. This type of documentation is much more reliable as it has a much higher engagement than long-lived whitepaper-style documentation in tools like Confluence or SharePoint. I've never met anybody who enjoys reading endless pages of docs and diagrams on those tools. Most engineers in my network would probably agree that there's a high likelihood the docs are out of date since the code is often updated and deployed numerous times without any updates to the Confluence pages.

Again, the solution is typically to generate the documentation from the code. We could hook into the Atlassian API in our CI pipeline for this scenario. It could generate or update those cloud docs as we deploy to an appropriate tier, and then the application becomes self-documenting.

For static documentation in cloud tools, I recommend documenting organizational mappings, like which teams are responsible for which epics and which repositories and pipelines need to adhere to which

roadmaps, and so on. This approach becomes useful when calling out shared static references like error codes or global component libraries several projects can leverage, and so on. We want multiple departments to know about these things, so we allow our static docs to link to them, and then the reader can dig into the README or source code depending on how deeply they want to understand the resources.

Where a README can have three or four different states on three or four different DevOps environments, a Confluence page should really have a static representation that's indifferent to the project's variability in its deployment lifecycle. And that's really why we want to avoid putting version-specifics inside these static published docs.

To recap, non-evaluated documentation typically adds more risk than reward to your project. Try to write self-documenting code that updates as you refactor and aims to integrate as quickly as possible.

CHAPTER TWENTY-FOUR

Numeronyms

If you're new to the term, numeronyms (NOOM-RA-NYM) are abbreviated words that contain a number, which commonly replaces the number of letters that have been omitted. A popular one in the world of containerized applications is Kubernetes, abbreviated as K8s, such that the 8 represents the eight letters u, b, e, r, n, e, t, e.

The idea was initially pretty clever, where a sysadmin in the early 1980s had given an employee named Jan Scherpenhuizen the email account S12n because his last name was too long. These days it's gotten a bit gimmicky and unnecessarily popular. For example, there's a book online titled "Numeronyms: I18n, G11n, L10n, L18n, L12y, A11y, S9y, M12n, C14n, P13n, M17n, E13n, S13n, V12n, V11n, C11y, D11n, N11n, P3n"

As ridiculous as that is, there are three in that title that I want to cover in this chapter, which are, in order of priority, **a11y**, **i18n**, and **p13n**.

Accessibility

A11y stands for accessibility, and it's by far the most important numeronym in this list. I believe people born with disabilities should be given the same opportunities as everybody else. They should be able to experience everything people without disabilities can to the best degree of accommodation possible. Paralyzed individuals shouldn't have to miss a concert simply because they can't walk up the stairs, and blind individuals shouldn't have to miss out on reading books simply because they can't see the words.

Several industries have addressed these hurdles and offer accommodations like wheelchair ramps and braille or audiobooks, and our industry should be no different. When building websites that provide genuine value to the world, we want to extend that value offering to as many people as possible, so we need to take extra steps to ensure disabled people are accommodated.

Take website development, for example. It's generally understood that it's a bad practice to assume the user has the latest browser that supports the latest fancy CSS rules. If we introduce a11y, then we also can't assume the user has a browser with any visual UI whatsoever. Text-only browsers for the visually impaired, like WebbIE for Windows, have been around for several years. Still, the newer versions of Windows and macOS have accessibility features built right in, so that can all be managed at the OS level. Regardless, our websites need to be made in such a way to support these screen readers, legacy and new or have fallbacks with direction if we're technically not able to offer support (like prompting the user to try another accessibility tool that we *can* support).

This means common sense things like not putting text inside images or video without offering `alt` text or video transcripts in the code that screen readers can access. In short, we want to follow the WAI-ARIA specification, which stands for Web Accessibility Initiative--Accessible Rich Internet Applications. Per the Mozilla documentation, There are three main features defined in the spec:

* * *

Roles -- These define what an element is or does. Many of these are so-called landmark roles, which largely duplicate the semantic value of structural elements, such as `role="navigation"` (`<nav>`) or `role="complementary"` (`<aside>`). Some other roles describe different page structures, such as `role="banner"`, `role="search"`, `role="tablist"`, and `role="tabpanel"`, which are commonly found in UIs.

Properties -- These define properties of elements, which can be used to give them extra meaning or semantics. As an example, `aria-required="true"` specifies that a form input needs to be filled in order to be valid, whereas `aria-labelledby="label"` allows you to put an ID on an element, then reference it as being the label for anything else on the page, including multiple elements, which is not possible using `<label for="input">`. As an example, you could use `aria-labelledby` to specify that a key description contained in a `<div>` is the label for multiple table cells, or you could use it as an alternative to image alt text -- specify existing information on the page as an image's alt text, rather than having to repeat it inside the alt attribute.

States -- Special properties that define the current conditions of elements, such as `aria-disabled="true"`, which specifies to a screen reader that a form input is currently disabled. States differ from properties in that properties don't change throughout the lifecycle of an app, whereas states can change, generally programmatically via JavaScript.

There are a ton of features we can leverage in the spec that will make the screen reader experience more accurate and enjoyable for the visually impaired user, so please familiarize yourself with the spec and implement it where it makes sense.

Of course, visually impaired users are only a subset of our user base whom we want to accommodate. Deaf users won't be able to hear words spoken in video unless they can read lips, so at the very least, try not to cover faces when possible or offer subtitles or transcripts.

I recommend simply visiting an online audit tool like Accessible or AccessibilityChecker, which can scan your website and present you with an accessibility score and an itemized list of issues that need attention.

Internationalization

I18n stands for internationalization. While it's not nearly as important as accessibility, it's sadly often a much more popular requirement on large projects. I18n allows us to provide a unique experience based on the user's locale or geographical region. The most common manifestation of this is string translation. German users see German text; United States users see English; Mexican users see Spanish, and so on.

I want to cover i18n not as a principle but as a process. Where accessibility commonly only requires some additional layering of properties or metadata and choosing semantically relevant HTML tags and whatnot, internationalization can demand we change several aspects of how we build our product.

For example, if we want to support multiple languages, then we can't hardcode English copy in our UI files unless we want a separate UI file for each language, respectively. That approach may be ideal if the i18n effort requires more than just translations. Maybe certain products or offers are only available to customers in certain regions, for example. We can add or remove specific UI components or entire pages, so users don't see what doesn't apply to them.

I often recommend storing static strings on the server side using a library or database to organize them and then fetch them through a locale-considerate API. This way, our users aren't forced to download static data they will never see. For example, operating systems are guilty of storing gigs and gigs of language files that we'll never use. They sit in the background for years, taking up disk space, and most of the time, users don't even realize they exist. We can delete these to free up space, and our OS will additionally run better.

For websites, we can serve the default landing page in our default language and then offer locale-specific URI params to serve those same landing pages in the translated form to users in specific regions. **We want to find a good balance between default and preference.** For example, we likely don't want to force every visitor from Germany to

experience a German-translated version of our product--some may be US residents visiting Germany who only speak English. So while our landing page for Germany can be initially delivered in German, there should be some easy way for the user to change their locale or language preference.

This complicates things if we're mixing language translation with regional product offerings. For example, suppose our online catalog includes a product we can ship in the United States that we can't ship to Mexico. Still, we want to offer Spanish-speaking US citizens the ability to shop our US catalog in Spanish. In that case, we need to implement a separation of concerns between our translations and our catalog. Each of those should have its own i18n gateway.

One solution is to detect the user's IP address and apply geoproximity detection to determine the *location* of the user, which can enforce the product catalog restriction. We can use a URI parameter or set a cookie if the user clicks a country flag in the navigation to determine the *language* of the user, which we want to use for the translation.

For dynamic data, like product titles or descriptions, I recommend an approach where the "products" table schema in the database doesn't have a single value for "title" or "description" but instead accepts a JSON object where the keys are the locales, and the values are the translated strings.

<u>id</u>	<u>title</u>	<u>description</u>	<u>upc</u>	<u>price in cents</u>
1	{"en": "Black Muay Thai Boxing Shorts"}	{"en": "Authentic plain black Muay Thai Boxing shorts."}	B11D13V5P247	2499
2	{"en": "Black Muay Thai Boxing Short w/ Lettering"}	{"en": "Authentic black Muay Thai Boxing shorts with \"muai thai\"; embroidered on the front (in Thai letters)."}	B11D13V5P248	2999
3	{"en": "Black/Red Muay Thai Boxing Shorts w/ White Stars"}	{"en": "Authentic black and red Muay Thai Boxing shorts with white stars on the sides."}	B11D13V5P249	2899
4	{"en": "Black/Red Muay Thai Boxing Shorts w/ White Stars & Lettering"}	{"en": "Authentic black and red Muay Thai Boxing shorts with white stars on the legs, and \"muai thai\"; embroidered on the front (in Thai letters)."}	B11D13V5P250	3999
5	{"en": "Blue/White Muay Thai Boxing Shorts w/ Red Stars"}	{"en": "Authentic blue and white Muay Thai Boxing shorts with red stars on the sides."}	B11D13V5P251	2899
6	{"en": "Blue/White Muay Thai Boxing Shorts w/ Red Stars & Lettering"}	{"en": "Authentic blue and white Muay Thai Boxing shorts with red stars on the legs, and \"muai thai\"; embroidered on the front (in Thai letters)."}	B11D13V5P252	3999
7	{"en": "Red/Black Muay Thai Boxing Shorts w/ White Stars"}	{"en": "Authentic red and black Muay Thai Boxing shorts with white stars on the sides."}	B11D13V5P253	2899
8	{"en": "Red/Black Muay Thai Boxing Shorts w/ White Stars & Lettering"}	{"en": "Authentic red and black Muay Thai Boxing shorts with white stars on the legs, and \"muai thai\"; embroidered on the front (in Thai letters)."}	B11D13V5P254	3999
9	{"en": "Black/Green Muay Thai Boxing Shorts"}	{"en": "Authentic black and green Muay Thai Boxing shorts."}	B11D13V5P255	2899
10	{"en": "Black/Green Muay Thai Boxing Shorts w/ Lettering"}	{"en": "Authentic black and green Muay Thai Boxing shorts with \"muai thai\"; embroidered on the front (in Thai letters)."}	B11D13V5P256	3999

Then when we fetch our products from the database, our API route will

detect the language from the HTTP request. The controller can iterate through the response payload and replace the title and description values with the literal strings respective to the locale before sending it down to the requestor.

This is one way to solve the challenge. Of course, there are plenty of alternatives, but you can see how we could quickly scale this to accommodate new languages without having to change any code. Alternatively, we could remove those JSON columns altogether and have locale-specific mapping tables with foreign keys that point back to our product ID. Or, we could have multiple locale-specific title columns like "title_en", "title_es", "title_de", and so on. I don't recommend this approach because we don't want to do database migrations every time we want to support a new locale on a table. It's just an option, and it's good to be aware of options even if they're not ideal so if we happen to run across them while working on a client system we won't be caught off guard.

A NoSQL database like MongoDB would eliminate the RDBMS mapping headache and remove the need to use a hybrid approach like my example. Still, we would lose the other benefits from foreign key constraints and whatnot. Pulling up a customer order and including a subset of product details would be an entirely different headache. I probably sound like a broken record at this point, but I can't emphasize enough: **however you use your tools, think ahead and leave yourself an exit should you change your mind or find a better approach and want to refactor.**

One thing to consider is that most devs can't speak multiple languages. Offshore devs can often speak their native language and English, but we need to support more than one or two languages, regardless. There will likely be a process involving an internal translation team or a third-party translation service. They will need a way to retrieve all the English strings and provide the translated strings back to us for each locale, respectively. How we send those strings back and forth and get them into our system is subject to some compromise, depending on who we're working with.

* * *

For my e-commerce example with translated product titles and descriptions, it makes sense to build a CMS to **present back-end complexity as front-end simplicity**. On the UI, we could create a flow where different people could add multiple titles and descriptions in clearly labeled languages for each product, adding support for more languages at any point down the road. The app would consolidate them into JSON objects for the API to create or update product details.

Remember, just because we want to organize data a certain way in one tool doesn't mean we need to simulate that experience in another tool. Each tool shines in its own way, so really, our focus as engineers should be either building the tools or building the handshakes and recognizing the importance of differentiating the two so when we change tools, our effort to update the handshake is minimal. If we're leveraging the strategy design pattern, we simply code a new handshake for the new tool, and then we can support both.

As a general principle, I recommend that when we're building applications, we ask ourselves or the client, "Will this product *ever* need to support international users?" If yes, even years down the road, we should leverage abstraction and patterns to **support the capability of i18n**. However, the initial population of strings can still be the application's default language.

Launch the product, make money from the initial target market, add new languages at a comfortable pace in the background, and slowly release the product to new geographical regions where it makes sense for the business. The more we can channel corporate pivots and **scale through data augmentation instead of code refactoring**, the happier everybody will be on the project.

Personalization

P13n stands for personalization, and it's the least important on my list because it's really more of a nice-to-have for most products. Users typically don't miss it if they never had it, but once they experience it, they never want to go back. It can always be added later and rolled out in layers at a comfortable pace based on customer feedback.

Personalization allows users to customize the product experience for their personal needs. For example, if our product has a dashboard with some widgets, we may find that only six or eight widgets fit on the screen, but as we scale our product, we end up building a total of twenty widgets. Who decides which six of those twenty should reside on the dashboard and which get stuck in a menu or through another UI portal?

Initially, the product team will likely choose the six most popular widgets to appear on the dashboard. Still, eventually, customers will start to ask for customizations. They may want widgets 12 or 18 to be on the dashboard instead of widget 3. Users will want different widgets on their dashboards based on their use case. In desktop software, this is very common. In applications like Unity or Adobe Photoshop, we can rearrange the UI completely, pin menus in different places, and even populate some menus with icons we use more than others to save time.

Another example of personalization in these products is supporting custom hotkeys. Most desktop applications have some pre-defined hotkeys, but if we're very comfortable working in one application and tasked to work in a competitor's application, we don't want our workflow to be affected too much. Mapping out our own personal set of hotkeys that do the same thing in different applications can be a very powerful accommodation.

Yet another example of personalization these days is dark mode. Some people enjoy dark text on a light background, and some want light text on a dark background. Some people enjoy both of them, but at different times of the day to allow consideration for eye strain. If our application

offers a toggle where the user can go back and forth with a single button click, that's a compelling and considerate feature.

So, what does this mean for us as software professionals? We need to consider componentizing the UI and using modular code very early on. Instead of coding big bloated files that do multiple things, we need to code multiple lean files that each do a specific thing. In object-oriented programming, it's as simple as disciplined class inheritance and hierarchy through interface design.

A button is a button, and it shouldn't care if it's in menu A or menu B. Equally, a menu is a menu and shouldn't care if it's in dock A or dock B. If the child doesn't care about the parent and only cares about itself and how it can talk to the system, then our options for UI flexibility really open up. We simply need to store a config somewhere bound to the user and write some code to organize the UI based on that config instead of some predetermined layout mapping.

Personalization can also live behind the curtain. For example, if we're creating a continuous integration pipeline, our needs will likely differ from the needs of another application. Earlier I mentioned adding branch rules on GitHub, which can execute "actions" that serve our business needs. Where *we* may want to run a linter and a static analysis tool, another user may wish to run a virus scan. GitHub as a product understands this, and so instead of offering a canned CI pipeline for pull requests, it provides personalization so users can configure their own. This is exponentially more valuable as a feature overall. This, of course, is nothing new. Tools like Jenkins and Hudson have been offering this style of personalization for years.

Again, as software professionals, this type of personalization can manifest as building infrastructure that supports an asynchronous data flow that leverages hooks and events through a message bus. The building blocks lean much more toward utility and much less toward application. Imagine a toy car vs. a toy airplane. Now imagine a lego kit that could build either. This type of mindset when building software systems typically shines internally before it shines externally--meaning the engineering team will see velocity gains by being able to reuse

components and modules in various parts of the application behind the scenes. When we extend that mindset slightly, we can allow for that same level of "plug-and-play" at the UX level. It's all just a matter of thinking ahead.

CHAPTER TWENTY-FIVE

Clean Code

Clean code is so important. In this chapter, we'll discuss ways to ensure the codebase remains clean and consistent and why we want to hold ourselves accountable to these rules.

Style Guides

The first and most common concept most devs are familiar with is the idea of a style guide or coding standards for a project. Before we get too deep on this topic, please understand that I have no intention of listing any of my coding standards or preferred style guides used on my projects for various programming languages. The fact is, they don't matter.

I've mentioned in earlier chapters that these preferences in syntax are merely *my* preferences, and they typically don't change the output of the code or the deliverable. And while I use them extensively on my own projects, I've observed that dev teams tend to produce at a higher quality and velocity when the morale is high, regardless of the coding standards. So while I love my standards, I'll never sacrifice team morale to get my way.

What does matter is consistency. The ultimate goal should be that if we open up any file in our project, we can't tell which dev authored it. Suppose we open a couple of files, and it's immediately apparent that Bob coded this one and Sarah coded that one because Bob likes camelCase and Sarah likes underscores. In that case, we're creating an atmosphere of confusion and just asking for a slew of problems. If a bug is found in Bob's file and Sarah is assigned the bug ticket, does she need to adhere to Bob's style in that file, or does she ignore his style and use hers (leaving a file with two styles which is essentially the absence of a standard)? Or do we assign the bug ticket to Bob because it's Bob's file?

At the end of the day, if we have a team of devs on a project, we should have a coding standard for the project by which every team member abides. End of discussion. We can have our preferences for personal projects, but on a shared project, the first step in collaboration is alignment on tools and syntax. Getting alignment here means we set aside our preferences, voice our opinion if it's asked of us, be happy for the ease of work when we get your way, and be comfortable expanding our skill set and comfort zone when we don't.

Some languages like Python, Haskell, and Bash don't offer a whole lot of flexibility with the syntax, so we don't have much of a debate at all here. Languages like PHP or JavaScript provide an extreme amount of freedom of expression. We can get our project off to a bad start if we don't set expectations and guidelines immediately. Ideally, don't invent anything here. Adopt an existing style guide, and either decide and mandate from a technical leadership position or offer a democratic vote to the devs where the majority can choose.

I mentioned in an earlier chapter that I used to believe that my ways of coding were the best. Of course, over time, I learned that my value doesn't come from whether or not my expression has a semicolon or whether I put curly braces around conditionals. I have my beliefs on what standards help *me* code better. Still, once I understood that these standards are subjective and that *other* devs code better with whichever standards they're used to, I took myself out of the debate entirely. I forced myself to believe that I could be the best dev on the team, no matter what language or syntax rules were thrown at me. **That's the senior mindset.**

Linter

So once the team adopts a coding standard, how do we enforce it? We certainly don't want to spend our code reviews suggesting syntax style changes or pointing out divergences from the coding standard. Never in our code reviews should we find ourselves saying, "This expression should have curly braces" or "This variable should be camelCase." We shouldn't have to look at a new function and say, "I see you defined this variable, but I don't see you using or returning it. Do we need it?"

Enter the linter. Linting is seriously one of the best process additions since unit tests. Tools like ESLint, Checkstyle, or Flake8 can act like a pair programming dev looking over your shoulder, pointing out typos and code smells as you write your logic. Additionally, many of these tools have an "auto-fix" option which manually applies semantically safe refactors to adhere to the coding standard.

We can hook these into our development pipeline at multiple steps to enforce some redundancy in our quality control:

- **Locally** in real time as we type or save our files.
- **Locally**, when we build binaries or packages or as a pre-commit hook to our VCS.
- **Remotely** inside VCS as a pre-review audit. For example, in GitHub, we can create an action that runs the linter; if it fails, it won't allow the branch to be merged.

Static Analysis

While some linters extend their capability and analysis outside of stylistic auditing, I prefer to configure my linters to do just that and reserve logical analysis for tools that specialize in this realm, like SonarQube. That's just me trying to avoid giving any one tool or service too much authority. I like to diversify my capability and assurance layers so we can swap out chunks of our projects without such a drastic ripple effect.

So, where a linter may yell at us for getting lazy on formatting or stylistic decisions, a static analysis tool like SonarQube or Embold.io can analyze the performance and optimization of our codebase. Is it efficient or not? Is it redundant or DRY? Are there opportunities for consolidation or abstraction? We can set rules to prevent nested conditionals or to detect potential infinite loops and help us not only author cleaner code that's easier to reason about but also help ensure we're getting sufficient unit test coverage for edge cases we may not have thought of.

So, given all these rules and guidelines for clean code allow for a codebase that's easy to refactor, audit, and maintain, what's the *real* win here for us as senior devs? Certainly, codebases like this are more enjoyable, but is there something here that can help our career? Absolutely. When we can offload all the trivial quality assurance to automated tools, we can focus our mentoring energy on other things like collaboration, helping junior devs recognize the importance of specific patterns, and helping the product team and project managers plan out the roadmap with architectural and infrastructure considerations. We can manage our time better and make ourselves much more available as a leader on the project.

Instead of us doing code reviews on potential garbage code and kicking back pull requests with syntax changes all day, we can reserve our eyes for working code that adheres to our guidelines. Our code review eyes can gloss over syntax and look for logical issues: sync vs. async execution, naming conventions to help convey intent, ensuring code is

abstracted to scalable, and so on. Questions like "Do we need a semicolon here?" become "Does this AWS SDK call here need to be abstracted into a cloud-agnostic service in case we want to deploy to another vendor in the future?"

As senior engineers and architects, our value *must* extend far beyond an automated tool's capabilities. **If all we're doing all day is work that can be automated, we become replaceable, and it's only a matter of time before we're phased out.** We need to stay ahead of this curve, align ourselves with our client or employer's definition of value, and use all the time we get back from automation to go above and beyond. We must exceed expectations and prove our contributions to be invaluable.

CHAPTER TWENTY-SIX

Pure Programming

Pure programming is a way to create functions whose output is defined by their input and have no side effects. Pure functions are only one component in a collection of functional programming methodologies. Still, its value is unparalleled as it can also be used in object-oriented and procedural programming languages. Let's digest what a pure function is, starting with "a function whose output is defined by its input." What does that mean?

Let's take an elementary function as an example. We'll make a function called `addTwo`, which will accept two numbers and return the sum.

```
const addTwo = (a, b) => a + b;
```

If we give this function 5 and 3, it returns 8 **every single time**. Very simple. The **output** "8" is defined by the **input** "5" and "3". The term for this part of the pure function is a "deterministic function."

```
addTwo(5, 3); // 8
```

It's very straightforward and obvious, but how could we change this function so that its output is *not* defined by its input, so you can understand *why* this is so important?

What if we add a condition here:

```
const addTwo = (a, b) => {
  let sum = a + b;

  if (user.isAwesome) {
    sum += 5;
  }

  return sum;
};
```

Now, for users who are not awesome, the function returns 8, but for awesome users, the function returns 13. The function is no longer intuitive, especially because it is named `addTwo`, not `addTwoAndMaybeAnAwesomeUserBonus`.

This example may seem silly, but where would this scenario pop up in the real world? I see it all the time in React components. A function is supposed to do something, but if some value in the state meets some condition, the function returns something else instead, like `null`.

This adds a lot of complexity when we introduce testing because now our tests need to consider external context factors like state or like `user` in our first example. If we want our code to be portable and reusable, like I mentioned when talking about the strategy pattern, it needs to be agnostic and unaware of its surroundings. The *output* needs to be defined by the *input*, **not** by the input and some other stateful criteria. Any data the function needs to deal with inside its context needs to be given to it as parameters.

The second part of the pure function definition is that it has no side effects. If something else in the app changes when we run our function, it adds a lot of confusion and complexity, severely slowing down troubleshooting and debugging efforts. One primary cause of this is mutation, which is when we change a referenced piece of data in memory so the next time it's accessed, it's different than before.

Imagine our function accepts two objects instead of two numbers and imagine its purpose is to merge them together:

```
const mergeTwo = (a, b) => ({ ...a, ...b });
```

The function should ultimately return a brand new object that is the direct result of merging the two given to it. In other words, imagine `objectA` lives in memory in bank 1, and `objectB` lives in memory in bank 2:

```
const objectA = {
  id : 123,
  name : "Sean Cannon",
  age : 42
};

const objectB = {
  id : 456,
  userId : 123,
  role : "Instructor"
};
```

Our pure function will only **read** from banks 1 and 2 and will return a brand new merged object that will be written to memory in bank 3 if it happens to be saved to a variable:

```
mergeTwo(objectA, objectB);

// Result:
// {
//   age : 42,
//   id : 456,
//   name : "Sean Cannon",
//   role : "Instructor",
//   userId : 123
// }
// objectA:
// {
//   id : 123,
//   name : "Sean Cannon",
//   age : 42
// }
// objectB:
// {
//   id : 456,
//   userId : 123,
//   role : "Instructor"
// };
```

If our function *writes* to banks 1 or 2, then we're creating a **side effect**, and our function is *not* pure:

```
const mergeTwo = Object.assign(a, b);
```

Other parts of the app that expect objects A and B to contain certain data will unexpectedly receive objects that contain *different* data thanks to our function, or rather "no thanks" to our function. This is bad news:

```
mergeTwo(objectA, objectB);

// Result:
// {
//   age      : 42,
//   id       : 456,
//   name     : "Sean Cannon",
//   role     : "Instructor",
//   userId   : 123
// }
// objectA:
// {
//   id      : 456,
//   name   : "Sean Cannon",
//   age    : 42,
//   userId : 123,
//   role   : "Instructor"
// };
// objectB:
// {
//   id      : 456,
//   userId : 123,
//   role   : "Instructor"
// };
```

Rule of thumb, if your function receives an object or array, it should clone them. If your goal is state management, you need a state machine that's very tightly coupled to the state itself, so it's simple to manage. Some languages like PHP will take care of this for you and default to passing by value instead of passing by reference, and you have to explicitly note that you're mutating your function's input if you choose to do so.

In languages that pass by reference, like JavaScript, you *need* to be careful and manually copy your data. Please note that this will slow down your app slightly and increase memory and processing requirements. Still, the cost savings in reduced troubleshooting can typically pay for the hardware so many times over that it's not even worth debating.

CHAPTER TWENTY-SEVEN

Functional Programming

Before we dive in, please understand this chapter is in no way a complete, in-depth guide to functional programming. It's far from it; this chapter barely scratches the surface of FP, but I cover two principles that drastically improved my projects and helped me grow my career: **currying** and **composition**. I highly recommend you read Professor Frisby's *Mostly Adequate Guide To Functional Programming*^{*} for an in-depth guide. It's free and excellent, and I refer back to it regularly to check myself and ensure my understanding of something is still correct. I also recommend a repo by the user Hemanth called *Functional Programming Jargon*[†], which measurably helped me initially when the FP buzzwords like Monad, Monoid, and Functor made no sense to me.

Once you start getting excited with FP, play with some Clojure or Haskell stuff in your spare time. Remember, these are just tools, so while you may want them on your toolbelt, they're not as widely adopted in the grand scheme of things, and the projects that pay top dollar for our services aren't typically Haskell projects.

So, similarly to how Object-oriented programming attempted to help organize the world of procedural programming by taking advantage of

^{*} <https://mostly-adequate.gitbook.io/mostly-adequate-guide/>

[†] <https://github.com/hemanth/functional-programming-jargon>

reuse and inheritance, functional programming aims to simplify logical expressions by taking state out of the equation. It simplifies our project functions by limiting what they accept and what they do internally. Instead of coding a method that does five things and knows about the state of `this` or `self`, we code five functions, each with its own tests, each accepting a single param and each doing one thing. It sounds like more work, but it's really just re-organized work, and the payoff is immense.

The simpler the function, the more portable and reusable it is. The function no longer needs to know about its context. In OOP, we might have a method like `User.validateEmail`, which accepts zero params and validates the string assigned to `this.email`. In FP, we can make a curried function named `validateString` which will ultimately accept a regular expression pattern and a string to test. We can leverage currying to create pre-loaded context-considerate functions to convey intent better.

We'll unpack that in a second, but for now, understand that the goal is simply to create functions we can partially invoke and then compose, so the result of function **1** is passed as the input param to function **2** and so on. Our code begins to tell a story--a chained sequence of human-readable instructions.

I wanted to include this chapter for two reasons. The first is that I believe functional programming is the most under-appreciated type of programming, even though it's been around forever. Lisp came out in 1960, and if you've ever piped commands in Unix, you're already on your way to grasping the value of FP. The second reason is that I think the biggest problem with enterprise digital transformation projects is logistics. The projects have too many people with too many tasks and not enough compartmentalization in the infrastructure and codebase. It becomes increasingly difficult to assign and delegate those tasks without blocking, clobbering, or building duplicate and conflicting features.

How can we simplify those huge projects? One solution I often recommend is to start leveraging more and more FP because it plays nicely with microservice architectures. Chunks of logic can be carved off

and built asynchronously and delivered in parallel with other logic because we're not worried about breaking monolith systems or bloating out huge classes that depend on state. Those systems are a nightmare. Every deployment introduces regressions. Features are put in the backlog because bugs are so prominent and fixes are so urgent.

Let's get into the weeds to better understand what I mean.

Currying

I briefly mentioned currying so let's start there. Currying is the process of converting a function that accepts multiple parameters into a function that takes them one at a time. Curried functions create a closure, so the inner functions have access to the outer functions' scope.

Imagine we have a function that accepts three numbers, adds them, and returns the sum:

```
const addThreeNumbers = (a, b, c) => a + b + c;
```

We can curry this function easily in ES6 by chaining fat arrows like so:

```
const addThreeNumbers = a => b => c => a + b + c;
```

So instead of invoking the function like this:

```
addThreeNumbers(3, 4, 5);
```

We invoke it like this:

```
addThreeNumbers(3)(4)(5);
```

That's cool, but why would we want to do this? So far, this is a parallel change and doesn't improve the project.

Well, the fewer parameters our functions accept, the simpler and more reusable they can be. And, because our curried function returns subsequent functions that each accept subsequent params, we can preload them with context or dependencies.

We can also create partial assertion functions called predicates. If you remember from English class in grade school, sentences have a subject and a predicate, and conveniently for us, so do our conditional expressions.

Imagine the sentence, "The television weighs 80 pounds." The subject here is "the television," and the predicate is "weighs 80 pounds". In

functional programming, we can create predicates that accept a subject when we want to make validations or type assertions. Predicates always return a boolean. The television is either 80 pounds, or it isn't. It's an objective assertion that's either true or false.

If we refer back to our user email validation, our curried `validateString` function will look like this:

```
const validateString = pattern => string => pattern.test(string);
```

Notice we put the subject *last* in the chain of params. This is so we can pre-load context and dependencies into our curried functions. I mention this because this is typically reversed from standard functions and methods where mandatory params are first, and optional params are last; this is a common hurdle for many devs learning FP.

Back to our example; from here, we could reuse this function to create predicates for different strings that have different rules:

```
const isValidEmail = validateString(/\S+@\S+/);
const isValidUsername = validateString(/^(a-z0-9_-){3,15}\$/i);
// and so on
```

Because we aren't accepting all the parameters at once, we can invoke the function with the first param, which returns a function that accepts the *next* param. This is a game changer in the effort to simplify complicated systems. `isValidEmail` is now a function that knows about its pattern and accepts a string.

Notice the naming convention; `isValidEmail`, and `isValidUsername`. We name our predicates in a way that implies what they are asserting. Some additional examples here that we might see in our application could be `isExistingCustomer`, `hasReceivedWelcomeEmail`, or `canAccessPremiumContent`. They can get as low-level as type assertions like `isString`, `isNumber`, or even `isNumericString`.

The guts of these functions would be up to you, but we want each only to accept one parameter, the subject (which I'll discuss further in a bit); the simpler, the better. I published an open-source predicates library on

npm called `Prettycats`^{*} that includes a bunch of these you can use for free.

So if we take one of these predicates as an example, we might imagine the first pass of `canAccessPremiumContent` looking something like this to get the ball rolling :

```
const canAccessPremiumContent = user => user.roles.includes('premium');
```

This is a stylistic decision here and not the safest because we trust `user` and `user.roles`, but for this example, you get the idea. As a next step, do we want to simplify the execution process or the internals? Or, in other words, should conditional logic live inside the function or in the context where the function is being called?

If we have a `user` object handy, calling `canAccessPremiumContent(user)`; is quite simple, so maybe that's what we want. This is a slippery slope, though, because if the `user` object isn't strongly typed, then the volatility of that object may require some intelligent logic inside our function to determine the result and contain error handling for unforeseen changes to the payload, but that can introduce sneaky bugs later on if we aren't careful.

The next iteration of this function might punt the burden of extracting the user roles to the caller and might look like this:

```
const canAccessPremiumContent = roles => roles.includes('premium');
```

Ok, this is getting simpler, but it still has room for improvement. Now our function accepts an assumed array of strings, so we'd have to invoke it like `canAccessPremiumContent(user.roles)`; but it's not very well protected still since the function will throw an error if we give it a value that doesn't have a native `includes` method on its prototype.

We can protect it a bit by invoking like `canAccessPremiumContent(user.roles || [])`; but that only protects against a falsy param like `undefined` or `null`, so truthy params of an

* <https://www.npmjs.com/package/prettycats>

unexpected type like `Number` will still throw an error.

If we leverage currying, we can privatize and control this risk a bit more:

```
const safeArrayIncludes = value => (arr = []) => {
  try {
    return [...arr].includes(value);
  } catch (e) {
    return false;
  }
};

const canAccessPremiumContent = safeArrayIncludes('premium');
const canAccessAdminContent = safeArrayIncludes('admin');

canAccessPremiumContent(user.roles); // true|false
```

By removing the context from the function, you can see that we get a generic utility function that knows nothing about roles or users. It can be abstracted into a helper library and used across files and projects. This type of utility logic is typically exempt from intellectual property claims. It's just a pattern, and nothing about it correlates to a client, project, or application. The IP comes with the predicates that use it; `canAccessPremiumContent` starts to get into the realm of intellectual property. Not all applications have premium content, and depending on the value given to this function, we may be using proprietary data that can't be leveraged in another project.

Simply adding a single layer of currying has allowed us to separate application logic from utility logic. It's given context to the intention of the function. `canAccessPremiumContent` is very descriptive and, more importantly, indifferent to the actual role name here.

Imagine the roles are just numbers or gibberish. If we had a function `hasRole('asdf1234')`, it's not conveying *why* we care if the user has that role. What's special about that role? We don't ever want someone to look at our code and say, "I see it's doing X but is it *supposed* to be doing X?" Or should it be doing Y? If that role is provided from a variable or inferred through some other function response, we may never see the word "premium" in the data, so **it's up to us to name our functions intuitively.**

Let's put predicates aside for now. When else would we want to

leverage currying? My number one personal gain from currying has definitely been easy dependency injection. Dependency injection makes our functions more portable and safe because state and context are *provided to* the function instead of *inferred by* the function. It also makes our functions more testable because we can inject mocks and shims; our functions won't know or care if they're running in our application or our tests.

Imagine we have a function called `sendOrderConfirmationEmail`, and it looks like this:

```
// FILE ONE
const MailSvc = require('/path/to/services/mail/Mail')
(config.email.strategy);

const sendOrderConfirmationEmail = orderDetails => {
  return MailSvc.send(orderDetails, 'orderConfirmationEmailTemplate');
};

module.exports = {
  sendOrderConfirmationEmail
};

// FILE TWO - USAGE EXAMPLE
sendOrderConfirmationEmail(orderDetails)
  .then(() => 'Email sent')
  .catch(err => `Email could not be sent : ${err.message}`);
```

This function is pretty simple. It accepts some order details, which would be parsed and interpolated into the email template, and it sends the email to whatever email address is attached to the order. But what about `MailSvc`? Our exported function references `MailSvc` in the scope of its declaration. It's essentially memoized and uses whatever strategy is in the config dictionary in that file.

If we want to test this file, we need to somehow hijack the `require` process, so it pulls in our stub or accepts a config override meant for the unit tests. Anytime we use tools that hijack the runtime and do magical things behind the curtain to get our tests running, we lose credibility and assurance that our test is actually proving anything that we need it to. It becomes so esoteric that we risk testing our tests and offering no value to the production deployments.

A simple enhancement to this function would be to move that `MailSvc` `require` statement to the file *using* the `sendOrderConfirmationEmail`

function and add it as a curried parameter.

For example, I like to use this pattern where I pass in a dependencies object that can grow or shrink as the project evolves, and JS is forgiving enough that most of the time, there are no regressions to address:

```
// FILE ONE
const sendOrderConfirmationEmail = ({ MailSvc }) => orderDetails => {
  return MailSvc.send(orderDetails, 'orderConfirmationEmailTemplate');
};

module.exports = {
  sendOrderConfirmationEmail
};

// FILE TWO - USAGE EXAMPLE
const MailSvc = require('/path/to/services/mail/Mail')
(config.email.strategy);

sendOrderConfirmationEmail({ MailSvc })(orderDetails)
  .then(() => 'Email sent')
  .catch(err => `Email could not be sent : ${err.message}`);
```

When we want to test this, our assertions can pass in a `FakeMailSvc`, to which we can attach a spy[†] and assert that the `FakeMailSvc.send` was called with the appropriate payload. For this example, I would likely create two initial fake services. One happy service and one sad service:

```
// UNIT TEST FILE
const { sendOrderConfirmationEmail } = require('/path/to/
emailFunctions');

const fakeOrderDetails = {
  email : 'eric@test.com',
  orderNumber : 'erictest1234'
};

const FakeMailSvc = {
  send : () => Promise.resolve('ok')
};

const FakeMailSvcFail = {
  send : () => Promise.reject({ message : 'failed' })
};

describe('sendOrderConfirmationEmail', () => {
  beforeAll(() => {
    spyOn(FakeMailSvc, 'send').and.callThrough();
    spyOn(FakeMailSvcFail, 'send').and.callThrough();
  });

  it('sends an email', done => {
    const MailSvc = FakeMailSvc;
```

[†] Example:

`spyOn(someObj, 'func').withArgs(1, 2, 3).and.returnValue(42);`
`someObj.func(1, 2, 3); // returns 42`

```

    sendOrderConfirmationEmail({ MailSvc })(fakeOrderDetails)
      .then(res => {
        expect(FakeMailSvc.send).toHaveBeenCalledWith(fakeOrderDetails);
        expect(res).toBe('ok');
        done();
      })
      .catch(done.fail);
  });

  it('fails gracefully', done => {
    const MailSvc = FakeMailSvcFail;
    sendOrderConfirmationEmail({ MailSvc })(fakeOrderDetails)
      .then(done.fail)
      .catch(err => {
        expect(FakeMailSvcFail.send).toHaveBeenCalledWith(fakeOrderDetails);
        expect(err.message).toBe('failed');
        done();
      });
  });
});

```

Here you can see we created a `FakeMailSvc` object and a `FakeMailSvcFail` object. These are mocks to trick our function into thinking it received code for a live cloud mail service.

Before any tests run, we attach a spy, a feature in Jasmine that essentially creates a binding in memory so we can check to see if our object methods were invoked.

Now each of our assertions can test that the function returns what it's supposed to and also that it can handle the service failure as well. We can take the surprise out of the failures and account for them instead of reacting to them when the customers or users experience them.

You can see here how this simple dependency injection capability can completely liberate our testing efforts. We can now test a range of service mishaps and edge cases with variations of our mocks and truly get 100% test coverage in our project.

This type of curried dependency injection also aligns with the strategy pattern, which you might have noticed in the line when we were requiring our `MailSvc`. Not only can our function accept a range of strategies to send mail on various cloud services like MailChimp or Sendgrid, but we could add a strategy to send mail on a local SMTP server or use a strategy that hooks into some monitoring tools that might not be prevalent or available on lower deployment tiers. We can

then write tests for each strategy and ensure our code behaves as expected without discovering unfortunate surprises in production.

Composition

So now that we understand the value of curried functions, the next step in evolving our code is to start composing them. In Unix, we just add a pipe. In JavaScript, the most common example is probably promise chains, which are essentially asynchronous functional composition. So if you think of `compose` as a simpler, synchronous promise chain, you've already grasped this.

I've mentioned in previous chapters my love for the JavaScript library Ramda*. It's a collection of functions that all play nicely together and polyfill functional programming pattern support in JavaScript. We can use it here and there or pick a couple of functions we like and use it a la carte, or we can completely take over our JS and fill our files with thousands of `R.something` calls. It's really up to you.

```
const decorateColumn = R.compose(
  d => moment.utc(d).valueOf(),
  R.last,
  R.match(/\b((([0-2])|([07][1-9]))|([0-9]{01})|([12]\d{3})|([07][1-9]))|([20]\d{2}))\b/g
);

const applyColumnDecoration = columns => R.compose(
  R.fromPairs,
  R.map((v) => [decorateColumn((columns.find(R.propEq('key', v[0])) || {})['display_name'] || v[0]), v[1]]),
  R.toPairs,
  R.map(
    R.when(
      R.isNil,
      R.always((a) => a)
    )
  )
);
```

* <https://ramdajs.com/>

```

const parseTree = columns => R.compose(
  RA.renameKeys({ columns : 'values' }),
  R.over(
    R.lensProp('children'),
    R.map(
      R.over(
        R.lensProp('children'),
        R.omit([columns])
      )
    )
  ),
  R.over(
    R.lensProp('children'),
    R.map(
      R.over(
        R.lensPath(['children', 'data']),
        R.when(
          R.is(Object),
          R.map(
            R.when(
              R.compose(
                R.is(Object),
                R.prop('values'),
                R.defaultTo({})
              ),
              R.over(
                R.lensProp('values'),
                R.compose(
                  R.omit([columns]),
                  applyColumnDecoration(columns)
                )
              )
            )
          )
        )
      )
    )
  ),
  R.over(
    R.lensProp('columns'),
    applyColumnDecoration(columns)
  )
);

```

Some of the functions are more valuable than others. In ES6, I can chain fat arrows to curry my functions. In ES5, we would have had to return nested functions using the `function` keyword with curly braces, which would get messy quickly--passing a single function that accepts multiple params to `R.curry` just auto-curries it for me.

One Ramda function I use every day, however, is `R.compose`, which is essentially this:

```
const compose = (f, g) => (...args) => f(g(...args));
```

Consider this example:

```

const getLinkedAccountIds = R.compose(
  R.pluck('id'),
  R.reject(R.isNil),
  R.flatten,
  R.defaultTo([]),
  R.pluck('linkedAccounts'),
  R.propOr([], 'accountAttributes')
);
***
```

Compose is based on algebra, so we go from right to left, or in this example, from bottom to top. This takes a bit of getting used to if you aren't math savvy, but it's the purest way to do it. Ramda offers an alternative compose function called `R.pipe`, which flips the direction, allowing you to order your functions left to right or top to bottom.

If you've never composed functions before, this might look like magic. Our `getLinkedAccountIds` function doesn't appear to be accepting anything or doing anything, but it's actually doing a lot.

`R.compose` accepts a series of functions as params, but it's curried, so once you provide all your functions, you invoke it once more with the payload that will be given to the first function in the sequence.

Imagine the following object :

```
const account = {
  id : 123,
  accountAttributes : [
    {
      name : 'foo',
      linkedAccounts : [{ id : 234 },
        {
          id : 345
        }]
    },
    {
      name : 'bar',
      linkedAccounts : [{ id : 321 }
        ]
    },
    {
      name : 'buz'
    },
    {
      name : 'baz',
      linkedAccounts : [{ id : 456
        }]
    }
  ]
};
```

If we invoke our composed function with this account object, this is our result:

```
getLinkedAccontIds(account); // [234, 345, 321, 456]
```

As much as I'd love to go line by line and explain what this code is

doing, the point of this example isn't to teach you Ramda but rather to show you how powerful functional composition is. I can't emphasize enough that tools should not define your value. They come and go, and while sometimes we get lucky and stumble upon a fantastic tool, we must remember that not all projects are ready for the tool. The tool isn't always appropriate for every team or codebase; eventually, it'll phase out as they all do.

So what's the takeaway? We can accomplish a **lot** with very little code here. The utility logic doing all the grunt work has been abstracted away, and the library already tests it. We're left with an intuitive application function that gets linked account ids and a series of instructions for how we intend to accomplish that.

We don't need to code a bunch of `for` loops here. We don't need to tell the computer how to do its job. We simply provide a list of instructions based on a type we've created, be it a strict type or implied type, via an object dictionary. You can see how we may want to put validation ahead of this, either by type assertion or schema validation, as we did with predicates earlier.

Ideally, we don't want our composition to handle too many edge cases because we litter the story with protective instructions that don't change the intent. Still, you can see how easy it is to do with lines like `R.defaultTo([])`, which is essentially just going to ensure that if the previous line *returns undefined*, the following line doesn't *receive undefined* because `R.flatten` expects an array; most likely an array of arrays.

Summary

Curry and compose are cool, but what about all the other FP paradigms? What if our project isn't using a functional language; maybe PHP, Python, or Java? We can absolutely use FP patterns in all of these languages. Transitioning from Java to Scala is a real and tangible scenario if we want to leverage FP's power more but want to stay on JVM.

Not only do functional patterns and principles help simplify the codebase and deployment efforts, but they also enable serverless cloud-based delegation. AWS Lambdas, for example, need to be fire-and-forget, so the sooner we stop caring so much about state in all our logic, the sooner we can quickly punt tasks to the cloud and ease the load on our system.

From here, I recommend you try to get familiar with more FP concepts like Monads, Monoids, Functors, Lenses, and Applicatives. When you're ready to take your JavaScript FP game to the next level, check out the JS library *Sanctuary* and read up on the *Fantasy Land*^{*} spec and see how bullet-proof your Node and JS projects will become.

And lastly, if you're concerned that this chapter might be taking you too far out of your comfort zone, rest assured that a high percentage of projects that I work on don't have any FP in them. Some projects aren't appropriate for FP, like video games. Additionally, clients often set guidelines and restrictions. While I can recommend improvements and patterns, they can't unjustifiably disrupt the business, so sometimes Java stays Java, or Ruby stays Ruby. We keep the patterns already in place, so our new code isn't some parasite that doesn't match the rest of the repository.

The nice thing is once you ramp up this skill, you'll have that much more leverage to pick and choose the projects you want to work on, so you can do what makes you happy, whatever that may be.

^{*} <https://github.com/fantasyland/fantasy-land>

CHAPTER TWENTY-EIGHT

Version Control

Years ago, the software development life cycle was much more linear and circular. A developer could be responsible for a file or an entire codebase. The change would be made, the code would be compiled or deployed, and it would run on whatever system needed. Changes to the code had to be applied back at that source code level on that original developer's machine, creating a circular lifecycle.

As systems became more involved and required more developers to collaborate, so did the need to figure out the logistics of how those devs could all contribute without tripping over each other. If each developer owned specific parts of the system, then each dev could contribute *their* part, and the project could be assembled and deployed just as if one dev had built the entire system alone.

As systems became more *distributed*, so did the need to adapt the workflow. What if a subsystem had a dependency that was being built by another dev? How can one dev fix a bug on a file authored by another dev? Early version control systems provided a way to "lock" files in a repository, so devs could essentially "check out" a copy of the file. When they were done making their changes, they could "check in" their copy which would update the master copy. No other devs were allowed to check out a file that was being worked on by another dev.

This was an acceptable model so long as no two devs ever needed to

simultaneously work on the same file. If that were the case, it would require the devs to pair up physically and work on the same file together or figure out a schedule that worked for them so one would wait for the other.

Eventually, version control systems (VCS) evolved to the point where there was enough code merging intelligence in the software to understand and detect when two developer contributions were unrelated or if they overlapped and conflicted. Handling unrelated changes in different parts of the file meant the VCS could figure out how to accept both changes simultaneously and intelligently generate a new master copy that reflected both changes. If multiple devs made unique changes to the same code area, the VCS would need to handle that somehow. Which dev needs to try again? Which contribution is the better contribution? Ultimately, that type of decision-making needs to be handed back to humans, and that's where we are today.

Today's version control systems like Git can convey these conflicts and offer brilliant ways of essentially walking the developer through the conflict resolution process, step by step, asking for human intervention where it makes sense, and then quickly getting the code to a place where all the multiple contributors' intentions have been presented and merged intelligently. What a huge responsibility for one tool.

As with every chapter in this book where I mention tools, I want to emphasize that while some flavors of tools will align better with our preferred workflows, our value as consultants comes from the graceful ability to use whatever tools the project demands. On my own projects and the majority of my client work, I will be using Git and hosting the repositories on GitHub because that's currently such a widely adopted paradigm, and it works great for my needs. As a bonus, GitHub has recently added a vast suite of additional features like GitHub Packages and GitHub Actions, so we're able to significantly simplify our workflow and replace the need for additional tools like npm Enterprise or CircleCI.

However, if we follow the 80/20 rule, we can assert that 80% of our projects will require 20% of our effort, and 20% of our projects will

require 80% of our effort. This is very accurate as one-in-five clients will typically bring their own VCS to the table. Unless the VCS is causing friction and challenges for the client, there's often no technical reason to suggest using anything else. Remember, we don't want our clients to accommodate us; we want to accommodate them.

VCS

Let's go over some standard version control systems (VCS) that we might encounter. This won't be an exhaustive list, of course. This is just to keep your mind open that the world of version control is not simply GitHub. I'll also be using the terms repository and repo interchangeably.

GIT

We might as well start with Git since we've already been talking about it, and chances are you've used it in some shape or form. Git is not GitHub. Most seasoned developers know this, but it's one of those things that you don't know until you know, similar to knowing that Java and JavaScript have no correlation whatsoever or that Microsoft has its proprietary flavor of JavaScript called JScript. Git is free and open-source software for distributed version control.

The primary benefit of Git is that there's no central repository. It's distributed, so each clone of the repo is an official repository in its own right. We can create repos and branches and decide which branches should be merged into other branches to offer a source of truth that makes sense for our project. These repos can live locally on our computer or remotely on another machine, intuitively referred to as a "remote." Remotes are non-local origin pointers that allow the Git software on our machine can talk to the Git software on the remote machine.

We can pull, push, and merge changes in either direction how we see fit, though typically, that's managed from the origin because devs usually work on the origin and deploy to a remote. Devs can also pull in changes deployed by other devs from that shared remote pointer. Git offers various ways we can apply these merges, too, like squashing and rebasing, which are ways for us to zip up a bunch of sporadic commits that convey a single contribution intention and keep the repo's `master` branch history in line with our release timeline.

When we have a branch in a state that offers stability and warrants a release to the public, regardless if it's the `master` branch or a feature branch, Git allows us to tag the branch, which essentially creates a snapshot. Then in our CI pipeline, we don't need to constantly move code around because we can tell each system in each tier to pull a particular tag from a specific origin. This allows for a distributed working model funneling into a linear release model.

Git offers a very friendly yet highly robust CLI, but if you prefer to use UI tools, some Git clients can leverage that CLI behind the scenes for you, like Sourcetree, Tower, or GitKraken. If you use an IDE like IntelliJ or an editor like VSCode, you can wire up Git commands as well, so you can click buttons in your UI to issue common Git commands against your project.

MERCURIAL

Mercurial is another distributed version control system similar to Git. It was developed in Python and has some pros and cons compared to Git. One significant difference is that Mercurial doesn't allow us to change history. We can amend our last commit, but we can't do a hard reset and move the head pointer as we can with Git. So if you're brand new to version control, you may benefit from a safer workflow that Mercurial offers. Once you become more advanced and comfortable, you can transition to Git and take advantage of the extended feature set.

Another difference is that where a branch in Git creates a commit hash forked off another commit hash, Mercurial branches refer to a linear line of consecutive "changesets," which are essentially a complete set of changes made to a file in the repo. Mercurial embeds the branches in the commits, where they are stored forever. This means that branches cannot be removed because that would alter the history, which is not allowed.

This changes our workflow options when we consider feature branches, squash commits, etc. Sometimes on projects, we may need to apply a hotfix to an existing tag that's been deployed to production. In Git, we

could create a branch off the tag, apply the fix, merge the commit to the tag, and re-tag with a new SemVer release number. Then, we could cherry-pick that commit and submit a pull request to merge it into the `master` branch of our development codebase. Mercurial would require a different flow, likely involving creating a patch diff. Process variance isn't the end of the world by any means. It's essential that we're aware of it and where our gaps reside so we can study up on some syntax and make sure we're using the current best practices for whatever task presents itself. Mercurial is phasing out quickly in our industry, so I would never recommend it as an option to a new client, but we need to be ready to interface with it when a client is already using it.

SUBVERSION

Subversion (SVN) differs from Git and Mercurial in that it's not a distributed system. It's essentially a server that manages the source code all the devs must pull from and commit to. This introduces many challenges I mentioned earlier in this chapter involving devs potentially blocking or clobbering each other. Conflict resolution isn't nearly as intelligent or intuitive. This is yet another VCS that I would never recommend to a client, but some clients are stuck with it, so such is life.

One convenience that comes with SVN is a Windows client called TortoiseSVN, which is implemented as a Windows shell extension. Essentially, you have your typical Windows Explorer file directory for your project. Right-clicking files will display additional menu options allowing you to pull and commit your file to the attached SVN server. If you're a windows developer, this is a convenient workflow because it hooks into menus you already use daily.

Of course, SVN, just like Git and Mercurial, is going to be supported in a range of extensions, clients, and plugins for your IDE or editor, so at the end of the day, if we can leverage these hooks and extensions as much as possible, then we don't have to leave our typical workflow to accommodate any tool that we don't usually use.

In the same way that we can tell our IDE whether to insert a `TAB`

character or any number of spaces when we push the TAB key on our keyboard, we can assign our "pull" and "push" buttons on our development environment tools to use whichever VCS is required for the project behind the scenes.

TFVC

Taken verbatim from the Microsoft documentation:

Team Foundation Version Control (TFVC) is a centralized version control system. Typically, team members have only one version of each file on their dev machines. Historical data is maintained only on the server. Branches are path-based and created on the server.

TFVC has two workflow models:

Server workspaces - Before making changes, team members publicly check out files. Most operations require developers to be connected to the server. This system facilitates locking workflows. Other systems that work this way include Visual Source Safe, Perforce, and CVS. With server workspaces, you can scale up to very large codebases with millions of files per branch and large binary files.

Local workspaces - Each team member takes a copy of the latest version of the codebase with them and works offline as needed. Developers check in their changes and resolve conflicts as necessary. Another system that works this way is Subversion.

The key takeaway from that excerpt is that this VCS is similar to Subversion.

VCS Hosting

VCS Hosting is different from VCS. We need a way to remotely manage our repositories and configure team permissions, branch restrictions, hooks, and triggers.

GITHUB

GitHub is a cloud-based hosting service that lets us manage Git repositories. Typically we would start a project and assign the Git **origin** repo to our computer and the **remote** to the GitHub URL. Similar to a locally installed Git client, GitHub offers some buttons and UI tools to handle running Git commands against the remote repos.

The typical value that GitHub brings to our workday is detecting branch pushes and handling pull requests and code reviews. It's a brilliant and well-built system that allows devs to collaborate, suggest changes, apply and merge changes through the UI, and run GitHub actions behind the scenes like unit tests, linting, or static analysis via SonarCloud or something similar. GitHub also has very intuitive and robust permissions and organizational features that make structuring and assigning to projects very seamless.

It doesn't matter what type of project we are building. I have GitHub repositories for many project types, from websites, platform APIs, serverless runtime logic meant for AWS Lambda, mobile games, Arduino projects, and more.

GITLAB

One popular alternative to GitHub is GitLab. Where GitHub offers "Actions" to create a CI pipeline, those actions often point to third-party vendors. GitLab offers its own CI pipeline and presents itself as more of a DevOps-friendly version of GitHub. There was a time when GitLab

far surpassed GitHub in the DevOps space, but GitHub has significantly caught up. At this point, I offer that GitLab is a lateral alternative, and the pros and cons will be superficial at best. We can recommend either tool to clients, and our workflow won't change *that* much.

BITBUCKET

BitBucket is a repository host created by and is part of the Atlassian suite. It can make sense for clients to want to go all in with Jira, Confluence, Bitbucket, and the other tools because they link up nicely and offer cross-tool interactions and intelligent reporting and metrics. It used to support both Git and Mercurial projects, but it deprecated support for Mercurial on July 1, 2020.

For a while, BitBucket was the go-to platform for startups because it offered unlimited free private repos with a limit on contributors-per-repo. The other tools, like Jira and Confluence, weren't so price-generous, so it was typical for a project to only use BitBucket and then use another free issue-tracking product like Trello. So if you had a small team and didn't want your codebase public and open source, BitBucket gave you a simple free option where you could still build through a collaborative model without accruing expenses before your product made any money.

GitHub has since changed its pricing structure such that devs can get unlimited private repos as well. This means I would currently also consider BitBucket as a lateral alternative to GitHub. When considering Git repo hosting for your project, choose whichever you prefer, but when conveying options to clients, providing a list of three or four interchangeable tools changes the conversation a bit. The deciding factor becomes more about pricing and features and less about engineering velocity or integration concerns.

AZURE

Azure Repos is part of the Azure DevOps (ADO) suite. It supports both

Git and Microsoft's proprietary Team Foundation Version Control (TFVC). To keep it brief, it's just a tool, and it's good to know about it as an option. Similar to BitBucket being a convenient choice if the project uses the rest of the Atlassian suite, this same principle applies to Azure Repos and the ADO suite.

As I've said before, I typically try to avoid using a full suite of things arbitrarily because I prefer to rely on patterns, not products. Still, if it makes sense for the business, then it makes sense for the business, and we should be willing and ready to support those business needs. Oftentimes the DevOps team will prefer to use all the cloud tools from a single provider so there's no reason to die on this hill if a particular suite of tools will help another team.

ASSEMBLA

Assembla is a service for managing hosted Subversion repositories. It allows for remote repository and team management features like permissions, hooks, etc. There are other competitors like Perforce that offer a similar set of tools. Suppose you encounter a project using SVN and need to interface with the service layer for configuration and management. In that case, you'll want to cross that bridge when you get there because there's likely a well-established process, and you'll need to abide by that process. In other words, I would never recommend Assembla for a new project just like I'd never recommend SVN for a new project. That said, the likelihood of working on a legacy client project with an existing SVN and Assembla setup is high, so we need to be aware of these tools and services, even if they are nearing EOL.

Binary Accommodation

One major caveat I want to cover here is that while we have several options for handling versioning of our programming projects, distributed VCS like Git or Mercurial are **not** meant for large binary files. Because we will have a local copy of the repo and all the changes on our computer, we can fill up our disk very quickly as we start making minor changes to large files because every change will save a unique copy of the file in the Git history.

Git offers a specific extension for this called Git Large File Storage (LFS). Essentially it'll make a compromise where it saves the binary diff in the local Git history and then saves actual binary files on a remote repository so you can offload that disk strain. If you think about it, this is really just Git's way of pretending it's a centralized VCS. In that sense, a hosted SVN like Assembla would perform much better for large binary files. It makes sense for a project to separate its source code and versioned assets into two repositories and VCS engines.

Azure's TFVC and Perforce's proprietary VCS, called "Perforce Helix Core" are advertised to handle versioning large binary files in an optimized way. I can't personally speak to whether those services can back up their claims, but hopefully, you can see how having options is more important than knowing the correct answer.

With every new challenge, we get a unique opportunity to comprehensively analyze available tools and see which ones make sense and which don't. Some tools and services will yield the ideal results this year, and their competitors may deliver them next year.

CHAPTER TWENTY-NINE

Data Persistence

In this chapter, we'll talk about different areas of our application infrastructure that will likely be responsible for storing various types of data for different reasons. This will be a light overview for two reasons. One, there are thousands of books and tutorials on all this stuff I'm about to cover, and it doesn't make sense to consolidate all of it here in this book. We'd be here for hours in this one chapter. The other reason is that throughout this book, I try to maintain that we want a "jack of all trades, master of some" philosophy with our tools and syntax.

We want to continuously know *just* enough to recognize variables and options to offer genuine consulting and troubleshooting assistance to our peers. We also want to free ourselves from trying to maintain expert-level competency in any given tool until we start a project that actually needs it. Then, we can deep dive into the tool and fill in the gap, so our high-level understanding becomes low-level understanding.

I'm not concerned that you know how all these tools and approaches work at this point; just that they exist and whether or not they're a viable option to consider in your current project or at some point in your career. That said, we're going to cover a **lot** of examples, so if you get lost or intimidated, rest assured your typical project will only use one or two of these mentioned examples (or another approach altogether). Just take it all one step at a time and keep it at a high level where you can.

* * *

So what is data persistence? Simply put, it's capturing some data somewhere besides memory, so as the user interacts with our application and the state of the app changes, the data remains intact and can be pulled back into the UI. I want to break up data persistence into three main categories for this chapter: **Short term** and **long-term**, where the latter can also be split into **dynamic** and **static** persistence.

Short Term

Short-term data persistence can also be thought of as **high-risk volatile** data persistence. It's most often not used for any historian or record-keeping requirement but rather as a means to an end to satisfy some other business or infrastructure requirement.

One common example of short-term data persistence is user authentication sessions. When a user logs into our application, we need to keep a record somewhere that lets us know the user is active and authenticated so that we can serve up the appropriate user interface and enable the proper API endpoints.

Two common variations of session persistence are **cookies** and **token profiles**. Cookies are basically encrypted files created by a web server that include some identifier data, often a session ID. The cookie is sent back to the browser or application client in an HTTP response, and the client then attaches the cookie to all subsequent requests. The server recognizes the cookie, understands how to read it, and can look up the included session ID to verify whether the user is indeed logged in.

Cookies can contain an abundance of information besides just a session ID, such as metadata we may want to include for sales funnel tracking and cross-site partnership purposes. Because they're essentially attached files, they can noticeably slow down HTTP requests and bog down a website experience or an application waiting on web API responses. Nowadays, more and more governance is being applied to user privacy, and websites are forced to include cookie policy disclosures and opt-in messaging.

Consider we build a website, and someone on our website views a product, adds it to their cart, and then doesn't complete the purchase. We could set a cookie on the user's browser that could carry over to other websites that share the same ad network so the user could see ads for that exact product on those other websites. Clicking the ad would take them back to our checkout experience so they could complete their purchase.

* * *

You've probably experienced something similar. The trick is that the browser binds cookies to their issuing domain, so cross-site cookie persistence comes from websites pulling in JavaScript that will fetch the initial cookie so every site with that same JavaScript will talk to the same domain, and the same cookie can be utilized.

So while the session ID needs to live in the cookie, it also needs to live on the server somewhere to look up when we receive a request containing the cookie. For this, we have some options. If we have a monolith server that's not behind a load balancer, we can store them in memory if we want. It's not the most ideal, but it's a realistic option. If our server is part of a cluster, then we need to store it higher up in the stack, like in the server database or a cache database.

For example, we can create a table in our database with an intuitive name like "sessions." As users log in, we insert a record, and as users request pages on our site, we look up the session ID in the cookie to find a match in the table. If it exists, that's usually all we need to allow them through, though we can always add additional security layers to compare a one-way hash or something. We have a few options here, so typically a team discussion with SecOps is appropriate to land on something that makes sense for the project.

If we don't want to use our primary database to store sessions, we can use another tool like Redis or Memcached, which are essentially key/value databases that live in memory. In cloud architecture, the provider can spin these up on SSD, so it's as fast as living on a stick of RAM. The idea, though, is that our server cluster is entirely separate from the cache database so that we could have twenty versions of our web server running, and no matter where an HTTP request lands in the cluster, every node in the cluster will ask the same instance of the cache database to find a record matching the session ID.

If we don't have a tool like Redis or Memcached, we can use a file cache on a CDN. Storing micro-size text files on Amazon S3 where the file name is the session ID is acceptable, though not nearly as fast as a key lookup in a cache database. I say it's acceptable because it works. While

it's not optimal, it's still an option that we can entertain, even if only to unblock our product while we figure out options for the ideal infrastructure behind the scenes.

Another type of authentication-based profile persistence is token-based auth, where the user's session profile would reside inside an encoded payload represented by a token. The current trend is JSON Web Tokens (JWT). Some people pronounce them "jots," which I think is as ridiculous as pronouncing GIF "jif" even though graphical has a hard "g." But I digress. The token is typically passed back and forth through HTTP in a query string or an [Authorization](#) header.

For this approach, when the user logs in, we fetch the minimum-required public-appropriate profile details from the database and save them somewhere on the back end for the duration of the user's session. We can leverage the same options as the cookies by storing the profile in our preferred cache tool. We will **encode (not encrypt)** the JWT profile details into a long string. Then, we'll create one additional token called a **refresh token**. We'll save the JWT details in the cache and use the refresh token as the key for that record.

We'll send both the JWT and the refresh token in the HTTP response. Because the JWT is not encrypted, it will be prone to man-in-the-middle attacks, so we want our tokens to be extremely short-lived. Our server can trust every HTTP request that includes the token because it likely signed the token with a private key, and we can verify that it's both our token and that it's not yet expired.

Once the token expires, we will have to send back a 401 response code, and the requesting client will need to refresh the JWT. For that, it'll make a refresh request and attach the refresh token we gave when the user logged in. We lookup the JWT in our cache with the refresh token as the key, and if we find it, we delete the JWT from the cache, generate a new refresh token, write the JWT back to the cache with the new refresh token as the new key, and respond with both the new refresh token and a new JWT.

This might sound like a complicated series of handshakes, but this is the

reality of token-based authentication systems. This chapter isn't about auth, though. The lesson here is understanding how the user who logged in has detailed profile information living elsewhere in a database. We're using a volatile cache layer solely to store session profile details for blazing-fast lookups to deliver a low-latency, safe, and secure user session experience.

If that repeated refresh/profile-lookup flow happened at the database level, we could realistically strain our database and bottleneck the throughput, so our application queries would be much slower. Offloading this burden to an in-memory cache database solves this technical limitation regardless of which tool, syntax, or pattern we decide to use. Principles like the "separation of concerns" and "least privilege" pop up all over the place and help us succeed almost every time, case in point.

Long Term Dynamic

Long-term dynamic data persistence is typically considered **low-risk volatile** data persistence. Both terms are subject to debate, which will become clear shortly.

RDBMS

The most common long-term data persistence layer in most software applications is RDBMS, which stands for Relational Database Management System. Currently, the most popular relational databases in our industry are Oracle, PostgreSQL, MySQL, and Microsoft SQL Server.

Oracle is a huge enterprise-level database that I personally believe has overstayed its welcome. It's very expensive and not nearly as influential nowadays when cloud providers can offer auto-scalable managed databases compatible with multiple syntaxes of open-source variants. AWS RDS is a perfect example where we can use MySQL throughout the entire development and release lifecycle. We can run MySQL in a local Docker container, and we can run MySQL in RDS on the cloud that will scale to accommodate millions of rows serving millions of users. I've had numerous debates with colleagues who feel PostgreSQL is a superior database. You can probably guess where I stand on tool debates. That said, I wouldn't personally choose PostgreSQL over MySQL on any of my projects because MySQL has caught up with JSON support, and I prefer the way it handles `AUTO_INCREMENT` ids versus PostgreSQL `SERIAL` ids which requires that extra step of calling `pg_get_serial_sequence`.

That said, using a UUID as the primary key instead of an incremented numeric ID for our tables removes that gap anyhow, so at the end of the day, these are just tools, and the snapshot of capabilities from any of them don't really matter in the grand scheme of things. Microsoft SQL Server would likely only be used on a Microsoft project where everything else in the stack is Microsoft, so if you're in that world, your options are minimal but clearly defined.

* * *

There are endless comparisons among the databases, like choosing a storage engine between MyISAM and InnoDB for MySQL. 99% of the time, this stuff is up to the client or employer. If we have the authority to make stack decisions, we should simply create a quick spike ticket so we can ramp up on the latest best practices and go from there. For example, MyISAM *used* to be better than InnoDB, but InnoDB caught up and is now widely accepted as the superior storage engine. **What we know to be true today may not be true tomorrow, so stay fluid and flexible with tooling.**

The upside of RDBMS is that we can clearly define our data models and schemas well in advance. We can easily set rules for our code when creating our validation layers or types that adhere to our schemas. RDBMS also allows many logical capabilities like stored procedures, though I recommend avoiding stored procs on your projects.

If your project currently has stored procs, I recommend immediately creating technical debt tickets to rip those out and replace them with code. It's a nightmare when business logic is split between code repositories and databases. Trying to track down what caused certain records to update or disappear can be highly time-consuming if we don't even know what parts of the infrastructure are technically capable or responsible for certain activities.

My philosophy is **business logic needs to live in the code**. It must adhere to a release cycle to be reviewed, tested, or rolled back if necessary. Ask a DBA, and they'll often have the opposite philosophy. To each their own, I do not support a model where a DBA can update a stored procedure in a production database and change business logic in real-time without that update being vetted and scrutinized by code reviews, QA, etc. It's essentially doing an FTP force push of code to the production server.

Cue Bill O'Reilly: "We'll do it live!"

If we don't have a say in the matter and stored procs are just a way of life for our project, then I recommend trying to work out a compromise

where the DBA or DevOps contributor only ever puts stored procedures in database migrations. DB migrations are tuples of commands that get run on releases and rollbacks. One-half of the tuple is the "**up**" command which is the instruction that needs to be applied moving forward. If we're adding a new table to handle purchase orders, for example, we could imagine that the "up" command might be the SQL to create a table called `orders` and define the columns. The other half of the tuple is the "**down**" command, which would essentially undo whatever was in the "up" command. The "down" command would drop the `orders` table in our example.

These "up" and "down" commands could also be SQL to create or drop stored procedures. Regardless of which team authors the SQL at that point, the stored proc would be eligible for a code review process like the rest of our codebase. We could add test coverage around models that depend on the procs or which might trigger them, and we could tightly couple documentation with the stored procedure since it will live in code. Again, **we want our projects to be stable, intuitive, and organized.**

If your project currently includes an RDBMS and you're not currently taking advantage of database migrations, I *highly* recommend you start immediately. Do a quick Google search and find a library that caters to your language. The approach to wiring up support for migrations varies across languages and frameworks. For example, npm offers a bunch of libraries for NodeJS, which can handle this for various databases, and even provides database-agnostic libraries. The one I've used for years on multiple projects is called `db-migrate`. Ruby on Rails has migrations built-in with Active Record, and CodeIgniter for PHP has native support for Active Record database migrations. Still, it's disabled by default, so you must activate it through your config.

Suppose we are building software for an OS. Regardless of whether it's a desktop or mobile environment, we will have some databases available to us through various SDKs and modules. Ideally, we aim for tooling that allows us to deploy to a wide variety of platforms without having to accommodate esoteric nuances. For example, where Windows and macOS may present significant syntax gaps, we won't experience

much pain between macOS and Linux because they're both Unix-based operating systems. Android and iOS will have some native gaps, but wrappers like React Native or Cordova can offer standard APIs and SDK options. Abstraction engines for game development like Unity can provide a single code source to be deployed to multiple platforms with little effort.

For example, on macOS, we can use Swift to interface with Core Data for our persistence layer. Suppose we want that code to also work on Windows. In that case, we need to start looking at tools like Cocotron-- a developer SDK that implements a usable amount AppKit and Foundation for Windows and Foundation for Linux/BSD in Objective-C. It requires cross-compiling via Xcode on macOS.

If we want to take a more straightforward route, we could change up the database and use the most popular option in the world, SQLite, since it's built into so many operating systems. It's a super lightweight, fully-featured database library that supports SQL syntax. It allows us to generate a self-contained database that resides in a file with our project. It's a transactional database that supports rollbacks as well, so if our code is talking to this one database, and the database lives with our project, then our project is that much more portable, meaning cross-platform accommodations can reside higher up in the stack, which is ideal.

It also makes it really easy to sync our project across multiple pieces of hardware or offer a backup feature by linking to a file hosting service like DropBox or something similar. Simply zip up and encrypt the database file and applicable config files and push the archive to the remote folder. Then, on a new install of your app, supporting an "import and restore" feature becomes very simple.

Additionally, suppose our desktop application offers additional cloud features besides backup or sync (maybe we want to leverage the cloud for license and subscription management). In that case, we can align our database schemas a bit so our server-side RDBMS and our embedded SQLite database adhere to the same schema rules, and data can be leveraged on both ends of the stack in a consistent and normalized way.

We can treat the server-side RDBMS as the source of record and the client-side SQLite database as a "snapshot" of sorts.

RDBMS isn't the most elegant solution when trying to support two-way sync. Maybe the cloud layer also offers a web interface where users can split their interactivity between the web and the desktop. If we want those changes to be reflected on both ends of the stack, then a NoSQL tool like CouchDB or Couchbase Lite may work better. Built-in replication can take care of a lot of that overhead for us.

NOSQL

Sometimes our data can't adhere to a strict schema. Sometimes we want our schema to evolve as our business grows. Maybe we want a database that can store and fetch the data as a complete record payload, and maybe we don't have a critical need to index on specific columns. In this scenario, it can make sense to consider a NoSQL option. RDBMS databases like MySQL do have a JSON column type which allows us to store a JSON blob in the field. We can also create indexes on nested JSON props *inside* the row value for that column as well as enforce valid JSON syntax, so we don't inadvertently store broken JSON in our database. Behind the scenes, this will create new temporary tables, and we're not really saving any processing time or disk space. Still, we benefit from simplifying our infrastructure at the expense of adding query and DB organization complexity.

If we want a distinct abstraction between our relational and non-relational data, we can simply choose a standalone NoSQL database for this JSON blob storage. There are plenty of options like any other tool in our career. Some popular ones are MongoDB, CouchDB, Redis, Firebase, and Cassandra.

When implementing a NoSQL database, it can be tempting to fly by the seat of our pants and just let the database do its thing as an afterthought while we focus on building our application. Inevitably, the database will bottleneck like any other, so we need to think ahead. One of the benefits of NoSQL over RDBMS is that we can **scale laterally instead of vertically**. RDBMS lateral scaling becomes increasingly difficult because

it relies on `JOINS` and indexes involving foreign keys. Splitting relational data across two MySQL databases can noticeably slow down queries. NoSQL is typically non-normalized, so data redundancy is often prevalent in these systems. Our option to scale laterally becomes *sharding*, where we can run multiple copies of the same database but subsets of data live on each one.

For example, we may shard on the `city` property of the `Users` collection. We can have users living in a city that starts with the letters A-M in one shard and users residing in cities beginning with N-Z in another shard. New users added to our collection will be stored in the appropriate shard depending on the first letter of their respective city. We could have as many shards as we want, broken up intelligently to satisfy our business constraints. You can see how if we had one million users broken up into twenty shards (so each shard only had 50,000 users), a query could be substantially faster because once we're in the appropriate shard--we only have to traverse 50,000 users instead of a million.

This extreme example proves that **we have options to improve performance as we begin to outgrow our database**. Typically we only shard when we reach bottlenecks, and we only shard where it makes sense to alleviate the bottleneck. Over a few years, we may only have 2 or 3 replica sets and 4 or 5 shards, but every project is unique. Hence, we need to read the docs when the time comes to see what the applicable version of whatever NoSQL tool we're using recommends for the process.

NoSQL is great for isolated services as well. For example, we can run batch jobs to replicate subsets of data we would like to be searchable and import them into a tool like Elasticsearch. We can toss whatever schemas we want into it, and we really only need to worry about schema definitions when setting up the search indexes for performance optimization.

This can all be done *after* all the data is loaded into the Elasticsearch database as well. We can constantly adjust our search performance and even get limited queries responding in real-time. We can search millions

of objects and return typeahead results for autocomplete suggestions or search field menu recommendations. We can treat this NoSQL database as volatile even because our source of record would reside in another database.

For example, we likely want to omit certain records if they are off-limits. They may have been soft-deleted, pending approval, or in a non-production state. Instead of including all those rules in our search query, we omit them entirely when populating the search database. Every few days, we could run a clean-up job to delete the search database completely and port over everything fresh again.

That would certainly change how we set up the search database, right? It would make sense for us to have separate search databases for users, orders, products, and FAQs to avoid regularly updating search records that rarely change. You can see how **different tools open up different options**, so we always want to be flexible and fluid when working with these tools so we can orchestrate them to make our lives easier.

GRAPH

The last type of database I want to touch on is a graph database. Unlike RDBMS or NoSQL, graph databases store data on **vertices** and **edges**. They are all about defining and mapping out deep relationships across data points. For example, where we might ask a relational database to get "*all the orders where the customer ID is 123*", we might ask a graph database to get "*the top three products most frequently purchased by customers where those customers purchased at least one item that customer 123 has also purchased but do not include any products customer 123 has purchased*."

Graph databases are commonly used in social media models where we want to understand relationships not just across friends but also *friends of friends, friends of friends of friends*, and so on.

Consider this excerpt from Neo4j News:

Depth	MySQL	Neo4j
2	0.016	0.010
3	30.267	0.168
4	1,543.505	1.359
5	> 3600 ?	2.132

In their new book, *Neo4j In Action*, Jonas Partner and Aleksa Vukotic [...] built this query in both MySQL and Neo4j with a database of 1,000,000 users, and the results are striking. Execution Time is in seconds, for 1,000 users.

For the simple friends of friends query, Neo4j is 60% faster than MySQL. For friends of friends of friends, Neo is 180 times faster. And for the depth four query, **Neo4j is 1,135 times faster**. And MySQL just chokes on the depth 5 query.

The results are dramatic. Read about the whole experiment in the first chapter of *Neo4j In Action*.

So a graph database clearly has a use case where it shines, but does it make sense to use one as our only database? Like all tools, it depends on the project. At this time, probably not, unless the entire product is a graph reporting or analytics tool.

Consider the e-commerce application I mentioned earlier, where we may want to show recommended products. We probably wouldn't want to store the complete customer orders in the graph database along with credit card and transaction details because most of that data is meaningless to the graph, and the principle of least privilege tells us that if a tool doesn't need sensitive information, then it shouldn't have it.

We also probably don't want to exclusively store IDs in the graph because then, as we run our graph query and find all the applicable IDs for our search, we would then have to do subsequent batch queries in our RDBMS to get details we may want to send to our UI or report.

There's likely a middle ground where the graph stores some redundant data that's meaningful to its use case, similar to how our NoSQL Elasticsearch database would store some redundant data that's meaningful to search results.

One caveat in the graph database world is that there's no industry-standard query language like SQL. We have several options depending on the database. For example, Cypher or Gremlin are recommended for Neo4j, nGQL for NebulaGraph, GSQL for TigerGraph, and others like PGQL, G-CORE, or Morpheus for other databases out there.

Ultimately, what I want you to learn from this, is that no matter which type of database you choose to store your data, you want to leave yourself a means to use an alternate database either as a replacement or as an augmentation if you happen to change your mind or if the needs of the product change.

Whenever we're choosing tools, **we want to recognize our options and also leave our options open so we can iterate and adjust as the project evolves**. A data migration from MySQL to MongoDB or vice versa isn't the end of the world. It's just a pain in the ass. If we can anticipate that the migration is a real possibility, then maybe we can leverage an ORM like Active Record or Sequelize in our application layer to **separate our query syntax from our query intentions**. Again, I don't recommend or against this approach. ORMs have pros and cons, like every tool. Some projects benefit from them, and some don't. It'll be up to you or your team to decide the appropriate approach for your project.

Long Term Static

Long-term static data persistence is typically considered **low-risk stable** data persistence. The two most common long-term static data persistence options are **data lakes** and **data warehouses**.

DATA LAKE

A data lake is where all the raw, **unstructured** data can reside permanently, agnostic to any particular business needs. For example, we could store raw HTTP POST requests in JSON files in the data lake before inserting them into our RDBMS database. In the future, if we want to replay HTTP requests and run them through simulation tools or monitoring tools, we could. This is a helpful approach if we want to reproduce a user's round trip experience or if we want to send last month's traffic through a load test tool like JMeter and see how our proposed infrastructure improvements might handle the same traffic load.

The most common use for a data lake is if we have multiple reporting services requiring similar input in slightly different variations. Their **response** data looks nothing like the **request** data. It's nice to keep that source data around in case we want to try out new vendors and provide historical data in a format that satisfies *that* vendor's API contract. Suppose we didn't keep that raw data around. In that case, it could be tough to "unwind" previously decorated or transformed data and get it in a format that would satisfy another vendor.

In other words, imagine we have twenty widgets that all display different reporting metrics from the same data set. Each widget could pull raw data from the data lake, decorate it how they need to, and save their generated report separately for our needs. If we ever wanted to add a twenty-first widget, it's straightforward for us to do.

Data lakes can also store raw binary files, like user-uploaded images or videos. We can keep the raw uploads in the data lake. Then with cloud

tools like AWS MediaConvert, we could pull those assets from the data lake and generate whatever variants we want, like compressed or resized images, watermarked videos, etc. It's hard to remove a watermark from a video once it's there, but if we keep the raw video around, we leave our options open.

I often recommend a simple cloud object storage for the data lake, like Amazon S3 Glacier or Azure Storage Archive. When we store the raw stuff in the data lake, we want to name the files and directories to line up with reference pointers in our database or consistently formatted date and time names so we can easily leverage the raw data by traversing through our parsed application data.

We certainly don't just want one big bucket of stuff that collects years' worth of random things in one place, or that would be a nightmare to manage. Organize and name things to make it intuitively simple to retrieve and "rerun" or "retry" business logic with the raw data. It's a fantastic tool for disaster recovery and lets companies change their mind and undo mistakes without reaching out to the users to request new data from them.

DATA WAREHOUSE

Like a data lake, a data warehouse can store archived **structured** data. If you imagine that our cache helps provide more of a real-time access layer to alleviate strain from our database, where the database could repopulate a cache wipe, then extend that concept such that our database provides more of a real-time access layer to alleviate strain from our data *warehouse*, where a data warehouse could repopulate a database.

A data warehouse is essentially an archived backup of our master database that we can also use for complex reporting and analysis. Our database is getting hammered all day by application reads and writes, so running a demanding report on that same database can noticeably decrease our application performance, cause user frustration, and significantly slow down our reporting efforts. Offloading that reporting

process to a data warehouse frees us up to tune our application database *for* our application.

Another benefit of a data warehouse is that we can regularly purge records from our application database that don't matter anymore to the application. Nowadays, data is rarely deleted; instead, it's "soft deleted." That means we somehow flag the record so that it's omitted from future queries but still lives in the database if we need to recover it for some reason.

A user may want to cancel their account, and a few weeks later, they change their mind. Being able to restore that account vs. telling the user they need to create a new one is a convenient capability to offer. Of course, our terms and conditions should disclose this so the user isn't left believing their data has been deleted when it hasn't. If our database gets hacked, and their data is leaked long after they canceled their account, we can only imagine the lawsuit that would follow.

So if every so often we move those soft deleted records out of our database and into the data warehouse, then we can benefit both from the reduced risk of data loss as well as improved performance from a leaner database with smaller indexes.

Another reason we might want a data warehouse is if we ever want to run reporting analysis on data that doesn't help our application do its job. Access logs, error logs, or sales funnel logs could be saved to a data warehouse, so they're not just sitting in a data lake offering no business value. Data warehouses often present a multi-terabyte or petabyte level storage option, which can hook into a third-party query service to get meaningful analytics on otherwise overwhelming streamed data. Some popular data warehouses are AWS Redshift, Snowflake, GCS BigQuery, and Azure's Synapse.

Summary

Choosing a database for your project will rarely involve one person or even one team, so don't feel like you need to own all this pressure. If the burden (or blessing) does fall on you, remember the principles I've been echoing: **leave your options open, keep raw data where possible, use abstractions where possible, and allow yourself to change your mind and iterate.**

Remember, today's popular tools will likely not resemble themselves in ten years. Capabilities, recommended processes, and best practices will probably shift around. The reasons required to change a database might not be because of any syntax mistake or judgment error, but rather a tool got an upgrade that simplifies things, and your client or employer can save thousands of dollars per month by using another tool; so be it.

If, when that time comes, you can relax and rest assured that the migration will be relatively painless because you thought well ahead and set yourself up for success, you'll be able to pat yourself on the back and enjoy the fruits of your labor while other teams who neglected to plan ahead are scrambling and suffering through their work days, wondering how you're able to stay ahead of the chaos so easily.

CHAPTER THIRTY

Code Reviews

In this chapter, we'll cover the process of reviewing another developer's code--what to look for and how to provide feedback that will result in positive collaboration and a quick turnaround for any change suggestions.

Code reviews aren't the most fun part of our job, regardless of which side of the review we're on. If we're tagged to review a peer's code, it can feel like a distraction, and if we're in a hurry and are waiting for our peers to review our code, it can feel like our velocity is at risk.

Ultimately, the trick is to set expectations for the team, mark our availability for reviews, and figure out a process that works for the team on how to notify the devs when a PR is waiting for them.

Understand Expectations

Before doing a code review, we first want to understand our role in the situation. Are we being asked to look for quirks or red flags? Are we expected to be aware of other dependencies or related efforts and consider them when looking at the pull request? Are we expected to pull down the feature branch, run it, and test the code to verify it all works?

We must understand our involvement because once we sign off on a pull request, we're putting our name on the code. This will follow us no matter if the code succeeds or breaks later. We can think of it as if we had paired with the author and are now inheriting shared blame if it breaks the system or needs refactoring. If we assume responsibility for the new code, we want to ensure it's of the same quality had we authored it ourselves. We want our name to be associated with positive improvements to our project, not bugs. **We want to be known for building and fixing things, not breaking things.**

With this in mind, we want to **automate as much of the verification process as possible** so we won't need to worry about it so much that our personal involvement is reduced substantially by the time we get our eyes on it.

Automate The Clean Up

We discussed linters and static analysis tools like SonarQube earlier in this book, and this is where their value presents itself. We can configure rules in our CI pipeline to run them against branches when a pull request is opened, and with GitHub, we can even prevent reviewers from accepting a pull request if the jobs didn't pass.

If we open a pull request for code review and see that our linter and static analysis tools passed with flying colors, our role changes a bit. We're no longer being tasked to review the syntax formatting or the function expression logic complexity because the automated tools did that for us. Our role becomes much broader, like offering pattern cohesion suggestions or recommending replacing some in-line logic with a reference pointer to a shared utility.

Consistency Is Key

The most important thing to look for in code reviews is consistency. We want a codebase in which we can open any file or review any pull request without being able to infer who authored the code. We want every dev to follow proper conventions and every file to reflect consistent patterns and paradigms.

If we open a PR and can tell that Dale is the author because Dale always codes how Dale wants to code, and none of his files match the rest of the codebase, then we have a real problem. I often reject pull requests the moment I see a deviation from project patterns. There's no point in reviewing 500 lines of code if the first ten lines ignore conventions.

If I'm leading the team, I'll often leave a comment like "Please follow project conventions for XYZ. Here is an example to copy: [link]." If I'm on a team and playing the role of a peer, I would likely change that from an instruction to an inquiry, like "This doesn't seem to follow the conventions I'm seeing on other files like this one [link]. Do we want this to match?". This tone is essential when working with client devs because we're technically consultants, and they can do whatever they want regardless of what we recommend. The sooner we know our role, the easier it will be to stay friendly and offer guidance instead of direction.

Keep it brief and professional, and give the author the benefit of the doubt when possible. Sometimes product teams divvy up the engineering team to build features in silos, resulting in different devs getting conflicting or no direction. The author may not even realize that the project *has* conventions, so the last thing we want to do is start a conflict and create a hostile situation by implying the dev is incompetent or "wrong" somehow.

Always imply the process is the problem, not the person. Suppose I'm leading the engineering team and notice that conventions aren't being followed in general. In that case, I'll enforce cross-feature code reviews so the devs can police themselves and share that knowledge among the

team. If it's just one dev struggling to align with the team's direction, then I'll likely prohibit that dev from working on solo tickets, and the dev will need to pair up with other devs moving forward to avoid derailing the project.

The most effortless and painless code reviews often reside on teams where we do a lot of pair programming and shuffle the pairs. The pull requests are pleasantly consistent, and we typically fly through reviews and rarely reject anything. The opposite is true, of course, on projects where each dev gets their own ticket in a silo from the PM; maybe they're working on their own feature and never think to look at any other feature in the codebase.

A project with ten devs working on ten features in ten different ways produces pull requests requiring the code reviewers to figure out what's going on in the surrounding code and make sure everything makes sense. It takes a lot of mental energy to review several pull requests in a day and reverse-engineer each one because nothing in the PR is familiar to what we're used to seeing all day in our own code.

Suggest Refactors

Most devs don't enjoy having a pull request rejected. Suppose it's rejected because of a seemingly lateral change, and it feels like the reviewer only wants the code to be written the way they would write it (irrespective of project conventions). In that case, the rejection can feel like an insult and result in a demoralizing experience for the author. Often the author doesn't really care if a line of code is written this way or that way, but having work kicked back and being told to "change it" when the underlying value doesn't actually change can be frustrating.

One process I've adopted is coding the suggestion myself. For example, I might see a pull request diff like this (keep in mind this is a fake diff I created on one of my repos to demonstrate the suggestion process):

Update platform.js #12

I Open SeanCannon wants to merge 1 commit into master from SeanCannon-throwsaway-1

Conversation ▾ ▶ Commits ▾ Checks ▾ Files changed ▾

Changes from all commits ▾ File filter ▾ Conversations ▾ Jump to ▾

0 / 1 files viewed Review changes ▾

Viewed ...

server/platform.js

```
+ 00 -21 A +51 18 69 loadSearchServer(Search)
  ,then(() => startServers(nodePorts))
  .catch(console.error);
33

34 setInterval(() => {
35   Promise.resolve()
36   .then(Search.batchProcess([Infinity])
37
38   *+ const displayNames = user => `${user.firstName} ${user.lastName}`;
39   + users.map(user => getUsersDisplayName(user));
40
41   setIntervall(() => {
42     Promise.resolve()
43     .then(Search.batchProcess([Infinity]))
```

I would like to eliminate the unnecessary `user` reference on line 34. Line 35 is subjectively arbitrary, and the function on line 36 will accept a user inside that map, so we don't need to notate that explicitly. Let's imagine this project doesn't have a strict and elaborate configuration in the static analysis tool so we still need to manually address things like this.

On GitHub, if I only want to suggest a refactor on line 34, I would click the "+" icon next to the line.

* * *

```

32      .catch
33
34 + const di
35 +
36 + users.map
37

```

If I want to suggest a refactor for multiple lines, I click and drag downward to highlight the lines I care about.

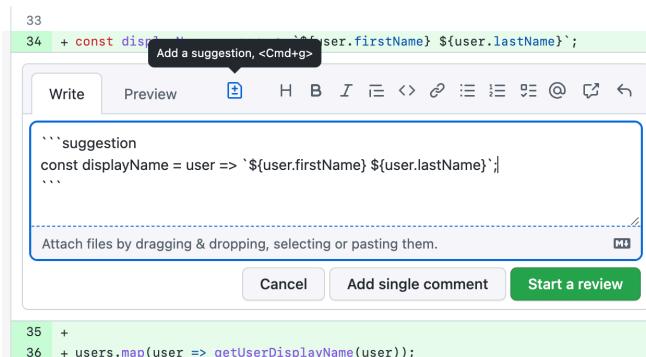


```

33
34 + const displayName = user => `${user.firstName} ${user.lastName}`;
35 +
36 + users.map(user => getUserDisplayName(user));

```

This will open up my comment dialog, where I can leave a code review comment that will be tightly coupled to this particular block of code. I can click the suggestion icon to pull the code into the comment, where I can offer a live refactor suggestion right here in the review:



When I'm ready, I click "Start a review," or if this is not my first suggestion on this review, the button would still be green but read something different, like "Add suggestion to review."

* * *

Succeed In Software

A screenshot of a GitHub pull request interface. At the top, there is a code diff section with three lines of code:

```
34 + const displayName = user => `${user.firstName} ${user.lastName}`;
35 +
36 + users.map(user => getUserDisplayName(user));
```

Below the code, a comment is shown: "Comment on lines +34 to +36". A user named SeanCannon has submitted a "Pending" suggestion:

```
34 - const displayName = user => `${user.firstName}
$${user.lastName}`;
35 -
36 - users.map(user => getUserDisplayName(user));
34 + const displayName = ({ firstName, lastName } ) =>
`${firstName} ${lastName}`;
35 + users.map(getUserDisplayName);
```

Buttons for "Commit suggestion" and "Add suggestion to batch" are visible. Below the suggestion, there is a reply input field labeled "Reply..." and a "Resolve conversation" button.

Once all my suggestions are offered, I submit my review and check the "request changes" option so that the author knows I have changes I'd like addressed. The author can dismiss them or apply them with a single button click for each suggestion:

A screenshot of a GitHub pull request interface, similar to the previous one but with the suggested changes applied. The code now looks like this:

```
34 + const displayName = user => `${user.firstName} ${user.lastName}`;
35 +
36 + users.map(user => getUserDisplayName(user));
```

The "Suggested change" box is still present, showing the original changes. Buttons for "Commit suggestion" and "Add suggestion to batch" are visible. Below the suggestion, there is a reply input field labeled "Reply..." and a "Resolve conversation" button.

I've personally found incredible success with this approach for a few reasons:

* * *

- It eliminates back-and-forth dialogue that can waste time.
- It eliminates any misinterpretation of a verbal comment if I had written out the suggestion with plain words and hoped the author understood how to apply the direction.
- It simplifies the process for the author when applying these suggestions because they don't need to go back to their editor or IDE, code the suggestions, and re-submit the pull request.
- My contributions to the PR are **actual** contributions. I actually helped out, so if the engineering manager ever does an activity audit on GitHub to see who's writing code and who's slacking off, they will see that my activity is very high.
- It keeps my chops up. Coding is easy when we code all day. Leadership roles can introduce huge gaps between programming tasks. Days, weeks, or even months can go by where I'm not "tasked" to code something, and I like to be able to code at any moment as if I haven't taken a day off in twenty years, so it's up to me to ensure I'm finding opportunities to program when I can.

Rule Enforcement

Imagine spending twenty minutes going through a several-file pull request, offering ten to fifteen change suggestions, and substantially improving the overall commit. Then you notice that the pull request has been merged, and none of your suggestions were applied. Other devs approved the pull request without making any change suggestions, and the author didn't feel like dealing with your one rejection because they got two approvals, and maybe that's all the branch requires.

This is, unfortunately, a common scenario on projects where team cohesion and accountability aren't where they should be. The good news is all those suggestions can be attached to a new branch for which we can open a new pull request and get all the credit for simply doing those refactors as a patch commit. It's not the end of the world, but it's pretty annoying, and it's hard to miss a rejection review, so odds are the author skipped the rejection on purpose. The most common excuse I hear is the author felt they didn't have time to make all the changes because the PM was breathing down their neck to deliver the feature.

This is fine, though the process is still broken. I'll typically ask the author then, "Moving forward, please let me know if you don't have time or capacity to implement my change suggestions. I'm happy to merge them separately. If you disagree with the suggestions, that's another scenario entirely. We should have a quick chat to identify the gap and see our options before dismissing the review."

Change It Up

On pair programming projects where two devs are behind every pull request, I find it's really common (usually among the junior devs) for one dev in the pair to submit the pull request and the other to approve it. It's not so much that this is cheating the system, but it's circumventing the whole point of a code review, which is to get a fresh set of eyes on our code to see if we missed something while we were deep in the weeds writing the code.

Devs commonly jump back and forth to specific places in a file or the project as we build up mental maps of reference points. In doing so, we can gloss over typos, incomplete code, or even blocks of code we commented out during testing and troubleshooting. Even skimming over our own pull request, we may not notice the commented-out code, but a fresh set of eyes can see the diff and ask, "Why is this code commented out? We never comment out code in our project. Should we uncomment it, or can we delete it?"

So, as a rule on my projects, I require at least three approvals on every pull request, so even if one dev of the pair approves it, the PR still needs two other devs to get in there and share accountability. Remember, it's not so much that we're asking the other devs if our code is good enough, but rather, **it's in our best interest to share some accountability with our peers.**

Imagine we introduce a bug, force-push it to the `master` branch, and break the application. The execs would have every right to be upset with us because we acted carelessly and jeopardized the sprint. If, however, our bug was missed by three other devs, then the frustration can be channeled towards the bug and the process instead of at the people. We haven't even considered QA or UAT yet, either. It's easy to point the finger at someone and accuse them of incompetence. It's a lot harder to point the finger at five or six people on multiple teams and accuse them *all* of being incompetent.

If that's the reality, then it's proof in the pudding, and we may want to

transition some folks off the project and start rebuilding the engineering team. That's pretty drastic and I've never seen it happen in person before. More likely, we address the process. Maybe we add another required approval or change the code review process to a mob review over a video conference app so the whole team can review the code together in real time. Whatever the process, just remember **every project is different and every team is different**, so don't get too caught up in your current reality and be ready to change it up at your next gig.

CHAPTER THIRTY-ONE

Architecture

This chapter will cover some general architecture concepts and pros and cons so we can make better decisions and prepare for potential bottlenecks and pitfalls.

Systems architecture is a topic worthy of its own book, and several excellent ones have already been written, so I want to keep it high level for now and talk about two significant paradigms: **architecture orientation** and **data flow coordination**.

Architecture Orientation

When discussing architecture orientation, we're referring to how all the moving pieces are laid out virtually. When we deploy the application, can we deploy one piece at a time, or do we need to deploy the whole thing, even for minor updates? Can we put the application behind a load balancer, or does it maintain its own state, and do we need to put more CPU behind it? Questions like these will help us recognize the pros and cons of each capability and setback, so we can ultimately land on a paradigm that both supports the needs of the business and considers cost and resource limitations.

MONOLITH

The monolith paradigm implies that our entire application lives in one executable. The application will typically be responsible for accepting input from a user, writing or requesting data to or from the persistence layer, processing the data, and presenting back to the user in a UI. The application can be anything from a website to a video game to a desktop application. If a single VCS repository contains all the application source code and all the bells and whistles of the application exist in one place, then it's a monolith.

This doesn't mean infrastructure dependencies have to be included here. An application that connects to an external RDBMS database can still be a monolith. It can make API calls to third-party vendor tools like Google Analytics or Mixpanel and still be a monolith. A monolith directs us to code all our application logic in one place. All the devs on our project will work on the same repository, and when we deploy our application, it'll be deployed as a whole.

It's worth mentioning that while a monolith typically uses a monorepo, not all monorepos are monoliths. In a monorepo, all project code is stored in a single repository rather than split into multiple repositories. This means that devs working on the project will typically need to have

a clone of the monorepo on their local machines to access and work with the code. Google is said to be an 80Tb monorepo*, so have fun working on that.

On the bright side, it's also possible to set up a monorepo so that devs only need to clone the specific parts of the repository they are working on rather than the entire repo. This can be done using Git submodules or a tool like Git subtree. Overall, whether or not all developers need to have a complete clone of the monorepo on their local machines will depend on how the monorepo is set up and how the development workflow is organized.

MONOLITH ADVANTAGES

Smaller projects can benefit from having all their logic and assets everything in one place. A mobile app that doesn't require online API support can be a self-contained monolith and likely won't cause much disruption or logistical pain for the engineering team. An offline Unity game that will be deployed as a versioned release can also be a monolith. A small marketing website that will never register users or scale to accommodate growth can easily be a monolith. We have plenty of options in this space. PHP frameworks like CodeIgniter or NodeJS frameworks like NextJS are convenient when building smaller self-contained applications. They can handle the back-end routes, controllers, models, and the front-end UI experience.

If a single developer can build the application, it often makes sense for the application to be a monolith. Of course, this isn't always true, but there's a realistic chance that if the product is small enough for one dev, it's small enough for one repo and one server.

The simplicity of putting all the code in one application also typically propagates up the stack to the DevOps tier. For a website, if the entire application can be deployed to a single AWS EC2 instance, it can also be deployed to a single DigitalOcean droplet or a single Heroku Dyno. The size of these hosting servers depends on the application size--simpler

* <https://www.wired.com/2015/09/google-2-billion-lines-code-and-one-place/>

infrastructure typically translates to less DevOps commitment and an overall lower cost for the project.

MONOLITH DISADVANTAGES

Code bloat is typically the first thing we'll experience. Finding something in the project or deciding where new features should go can become challenging if there's no established feature abstraction and organization pattern. It can also become challenging to delegate multiple tasks to multiple devs in a monolith because features can introduce circular dependencies, complicating testing and deployments. The risk of introducing regressions increases when one application is doing everything. A new line of code can break another line of code, and the application becomes too big for its own good.

State management can also become tricky with a monolith. For single-player games, desktop apps, and mobile apps, the state can typically live inside the monolith because there's a 1:1 ratio between user and app. A single user is playing the game, or one user is running the mobile app on their phone. There's no need to load balance because one user shouldn't be able to generate any load at all.

The moment we bring our application online and into an environment where multiple users can access our application at once, we have to address load and server strain. Can one server accommodate 100 users, all using our app simultaneously? How about 1,000 users? How about 100,000 users? If our objective is growth and profit, then at some point, the load will exceed our server's ability to "serve" the application, and users will start to experience lag.

Let's assume we've done a performance analysis and narrowed our server lag to CPU limitation. Let's also assume we've already offloaded our static binary assets to a CDN, so our server is only running our app, but it's still strained. We typically have two options at that point. We can get a bigger server with more CPU (essentially scaling vertically), or we can put a load balancer in front of our application and run multiple instances of it in a cluster (essentially scaling horizontally).

If our application maintains its own state, then we can only scale vertically because the moment we run multiple instances of our app, we introduce multiple instances of the state. Requests that go through the load balancer will be directed to the most readily available instance, and the state we experience as a user will be all over the place. We will likely get logged out, or the app will kick us back to the beginning of a UX flow if the server is somehow responsible for tracking our progress.

Let's step back and use an analogy because I think state management can be tricky to grasp until that "Aha!" moment presents itself.

Imagine we need a car. We don't own one, but our wealthy neighbor has a spare car he doesn't use and tells us we can borrow it whenever we want. What a nice guy! Every time we need the car, it's there in his driveway. We set our favorite radio stations, adjust the seat and mirror, drive the vehicle, and return it. The following day, we get in the car, and everything is exactly how we left it. This is the ideal monolith scenario.

Now imagine some neighbors also need cars and heard about our generous neighbor. He offers them the same courtesy, and now five of us all need the car at various times. We show up one morning, and the car isn't there because another neighbor is using it to run an errand, so we have to wait until it's available. This is a typical lag scenario.

Now imagine fifteen more neighbors want in on this car thing. Our wealthy neighbor decides to buy nine more identical cars, so twenty of us have access to ten of his vehicles. Now the likelihood of a car being available when we arrive at his house is much higher. Instead of just picking a car, we say hi to our wealthy neighbor, and he hands us one of the available keys and tells us which car it belongs to. This is a typical load-balancing scenario.

Now imagine we get in one of those ten cars. What are the odds it's the one we used when we set our favorite radio stations and seat adjustments? 10%, of course, and that's assuming another neighbor didn't reset our preferences. If the car only has one station and one seat setting, then we have a non-issue, but if it allows us to save some things, this becomes a significant issue. If we set our preferences on Monday

morning, we have a 90% chance that we will get in a car with another neighbor's favorite stations on the radio for the rest of the week. This is a typical state management debacle when load balancing a monolith.

So how could we solve this? Suppose the car somehow saved our preferences somewhere else, like in the cloud. In that case, when we got in the car, it could detect who we are as drivers, ask the cloud for our preferences, and automatically set the radio stations and seat adjustments. Now it wouldn't matter which of the ten cars we drove; they would all behave exactly the same. This is an ideal load-balancing scenario with stateless servers, where we need to ditch the monolith model and start entertaining a hybrid approach or fully service-oriented architecture.

SERVICE ORIENTED ARCHITECTURE (SOA)

Service-oriented architecture is a paradigm in which we split our application into multiple smaller applications that each does very specific things. The most common version of this paradigm is microservices, which sometimes can also include *macroservices*. For this lesson, let's pretend they're subjectively indistinguishable.

Let's start from our monolith, so we have a frame of reference. Imagine our application, for the most part, is very slim, but it has a couple of functions that really hog the CPU. For our example, we'll say that when users upload files, they need to be scanned for viruses before we save them, and we'll say that when new users register with our product, we need to do some very expensive provisioning.

Let's use a really niche example for fun. Imagine all our users are doctors, and when they register, we crawl PubMed and scrape all the articles written by them, including studies on which they assisted with research. We fetch the files intended for the doctors' respective profile pages and save them on disk, so we're not relying on a third-party tool for our archive.

* * *

Our imaginary "directory of doctors" product has two CPU-consuming processes (scraping and extracting), and the rest of the app is static or basic. It would make sense for us to spin up a new mini-application with a single API endpoint that accepts a doctor's name or some identifier. This mini-application can be called the `pubmed-service` and would be responsible for doing all the work of fetching and extracting the documents from PubMed. That's *all* it would do.

Our main application then does a fire-and-forget instruction to that service. It *could* wait for the service to respond, but this process could take several minutes, depending on how many documents are being pulled in. So why make the user wait? We can register the user, delegate that instruction to our `pubmed-service`, and respond to the UI with a message like "Registration successful. Fetching related articles from PubMed now. Please check your profile shortly." Suppose we wanted to enhance this user experience. In that case, we could replace that last line with "We will notify you when the process is complete" and introduce a notification feature in our product.

This exact process can be applied to file uploads. Instead of a user uploading a file into our application, we can create a new microservice called the `upload-service` or something. We could send a request to it with the name and size of the file, and it could talk to a static asset storage cloud service like Amazon S3. Our `upload-service` can request a pre-signed URL from AWS and return that URL to our main application. Our main application can then upload the file directly to S3 instead of piping the binary through our app. From there, we can configure CloudWatch events and hook them into the S3 bucket, so we tell a Lambda to scan it for viruses when a file is uploaded. Once the scan is complete, the lambda can write a record to our database or dispatch a message into our event queue so our system knows where we are with the file.

I used this example earlier when referencing a Media Platform microservice. This is how we carry **utility logic** from project to project because established patterns are just that. We want the **application logic** to contain all the intellectual property so it can stay with the client. Our client is *using* HTTP, *using* Amazon S3, *using* a database--if these tools

weren't invented by our client or created by us *for* the client, then they're fair game. Our contract may require that all the code we author under time and materials stays with the client, but that doesn't mean we can't recall how we solved the problem and rewrite the utility pattern in the future.

This is why open-source software is so powerful; who wants to rewrite the same stuff repeatedly to solve the same problems? I've authored several open-source utility libraries and microservices that I can leverage on future projects, and I recommend you do the same. The trick, of course, is to author these on our own time. We can't bill clients to write code for them and then open-source it--that's just stealing time. We can, however, spend our weekend authoring a handy open-source library that happens to solve a utility challenge in our commissioned project. On Monday morning, our billable work becomes pulling in our published library vs. pulling in someone else's published library or coding it by hand inside the project. We now have more control over the project even though the dependency abstraction hasn't changed, and we're now more valuable for future projects.

SOA ADVANTAGES

Our main application is now *much* leaner. All the heavy lifting is abstracted into microservices that only care about the process that matters to them. Suppose we scale this paradigm so that every aspect of our business can be handled as a one-off step by a compartmentalized service. All the services need to do is receive instructions and dispatch a status message. In that case, we can have an extremely flexible, well-orchestrated infrastructure that allows us to build services out of execution order.

We can easily allow our application to behave differently in different environments because all we have to change is the orchestration of payloads. None of the code has to change. We can keep all our services extremely simple and delegate some to handle the happy-path logic flow and others to handle errors and logic breaks.

We can more easily create pipelines and sales funnels. We can A/B test flows and leverage blue/green deployment efforts to try out different

pipelines for different user pools. We can break up the engineering team and isolate work streams on a per-service initiative because the devs only need to consider the IO contracts. Each service can be tested as its own unique deliverable.

Services can be authored in different languages to suit their responsibility better. For example, since JavaScript is notoriously bad with math due to its floating-point Number type, we can build a math-heavy service in Python to address that one gap in a contained release effort. Imagine trying to maintain a monolith that executes subroutines in different languages, requiring multiple runtimes to be loaded in parallel to get the application working.

SOA DISADVANTAGES

Of course, every tool and paradigm has trade-offs. To understand the microservice's risks, we don't have to look much deeper than the monolith's benefits. Where we gain the flexibility of development logistics, we also gain overhead in management and release coordination logistics. It's much easier for project managers to consider a monolith feature as part of a monolith application. It's harder for them to understand a microservice as an entity with its own release with no UI, and features will passively leverage it asynchronously through API, message daisy chaining, or some other implementation flow.

Microservice architectures also introduce complexity in the DevOps space. We typically need to introduce stateless containers and coordinate around service discovery, cross-service authentication and authorization, and overall networking strategy. Microservices add complexity when introducing key rotation and invalidation because auth retries often happen in the middle of a request and only involve one step in a multi-step request journey.

New microservices need to be provisioned and introduced into the architecture suite, so we likely need DevOps to create scripts and templates with tools like Terraform to automate our scaling efforts. If our product needs to be cloud-agnostic, we can introduce other tools like Kubernetes, which changes the tools and strategies DevOps will

need to use to support everything.

Microservices also add a lot of overhead during local development. I have a friend who works for a company with over 200 microservices. That means over 200 repositories and potentially 200 containers running in concert on a laptop when writing code. It's different in a cloud environment where we can scale hardware as we grow. Our development workstation is one computer that can only do so much. A monolith with 200 features typically has a staggeringly smaller footprint on our computer than an SOA with 200 containers.

There has to be a way to run a subset of services based on whatever the dev is working on at the moment. Even on my smaller projects with eight to ten Docker containers, my Macbook Pro with a 2.4 GHz 8-Core Intel Core i9 CPU and 64 GB 2667 MHz DDR4 RAM starts to hate me. Keep in mind I also have browser windows open, plus IntelliJ and potentially other tools like Postman or Jmeter. If I get pinged on Slack to help on another project quickly, it becomes a hefty undertaking to tear down my local environment to spin up the other project.

This is where finding a realistic balance between microservices and macroservices comes into play. We don't necessarily need twenty microservices that all do different things for a user if we can have a `user-service` that does twenty things. It's still an abstraction, and we still get the benefit of offloading work to a background task, but ultimately we run into the theme we've been seeing throughout this book. Every project is different, and every team is different. What matters is that the team lands on a process that works for the project. These implementation details change frequently, and the money will be spent either on cloud compute costs or developer overhead costs.

Data Flow Coordination

Data flow coordination is self-explanatory. Typically, we get some input from a user or trigger, the system needs to do some things with it, and some resulting process needs to be notified. The two types of data flow concepts I want to cover are **synchronous** and **asynchronous**.

SYNCHRONOUS

Synchronous data flows are blocking data flows. A system invokes a command and waits for the result. On the UI, this could manifest as a user clicking a button, seeing a spinner, and waiting. The user is blocked until the spinner goes away. Sometimes this is appropriate and required, but sometimes this is just lazy engineering.

Most of the time, waiting for a process is a frustrating experience, especially if there's no indicator of when it will finish. A spinner only tells us that we have to wait. A progress bar gives us an idea of *how long* we will have to wait. Users are typically much more forgiving of synchronous UX when there's some indicator conveying "Please wait for 10 seconds" vs. "Please wait indefinitely."

Sometimes operating systems or video game updates will choose to do this. We can't use our OS or play our game until the update is done. Even if there's a progress bar, sometimes it tells us the update will take several minutes or hours to complete, and then we have to change our plans for the day.

If we're making web APIs, this will look like a controller accepting a request, processing the request, and returning the response when the processing is complete. Not only is this potentially creating a frustrating experience for users since we force them to wait for our APIs to do stuff, but it is also introducing a race condition because some operations may take longer to complete than our HTTP request timeout duration allows.

Stubborn synchronous websites will typically extend the timeout

allowance. Travel sites like Expedia or Travelocity sometimes spin for several minutes while trying to calculate an itinerary. That industry has somehow convinced users that patience is a virtue and that clicking anything or refreshing the page will force the experience to not start over, risking the loss of an airplane seat or hotel room availability. I tip my hat to them since most sites I work on these days need to respond to user feedback nearly instantly, or the user leaves and searches for a competitor.

ASYNCHRONOUS

Some games or applications are very considerate of our mental health and will update in the background. We get to play the game or use the app while the update is installed, and then when we're ready to take advantage of the update, we only need to reboot the software.

In the code, this can typically be leveraged through a PubSub model, where we receive the user instruction, dispatch a message for a worker or subroutine to do the work, and immediately respond to the user, letting them know "We're working on it, keep doing your thing." When the worker is finished, it dispatches its own message, and then our application can notify the user, "The work is done, in case you want to do something with it now."

In APIs, this becomes a simple separation of concerns, where our route will still hand a request payload to a controller, but the controller won't *do* the work. It'll just delegate the work and then respond to the route, saying it has been delegated. Now our APIs are all snappy and real-time responsive. This is a non-blocking data flow model.

```
const MySvc = require('/path/to/my/service')('someStrategy');
const myNonBlockingRoute = (req, res) => {
  try {
    MySvc.doSomething(req.body);
    res.status(200).json({ message : 'immediate response...' });
  } catch (err) {
    res.status(500).json({ message : 'service threw...', err });
  }
};
```

In languages like PHP, every line blocks the following line. In NodeJS, most core functions are inherently non-blocking, meaning they return a

promise instead of a response value. The promise is just a function that defers the response into one of two callback handlers, `.then` or `.catch`, respectively.

I mention this because I often see devs taking advantage of non-blocking logic flow in the code but then they don't apply it to the network, so while we may have a controller function that returns a promise, our API route is still *waiting* on that promise before it can send back the HTTP response.

```
const MySvc = require('/path/to/my/service')('someStrategy');
const myBlockingRoute = (req, res) => {
  try {
    MySvc.doSomething(req.body)
      .then(data => {
        res.status(200).json({ message : 'blocked response...' ,
data });
      })
      .catch(err => {
        console.error(err);
        res.status(500).json({ message : 'service rejected...' });
      });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message : 'service threw...', err });
  }
};
```

This is one of those patterns that's not obvious until it's obvious. I also don't think every API endpoint needs to tap into a message queue and follow a non-blocking paradigm. Suppose we have 100 API endpoints, and only two are computationally expensive. In that case, we don't necessarily need all 100 littering up our message queue with events that are handled immediately after they're added to the queue.

We may want 98 "blocking but fast" endpoints and two non-blocking endpoints. If introducing multiple patterns is ok with the team, I'd recommend trying it out for extreme cases like this. Suppose the team is at risk of getting confused by a hybrid paradigm. In that case, I'd say load up the queue and stay consistent so new, and junior devs have clear and consistent patterns to follow and copy when adding new endpoints.

Summary

At the end of the day, learn from existing working models that have been proven by other companies. Consider whether or not those models apply to the project. Consider whether or not the team is ready to adopt the model, and take steps in the appropriate direction to help automate and simplify as much of the process as possible so the team can leverage the process to build the product. If at any point it's determined that the process is in the way and hurting velocity, adjust as necessary and re-approach the situation with the newly considered information.

There's no magic bullet for this. Just keep as many options in mind as possible and recognize where you are in the chaos of systems architecture.

CHAPTER THIRTY-TWO

Technical Debt

Developing software is an ongoing song and dance. It's a constant balance between **product goals vs. engineering reality** and **engineering goals vs. product reality**. The product teams work with execs to bring corporate visions to life. It could be an innovation initiative, or maybe it's a rebranding effort. Most of the time, these goals have deadlines that are established without any clear understanding of anything in the engineering space, so by the time engineering is brought into the conversation, the foundational requirements may already be laid down.

I've worked on a project where the client had plans to roll out technologically impossible features to deliver because mobile phones didn't support the capabilities they wanted. Trying to communicate to execs is frustrating that if they want their vision realized, we need to consider establishing a partnership with smartphone manufacturers like Apple and Samsung and firmware vendors like HTC and LG. The execs are like, "Can't we just build it?" which is painfully frustrating because they're so detached from reality and, ironically, quite flattering since they see us as people who can solve all the problems for them.

Often in these types of situations, we need to find a compromise. We need to build something that can realize *some* of that vision, or we may need to adjust how we build something to get a feature in front of customers immediately. Then, based on customer feedback, we can decide if it's something we want to maintain or throw away.

Diverging from established processes, patterns, and paradigms to deliver something quickly is called technical debt. It makes the product team happy, and it makes the engineering team sad. Consequently, when it comes time to refactor the ugly code to align it with our conventions, the product team can see that as "the devs are rebuilding something that already works," which makes the product team sad and the engineering team happy.

So how can we solve this in a way where we keep the sadness to a minimum and the happiness to a maximum? Unfortunately, like many other scenarios in this industry, no magic bullet will yield the same success rate on every team and project. There are, however, some initial approaches and high-level concepts we can consider that work well most of the time, so it doesn't hurt to start there and either run with them or adjust as necessary.

In the "Communication Mastery" chapter, I will cover a brief scenario where a PM doesn't want to pull technical debt tickets into the sprint (we have to work with them to help them better understand why this is an important step in the development lifecycle.) I want to get a head start with that here because it's the foundational element of how we align on things. We need the product team and PMs to see the same value that we see in paying off technical debt. We need to communicate that in a way that conveys a couple of things:

- We as engineers need some sanity and consistency in our projects, so our brains don't have to work so hard all day, keeping track of loose ends and pattern divergences.
- Consistent and predictable code is reliable code. We will be able to deliver their features with fewer bugs and be able to reuse portions of our application to get new features and capabilities for free.

For example, if the feature is a new table for a new type of data set in the application, the product team may ask for subtle capabilities, like "We need to be able to sort by x, y, or z, and we need pagination." If we

can say, "All of our tables do that because we built those features into the generic table component," that's a massive win for them. A 5-point ticket becomes a 2-point ticket, and we can add more work to the sprint.

That example may seem silly and insignificant, but it still amazes me how often I'll see devs cut corners on things like that. The next thing we know, three tables in the application all behave differently, and each has its own logic handlers simply because the PMs rushed the devs to get each table out the door as quickly as possible. If we know that our app will have multiple tables (and for whatever reason, we decide we're not going to use a component library and roll our own), then the first table that's tasked to be built is at risk of having all the common logic built into it. Come time to add the next table; if the next table can't share any of the features because they live inside the first table, then we have a speed bump.

So, imagine we tell the PM, "Instead of building this table here inside this first page, I want to build a generic table component that can be pulled into any page that needs a table. It'll take a bit longer, but it'll make future tables much faster to implement, and they'll be consistent." If the PM says, "No, we don't have time," then we need to communicate to them immediately the repercussions of their decision.

We can tell them, "That's fine, but we don't want to repeat this one-off process. We'll need to build the generic table component before working on the next table, and the ticket will take even longer because we'll have to refactor the previous table so it can leverage the generic component. In other words, we can get this first table out the door faster, but it will push back delivery of the next table."

Now, it's on the PM to make a decision. We're not laying down the law; we're simply giving them some sprint planning options so they can plan the sprint better. That's fundamentally their job on the project. Suppose they still say, "No, the second table also needs to be a one-off; just copy/paste the first table and make in-line edits where needed." In that case, we need to communicate that this is only making the debt worse, as the third table will require refactoring the first *two* when we introduce the

generic table component.

Ultimately, the PM is providing implementation details to the engineer, and we need to have an established separation of authority between product and engineering. We, as engineers have a stake in things just like the PMs do. We want a portfolio we can be proud of for future employment opportunities just like theirs. If the PM is setting us up for failure so they can get a quick win on a shortsighted promise, then we need to intervene a bit. We do that by insisting that we include some tech debt tickets every sprint **so we can pay it off quickly, responsibly, and consistently**.

It's a bit like taking out a loan to buy something. Paying off the debt means we pay off the loan. If we ignore the loan and take out new loans to buy new things, we're getting deeper and deeper into debt, and eventually, our finances will become unmanageable. This reality unquestionably presents itself in our code. The more one-off features we build that don't support reuse or follow conventions, the harder it becomes to build new features because new code breaks existing code.

Of course, as an aside, if we take out another loan to pay off the first loan, then we don't catch up on our debt at all; we just move it around. That manifests in our code as replacing one shim with another shim. If the new loan has a better interest rate, it might be a good idea to go that route. Similarly, if we replace a terrible shim with a better shim that's easier to swap out, it's not our ideal scenario, but it's in the right direction.

Accruing a significant amount of technical debt also makes it harder for us to train new devs because they might be exposed to twenty different ways of solving the same problem. If they have no guidance or frame of reference when adding new code, they'll be relying on more veteran devs to provide that reference while they're working, which can be distracting and time-consuming.

The typical result is team velocity grinding to a halt and developer morale dropping to the point that several individuals don't want to work on the project anymore. The aftermath is our team having to focus

on accommodating multiple staff transitions instead of focusing on building product features on a codebase that enables rapid development.

By now, we understand the importance of paying off technical debt. PMs who've experienced the catastrophe I just mentioned will likely be more inclined to support tech debt payoff efforts in every sprint too. Some PMs are incredible to work with and are legit ambassadors for the engineering department when they communicate with the product team. That ambassadorship usually stems from the PM having previously been a dev and has since transitioned into the new role, or else the PM has been on a project and seen firsthand what unmanaged technical debt can do to the project's velocity.

I typically recommend paying it off as quickly as possible, like the following sprint. Some devs love doing that kind of work, so it might be a simple matter of asking the engineering team who's up for it and dedicating one or two devs to deliver the tech debt tickets. Tracking technical debt is just a matter of tagging the tickets appropriately in the issue tracking system and subsequently tracking the debt *dependencies* so the debt can be paid off in order if needed.

One thing to consider that's entirely within our control is that we can be courteous and forward-thinking if we need to implement a temporary solution that we know will be refactored through a technical debt ticket. We can code it in a way that helps make the tech debt refactor easier. Maybe we need an abstraction layer but don't have time to abstract it to a remote service. We can add a seemingly useless abstraction right in the code above the thing we're building.

Then, when we want to pay off the debt, the abstraction part is already taken care of, so we only need to move some things around instead of unwinding some stuff to apply the abstraction as a concept after the fact. Even something as simple as documenting a plan of action for the next steps and putting the thought effort into the debt ticket can go a long way and prevent the need for future grooming work if another dev happens to receive that ticket.

* * *

If we can accept that technical debt is just part of the process and that getting customer engagement sooner through our temporary code can earn extra money, then we can start to see how the technical debt can pay for itself and then some. Now we're getting a loan to invest, and the money we earn on the investment can pay off the loan and continue earning for the long term.

Ultimately, technical debt as a concept rarely causes friction across teams. The underlying issue is that both the product and engineering teams are often on the hook for specific deliverables. When one team demands a process that prevents the other team from delivering at a velocity representing their character, that team can feel like their character is being compromised. If we can find the right balance between give and take and agree that we all want the product to succeed, then **it essentially just comes down to communication, the number one most valuable skill on our toolbelt.**

Part Three
Collaboration

CHAPTER THIRTY-THREE

About The Collaboration Section

Let's face it; software engineering isn't just us with our code anymore once we join a company. When we first started programming, it was a brand new thing, right? We were introduced to a new world of possibilities with endless ways to express ourselves creatively and challenge ourselves intellectually.

As our skills increase, we become more desirable to the workforce, and the projects become more prominent as the clients get bigger. With those bigger budgets and projects also come bigger teams; initially, we can become peas in a pod. We're not just gunslingers in the wild west anymore. Now we're expected to cooperate and behave, stay in the lines, don't break things, and deliver deliver deliver.

We have to take direction from engineering leads regardless of whether we agree with them. Maybe we learn something new, or maybe we learn how *not* to do something. Sooner or later, we learn enough that we begin to anticipate things and can start to offer as much value as our superiors, but what if there's no obvious opportunity for growth? Were those leads once junior devs on our team, or were they hired from another company? Is this a conversation we can even have with them?

How do we break out of this rat race? How can we move up in the team hierarchy, so we're giving more direction than we're receiving? So we're using **our** creative and intellectual minds to spark ideas and then

delegate those ideas downstream to the engineering workforce instead of simply executing orders from above?

Well, our success in this field depends much less on our technical capabilities than you might think. Consider leveraging the 80/20 rule here, where 20% of our career potential comes from our ability to engineer software well and 80% from collaborating and leveraging relationships.

This section will focus on that, so buckle up and let's get started!

CHAPTER THIRTY-FOUR

Communication Mastery

In this chapter, we'll discuss some highly valuable communication practices to help you collaborate with your teams.

Know Your Objective

First and foremost, when communicating with anybody, have an objective in mind. Are we trying to convince someone to see our point of view or change their mind? Are we trying to build rapport with someone to secure a closer working relationship? Are we trying to give direction or coaching guidance to a peer?

Different scenarios present different ideal outcomes, and the way to achieve our goal will also be different. Having a clear objective in our mind at all times gives us measurable direction for our conversation. It helps prevent endless rambling or non-constructive arguing. Sometimes we get caught up in the moment and try to **be right**, but most of the time, what we ultimately want is to **get it right**. Different people respond differently to different types of communication. In the same way delivering an agile project is not going to be a straight line, neither is the path to getting aligned with another person on a particular issue.

Know Your Audience

With that objective in mind, the next thing we need to do is know our audience. In our work, we must consciously translate our words for different teams and individuals on our projects. For example, in a meeting I had this afternoon, I needed to explain to the client how we could solve some cumbersome cross-team logistical issues. Our team could focus on building the mock local strategies for some service APIs on infrastructure scaffolds, enabling the *client* engineering team to build production-tier strategies that talk to their remote partner APIs.

Unfortunately, when any of the devs mention anything about architecture or code, the client smiles and says, "That all sounds great; I don't know what any of that means, though." So, when I spoke up, I paraphrased my point for him by saying, "I recommend just assigning the tickets to our team first so we can build the pre-production stuff, and then we'll re-assign the tickets to your devs to take over for production stuff. We can leave obvious breadcrumbs for them to fill in the gaps to save time and guesswork." He said, "Perfect! Let's do it!"

Speaking with fellow engineers, we can get very low-level with nerdy terminology because it's part of our world. We can visualize code and infrastructure during the verbal exchange. In contrast, an executive, product manager, or designer won't necessarily be able to do that, so all they can do is tune us out or get frustrated with us and risk looking like a novice. We can inadvertently make them seem ignorant, or we can come off as condescending or even uselessly nerdy. If we can't use words that they understand, why would they ever want to promote us and work more closely with us?

Learn To Model

One approach that works very well is called modeling. In the spirit of knowing our audience, we can also casually mimic them. If the person we're speaking with is standing with their arms crossed, we should stand with ours crossed. If they take a sip of coffee, we do the same. If they take long pauses instead of saying "Ummm" when they speak, we should try to speak the same way.

We don't want to imitate them to their face rudely; instead, we want to try to "fit in" so we seem more like them. **People like people like themselves.** This is a known fact in human beings. To better understand this, let's imagine someone whose personality is opposite ours. If you're a quiet and reserved person, imagine having a conversation with someone like Sam Kinison, who's shouting every word at you at the top of their lungs. How long would you want to sit there and listen to that?

Now imagine you're amped up and motivated about something, super excited and happy, and trying to have a conversation with a slow old person from another generation. How long before they suck the energy out of you?

Have you ever seen a clip of a celebrity who stepped out of their element to interact with a young fan, and it resonated with you as the right thing to do? There are videos online of loud, tough wrestlers breaking character to engage with kids or special needs fans. One example that sticks out is when the Undertaker meets a young female fan for a charity event*. In the video, he brings his energy way down, speaks to the young girl with a quiet and calm voice, and presents himself as completely non-threatening and genuine. He matched her energy, her volume, and her demeanor. He even sat down to be closer to her level, so he wasn't towering over her. He modeled after her to gain trust.

* * *

* <https://www.youtube.com/watch?v=U91MEWG2zHQ>



Had he stayed in character, talked to her like a fellow wrestler, or even just as an adult, it could have created an awkward, intimidating, or uncomfortable situation for her. She may not have wanted to talk or even make eye contact with him, especially because she was a fan, so the relationship was already very one-sided. Modeling helps build subconscious rapport with the person we're speaking with and can help them open up and be more agreeable and engaging.

Agree Before Disagreeing

That brings us to our next practice: **agree before disagreeing**. If we happen to be at odds with someone and need to pull them onto our side of a debate or conflict, the first thing we want to do is find common ground. Find something we can agree on and go from there. For example, imagine we need to convince our PM to allocate more bandwidth to technical debt, like during sprint planning if we keep trying to pull in tech debt tickets, but the PM wants more features developed and says the tech debt can wait. You and I both know this is a road to disaster.

How might we get the PM to agree to start our negotiation efforts? We want "yes" answers, so we could start with something as simple as "Can we agree that bug tickets are a pain and make us look bad?" Of course, bug tickets are a pain. They're embarrassing and stupid; our customers shouldn't be our QA. So far, so good.

A follow-up might be, "Can we agree that our aim is to deliver features with as few bugs as possible?" Again, of course. We're intentionally asking self-evident, no-brainer questions to get them to agree with us. We both want the same thing; we're just misaligned on the approach.

One more follow-up might be, "Can we agree that feature tickets that get kicked back by QA for introducing regressions and breaking integration tests cause a lot of stress during the releases?" Again, another yes. So we're batting 1000, and they agree with everything we've said.

Learn Socratic JiuJitsu

Once we've framed the conversation and gotten the PM to recognize the problem and agree that it should be addressed, we want to pivot from yes/no questions and ask a question that puts them in a position to collaborate. Very often, if they don't have a solution, they'll adopt ours without much resistance. And our goal is to collaborate and solve the problem, so if they offer a better idea, then great! We both win, and we learn and grow. Our rapport will strengthen, and mutual respect will evolve.

For example, we might say, "Given that we both want a stable, bug-free codebase that speeds up releases and reduces stress, let's ensure that our codebase supports that every sprint. The more features we build on placeholder code, the higher the risk that they won't work as expected; the harder it'll be to refactor later, so we'll lose more time. I recommend we never have more than 100 story points worth of tech debt in the backlog. How much would you feel comfortable allocating each sprint so we can catch up?"

Notice this isn't a yes or no question. We're not giving the PM an option to back out or dodge this because we're putting the PM's competency on the line. We also want to try not to explain the *why* behind our technical debt request--the *why* is important to the engineering leadership and us but not to the PM. We only want to understand the PM's *why* for their resistance so we can appropriately counter and adjust the dialogue to address their concerns. If the PM refuses to budge, that says more about them than it does the issue, and we may need to escalate above the PM to get a green light; this is when we would outline the *why*. If the PM is simply rejecting tech debt because they feel pressure from above, we can infer that the debate here isn't about the value of addressing technical debt. More likely, the PM's concern stems from job performance and security. This is a genuine human concern; moving forward, we can use this to our advantage. This is called finding the underlying motivation.

Discover The Underlying Motivation

Everybody has a reason for doing or not doing everything in life. If we want to understand these reasons genuinely, we have to be willing to build rapport and peel back the onion layers to find the most authentic, honest, and compelling reason. Then we can convey our goals to help these people get what they want.

For most of my professional life, I've split my career between being a software professional and fitness professional. It's a nice balance between being active and being creative. As much as I enjoyed training clients and leading group fitness classes, I never enjoyed prospecting or selling personal training. It always felt invasive to walk up to someone working out on their own and offer to correct their form or start doing a sales pitch. So while I excelled as a trainer, my sales numbers were never close to some of the top-grossing trainers in the clubs where I worked because they were willing to do the hard sales style of prospecting.

In 2001 I was a personal training director at Bally Total Fitness when it acquired Crunch Fitness. Part of that merger was that our personal training departments were to begin modeling after Crunch's. One of the most significant changes was how we approached sales and intro sessions. Intro sessions are like the "Free personal training session" new members get when they sign up for the gym. Previously, we'd spend an hour showing the new member the club and walking them down the row of selectorized machines, ensuring they knew how to use proper form and which machines targeted which muscle group, how to count and log reps, and so on.

At the end of the session, we'd have inadvertently taught them *not* to need us, which wasn't very lucrative in the long term. The new approach was a fifteen-minute one-on-one high-intensity session. At the end of the workout, they were exhausted and had no idea how to reproduce the experience without us, so we created that dependency for them.

* * *

Our prospecting also changed such that we would still walk around and correct exercise form now and then, but after we'd help or complement them, we'd walk away without a sales pitch. We would repeatedly demonstrate value and plant the seed that they can't do it without us. At this stage, we still didn't know what "it" was, and that's the main point--how can we possibly sell personal training if we don't understand why the member wants it? Sometimes we'd have built some initial rapport with a newer member and say something like, "Hey, come work out with me for fifteen minutes again!" There's a good chance they hadn't reproduced the intensity or effects of our intro session on their own, so they would happily accept.

During this follow-up workout, we begin to peel back layers **as a dialogue, not a sales pitch**. It was a genuine curiosity. An example would sound something like this:

- | **Us:** "So tell me, what made you decide to join the gym and start working out?"
- | **Them:** "Oh, I've just put on some weight and decided it's time to make a change."

Layer one, peeled. Nothing yet.

- | **Us:** "That's great. What sort of change are you looking for?"
- | **Them:** "Just trying to lose that weight and get back to where I was twenty years ago."

Layer two, peeled. We're getting warmer.

- | **Us:** "It's certainly tougher to stay lean as we get older. You're definitely on the right track to being consistent at the club. Tell me about your life twenty years ago."
- | **Them:** "I was in college on the wrestling team. Life was great back then, but I suffered a bad knee strain during a match and had to sit out for the rest of the season. I couldn't train on the mats for months, but my appetite didn't know that, so I'd eat as I always did. I started gaining some weight and losing motivation and then just ended up focusing on career and

family."

Layer three, peeled. We're getting hot now.

- | **Us:** "Injuries suck. Wrestling certainly abuses the body if you're training hard. Are you thinking of getting back into wrestling? Do you miss it?"
- | **Them:** "Absolutely! I wouldn't compete as a wrestler in my 40s, but I'd love to be able to coach my kid's wrestling team. Unfortunately, I can't move like I used to. I need to lose this weight first."

Layer four. Here we go.

- | **Us:** "That's a fantastic idea! When does your kid's wrestling season start?"
- | **Them:** "Well, the current season is already well underway, but their coach is retiring at the end of the year, and I'd love to have my ticket in for next season as his replacement, so six months."

Layer five. Now we have an underlying motivation **and** a timeline.

- | **Us:** "Ok, I have some good and bad news. The good news is that six months is plenty of time to lose excess fat and regain significant mobility. Bad news, it's not going to be easy, and you're not going to get there using these machines. Think back to your college training. Sitting on the chest press machine and doing 8-12 reps won't cut it for you. It would be best if you had mobility drills and a functional training regimen to rebuild your timing, balance, tendon strength, and range of motion. Let's do this. I have this time slot open every weekday and Saturday morning open. I want to book four days a week with you and hold your feet to the fire on this so that come next wrestling season, your son and his team will be looking up to you as an example of prime athletics, and you'll be able to keep up with them and put them in their place on the mats during practices."
- | **Them:** "Hell yea, man, let's do it."

So let's unpack what just happened. When we started the conversation, it was vague and typical "I want to lose some weight." Most people would assume it's just about looking better or feeling better. Peeling back and understanding *why* the prospect wants to lose weight, we inevitably learn that the prospect wants to feel significant again and make an impression on his son.

It's an entirely different incentive at that point, and so long as we regularly remind the client what he told us he profoundly needs, it re-frames everything. **He's no longer paying us for personal training. He's paying us for that bond with his son.** He's paying us to feel how he felt in college. Deciding factors like price, scheduling, etc., are **much** less concerning for the client. This isn't sales theory; this is a proven model. Learning to talk to prospects and tap into that underlying motivation completely changed my fitness career. Naturally, it didn't take long before I began applying it in my software career.

For us, this typically manifests as an idea or direction from someone, and we disagree with it. We don't see its value, so we get the urge to push back. Often, all we need to do is inquire about the idea and find its underlying motivation, which opens up multiple avenues for variability in the implementation approach. Coincidentally, I would tell the other person the same thing to get through pushback from their idea. This is the sales dance--a sale is made to proceed in one direction or another, depending on who can lead the alignment effort better.

For example, maybe we have a new client, and while planning the initial stack, the client chimes in, "I'd like to use Ruby for the code." That's a niche requirement. Don't just say, "Sure, boss, no problem"--ask why. Why Ruby? Ruby isn't exceptional and is on the tail-end of popular languages these days. If it's a relatively small project, it doesn't matter what language we use, but if it needs to scale and be maintained for several years, an older language like Ruby will likely hinder the project. Regardless of what Ruby can or can't accomplish, it may be harder to find Ruby devs than devs offering expertise in newer tools.

Maybe the client says, "We just hired our CTO, and he recommended Ruby"... again, why? It's an obvious guess but let's play this out. If we

can imagine peeling back layers of the onion similarly to my personal training example, maybe we learn the CTO is an old Ruby dev and misses being involved in the implementation. Maybe they regret the career move up to management and miss writing code, and the only code they know is Ruby.

Knowing our approach to winning this sale, we must also be mindful of the other party's efforts in doing the same toward us. For example, if I worked for the company hiring our agency, I would advise the CTO to argue that the programming language is an implementation detail and that if the agency is competent, it's a non-issue because we could containerize our logic and swap out microservices with other languages later. We should decide with Ruby and move on. For the sake of our chapter, however, let's play out our side of the sale.

From there, we can pivot the offering a bit, saying, "I absolutely understand that position! We all code because we love it, and I think if we spend too much time in management positions, we can yearn for it because it's our roots. This project needs something newer and more applicable than Ruby, but we want to involve you through the product lifecycle. We recommend either Go or Node for the microservices, and we can teach you all the basics as we build the infrastructure so you can ramp up on the syntax and be in a great place to lead maintenance efforts when it's time for us to transition off."

By peeling back the layers to find the underlying motivation, we learned that the decision to use Ruby has nothing to do with Ruby's capabilities. It is about the CTO wanting to be involved and feeling Ruby would allow that. Re-framing the solution to propose that the process can accommodate any tool on the stack, we make ourselves look like an outstanding partner and put the project's needs first.

If we bring this full circle and apply what we just learned about finding the underlying motivation for our PM who avoids technical debt, we may learn some interesting things. The PM may have missed deadlines in the past and been scolded. Maybe the PM made some promises to an exec and is worried they will lose their job if we don't finish all the features.

* * *

Solutions to these issues may not involve technical debt. The PM may not be getting the backing they need from engineering in director meetings, or the PM struggles with understanding and communicating engineering needs to the execs. You can see that pressing about the importance of technical debt can keep the situation argumentative instead of collaborative if we're not addressing the PM's real needs or concerns.

We need to put ourselves in a position to communicate to the PM that we want them to succeed. The path to making that happen is ensuring our features work, that they can scale, and are bug-free. For that to happen, we need to do regular maintenance work on the code to keep things running. Framing the conversation as "For your ticket to work, our ticket needs to be completed" instead of "We need to do our ticket instead of your ticket" will communicate that we're not trying to steal the victory from the PM. We're trying to enable them to shine in their position truly.

Address Process Not People

Notice we're not addressing this PM's choices or questioning their competence or anything. We're addressing the **process, not the person**. Whenever we're trying to improve something, always try to zero in on the process. We can all work together to improve the process, but implying the problem lies with a person also implies they somehow need to change, which almost always results in pushback or ruined relationships.

For example, if one dev on the team is constantly authoring buggy code, it can be tempting to blame that dev. Still, we should be blaming the process for allowing bugs from *any* dev to get into the production release. Think of all the steps that could prevent that bug:

- Linter
- Static Analysis
- Unit tests
- Code review
- Peer review
- Integration tests
- E2E tests
- QA review

If a bug makes it past that many gates, we should look at the integrity and robustness of our quality gates.

- Is the linter as good as it can be?
- Do we require enough devs to review the code before it's eligible to merge?
- Is that a manual "honor system" or an automated process that tracks who signed off on what?
- Do our tests cover the code in question?

- Are the tests being reviewed as well?
- After a code review, do peers pull down the code and try to run it, or are they doing blind approvals?
- Is QA set up to verify things and block releases, or are we always playing catch-up?

Having a bullet-proof process for allowing only the highest quality working code into the product would eliminate customer-facing bug tickets, and our problem dev would never make it past the first couple of gates. **The challenge then wouldn't be addressing their bugs but rather their velocity.**

Okay, now this dev is taking much longer to do work because the automated tools and processes keep kicking their code back and preventing them from delivering. Let's identify the gap. Where are they failing to align with our standards, patterns, and so on? Are they making mistakes and doing things wrong? Or are they just doing things differently, where their code could work, but it doesn't match our conventions?

Pair programming can solve most of this. Pair the dev with one of our top 20% output devs (which might be us) and watch the knowledge and behavior naturally propagate from the senior dev to the junior dev. It's easy to write off an underperforming dev. Try to communicate with the dev and work with them to improve. Let's focus on the process and ensure it enforces quality and enables the devs to do their best work. In that case, they can focus less on everyday technical nonsense and more on solving complex problems for business requirements.

If the bugs are logical, then we might adjust the process so that QA is brought in earlier and can work with the product owner to ensure that what's being delivered aligns with the acceptance criteria for the product. There's usually an opportunity to revisit processes. The sooner we accept that our teams will always have individuals, the sooner we can tailor the process to our unique team to let everybody find their place where they offer the most value.

Speak Last

In situations where multiple people offer up ideas or opinions for something--like maybe the team is ready to start improving processes where we've identified gaps--we want to do our best to listen first and speak last. If we speak first, we put ourselves in a position of having to defend our idea or witness it get passed over as new ideas are presented. Additionally, there's always a risk that someone else's idea considers something we hadn't thought of.

Speaking last ensures we've heard all the information and all the proposed options and can consider as much as possible when proposing our own solution. We can look like a team player if we quietly abandon our idea and agree with a better one that's proposed. We look like a leader if we offer our idea last and point out that it addresses an item that the other ideas had missed.

The pros of speaking last vastly outweigh the cons, and I highly recommend trying it out, especially if you're the type of person who always has to get your opinion out there and heard by folks. Remember, other people on the team also want to feel special, and often the higher on the ladder we go, the less we **do** and the more we **decide**, so consider it practice for your future leadership position; if someone has a good idea, even if it's different from your own, pick theirs and watch their expression. You'll likely gain rapport with that person, and they'll remember you taking their side.

Reflect

Whenever we debate with someone, we want to let the other person know they are being heard. People often care more about being heard than they do about people agreeing with them. One way we can do this is to use **reflection**.

Imagine one of the devs says, "I don't think we need three approvals on pull requests. It's too time-consuming for us when we're working on other stuff to be pulled onto so many code reviews." When it's our turn to speak, we might say, "I heard you say you think three pull request approvals are too many and that you're concerned about being distracted during your work day. Before I respond, did I hear that correctly?"

That brief reflection can lower that person's guard and change their tone so that it's not us versus them but us working *with* them to solve a problem. We're hearing their concern and validating them instead of dismissing them. Our proposal or decision may not align with or satisfy their concern, but we can address it now that we've heard the details.

Imagine we could respond with, "I believe we changed the process from two to three approvals on pull requests last quarter. Have we noticed fewer bugs in production? I'd like to see a metric on that, as well as how many pull requests were rejected by the third reviewer over the last three months where the first two reviewers missed something. That'll help us better understand if we need three reviewers or if it's safe to go back to two. If we need to keep three reviewers, how would we all feel about allocating time slots for code review availability? Maybe one-to-two hours per day, you're available for reviews, and the rest of the day, you're off limits. Would that help you manage distractions better?"

That type of response puts the dev's needs as the **primary** concern and the code quality as the **secondary** concern. Reflection allowed us to convey to the dev that we hear them and that whatever we're proposing will not leave them hanging.

Empathize

The trick here is to apply empathy with people on our team. Genuine empathy isn't being able to walk a mile in someone's shoes. Genuine empathy is standing with them and **believing their experience is true and honest, even if we haven't lived it ourselves.**

So, imagine if we've never personally felt distracted from doing code reviews, it might be hard for us to imagine how or why the dev would feel distracted. It might be tempting to judge this person as having poor time management, being disorganized, or simply being irrational.

Being empathetic is hearing them, reflecting on their concern, believing them, and letting them know that we're here to help them through the situation. Suppose we have the authority to implement a change for their benefit; great. If we don't, that's ok. We can still build rapport with them by aligning with them and helping represent them when possible to the key stakeholders.

If we genuinely don't understand where they're coming from and we're struggling to believe them, we can open an honest dialog and see if they'll share some insight to help us better understand their situation. We have to do this from a place of genuine concern and curiosity, not from a place of authority where we imply that we won't make the change until we understand their pain. That's not what good leaders do. Good leaders believe the people on their team. We work with them to assure them that they are heard and appreciated and that while not every solution can accommodate their unique situation, we can still consider their needs in every decision. Hence, they feel appreciated and represented, and when that happens, they'll be much more likely to pay it forward and pick their battles with the rest of the team.

All the communication principles I outlined take practice and energy. They aren't easy to apply because emotion is often in the way and skews our judgment. When we feel frustrated, offended, or burned out, we can feel like we don't have the time or energy to deal with difficult people.

* * *

I recommend applying the same iterative approach we take with our code or with architecting systems. Not every tool plays nicely, and not every API is easy to work with. It's ultimately up to us to work around these challenges, just like it's up to us to work differently with different individuals on our teams. Just like anything, the more we practice, the better we get.

CHAPTER THIRTY-FIVE

Pair Programming

Pair programming is one of those practices where most devs either love it or hate it. I remember for years hating it. I just wanted to focus and build software. I didn't need help, and it didn't feel the same as teaching another dev or learning something from another dev. It felt clumsy and condescending as if the management didn't trust that we could build the feature ourselves.

Around 2014 I took a position as Principal Engineer on a team at a company I won't mention by name. It had been around for a long time, but its primary product was a desktop application. Shortly before I joined the company, it acquired another company that had built a mediocre and bug-littered web application.

My employer formed a new team to essentially build a better, cleaner version of their online tool to replace that one. The CTO's philosophy was, "We'll hire a collection of junior devs fresh out of a coding boot camp and then hire two veteran senior engineers to show them the ropes and keep things in order." My position was to take a group of ten hungry amateur coders and channel their ambition and motivation into work streams that wouldn't escape our product roadmap and technical standards adherence.

The other senior engineer was a veteran software professional. We ultimately divided our duties up such that he would oversee logistics

and processes, and I would oversee technical consistency and quality. His first rule was we become a full-time pairing shop. Devs paired up in twos; if we had an odd number of devs on any particular day, I would be that dev's pair.

I didn't initially see the value in forcing the devs to pair up eight hours a day every day, but that was his call, so I supported him. Immediately I noticed a difference in his approach. We had five pairing stations, each consisting of a single Mac Mini, two displays, and two keyboards/touchpads. Both screens mirrored the same image, and both keyboards could work simultaneously. First of all, this was a brilliant setup. It wasn't one dev looking over the shoulder of another dev. It wasn't one dev coding while the other sat on their phone bored. Both devs were engaged.

Another adjustment was that a story ticket lived with the pairing station, *not* with a developer. The general rule was no dev is coding for more than fifteen minutes straight without handing off keyboard control to the pair. Often we would trade off one function at a time. Every day, pairs rotated. One day the right side of the pairs would rotate clockwise to the next pairing station. The next day the left side of the pairs would rotate counterclockwise to the previous pairing station.

The result of this system was that any dev would never spend more than two days on any ticket, and they were constantly starting their day in the middle of unfinished work on an unfamiliar ticket and had to ramp up on it. My job was to enforce coding standards, consistency, patterns, etc.

So after a few weeks of getting comfortable with our process, we could hop on any pairing station, and the code would look like "our team's code." There was no individuality in the code. It was so consistent and predictable that as devs would transition to new tickets, the only thing they needed to ramp up on was business requirements for the day.

On the other side of the office was the acquired company's staff. They had an existing engineering department maintaining their existing product, and our team of ten junior devs ran circles around their team.

We were extraordinarily agile and could pivot and fork efforts because that's what we practiced daily.

You could ask any dev on our team about any feature in our codebase, and we all had the same understanding of all the moving parts because we all would work on all the code all the time. No one dev owned any one feature. If any dev left, that insider knowledge didn't leave with that dev. Our engineering team was essentially a RAID setup, so when a dev transitioned off, and we brought in a replacement, there was virtually no onboarding required. We just had them pair up as usual, and the knowledge naturally transferred, the patterns were naturally consistent, and the new dev had no issue integrating with the team.

Because all the devs worked on all the code, there was a shared sense of ownership and pride in our project. We built it together, and all wanted it to work and shine above anything the acquired company could build. The acquired company was spending all its time fixing bugs, and we were spending all our time building features. We also lived by functional programming and test-driven development, so the modularity and robustness of our code were top-tier.

Ironically, our CTO was fired, and the head of the acquired company's engineering department was promoted and took over our team. She did not understand anything we were doing. Her team was understandably bitter that we outperformed them substantially, so she held a meeting with us and said, "No more pairing, no more unit tests, no more functional programming." She wanted us to start working like them and refused to work like us even though we had objective results that proved our process was successful.

So we all quit--all twelve of us. Ten junior devs and two engineering leads all left in the same week. You don't get that kind of bond casually working solo on tickets. You don't get that kind of knowledge transfer and deep understanding of the process and what agile can do until you fully invest in pair programming.

Pair programming has a couple of variations I want to mention because I approach them differently and reserve them for different scenarios:

remote pairing and mob programming.

Remote pairing is pairing virtually, like you and your pair are both working from home and collaborating online. It doesn't work *nearly* as well as an in-person pairing, so I limit remote sessions to two hours tops these days. The online real-time code collaboration tools are clunky. They don't offer the synergy that coding right next to your pair does. It's not always possible or realistic to commute to the office, depending on your situation, but you should know what you're getting into ahead of time. If you're considering an employer that requires pairing and you plan to work remotely, you're adding a layer of difficulty and stress that you may not have if you chose another employer or worked in the office.

Mob programming is where several devs work together at once instead of just two. At our company, we needed to implement a graph API refactor. The effort was rather significant, as we needed to update all our models to ensure their queries worked and, of course, update our tests and so on. We couldn't assign this to one pair because the massive overhaul affected all the pairs.

So we got all ten devs in the conference room, put our code up on the presentation screen, and had one keyboard front and center on a table. One dev would code for fifteen minutes, and then we'd rotate. We all knew what we needed to do, and watching it unfold in real time where all the devs got to touch the code and see the changes, and we applied a single-handling approach with it, it didn't take very long at all. We were in the conference room for a few days, and it was done and working.

Remember, **it's not about developer capability. It's about department reliability**. That mob session resulted in all the devs knowing how that migration happened, how the new queries changed things, which files we touched, how we tested it, and so on.

Imagine, in contrast, a team setup like we had, ten juniors and two seniors. Assigning the easy grunt work to the juniors and the architecture and high-value tickets to the seniors would be tempting. But then what? Hopefully, you can see now how common this is and

why this type of task assignment doesn't work. This is why we are so frustrated with our projects when one person owns something. Anytime we need to work on it, change it, or use it, we have to ask this person. What if they're busy? What if they don't want to help? What if they quit?

If you don't currently pair at your job, propose that your department start pairing. If you currently pair and aren't feeling the value, try the pairing station setup and rotation process we used and see if it works for you too.

CHAPTER THIRTY-SIX

Mentoring

One of the most critical milestones in our career is when we start to notice we're no longer reaching out to others for guidance, and others are reaching out to us instead. When it first starts happening, it's a strange feeling. It's a feeling of validation that our hard work is paying off but also a feeling of responsibility and pressure. It's one thing to screw up our own stuff and fix it before someone notices. It's another thing entirely to screw up someone else's stuff, and then we look bad to the whole team.

Teach Them To Fish

There's an old saying, give someone a fish, and you feed them for a day. Teach them how to fish, and you feed them for a lifetime. This is the first and most fundamental aspect of being a mentor. Our goal is to help junior devs help themselves. If every time they come to us for help, we fix their bugs or solve their logic problems for them, we're not helping them in any meaningful way.

Often if a dev comes to me and says, "Hey Sean, can you help me with this? I don't know why it's not working." I first ask, "What is it supposed to do, and what have you tried?" This is a practice I picked up on years ago on StackOverflow.com. Instead of blurting out answers to vague questions, we comment to get more information from the asker. 9/10 times they can solve the bug independently but just forget to do a simple troubleshooting step.

Imagine the bug is their promise chain isn't working; we look at the code and immediately notice that one of the functions in the chain doesn't return anything. Of course, the subsequent function gets `undefined` as its param, and things can break.

Instead of simply pointing out that obvious detail, we can start to ask process questions: "Are your unit tests passing?" That question implies that they should be covering their code with tests. Tests would undoubtedly reveal the issue here because if each function in the chain works fine in the tests, one only needs to compare the test code with the live code and see what is being reproduced. The function should have a test that fails when its param is missing, so that assertion should present itself as an option of something to rule out when troubleshooting.

Imagine our function has five assertions: one happy path and four sad paths. The first thing we should do if our code isn't working is check off those four sad paths in our live code and see if we're causing a sad path.

So the lesson here to the dev is to *check the tests before asking Sean*. That's a much easier thing to remember than making sure your function

returns something because the missing return was a typo mistake, not a misunderstanding of how functions work. It would be like telling someone, "Remember not to stub your toe in the middle of the night." Of course, we know this, but sometimes it happens.

The more we can teach a dev to fish, the more they will remember that we have our shit together, and they should come to us when they are *really* stuck on something. In the past, I've taught brand new devs how to use StackOverflow and have told them, "If the linter and the docs aren't helping, post the question there, and maybe someone has a quick pointer to unblock you."

In practice, we can't possibly babysit junior devs all day on larger teams, teaching the basics of our language syntax. That *is* their job, and they *are* getting paid for it, so they need to own responsibility and accountability for that skill set. Where we as mentors can come into play is when the dev needs help adopting our team's way of doing something. A design pattern may be new, or we're using a tool they've never used before. We want to treat them as intelligent and capable people who happen to be new to an environment in which we're very comfortable.

Practice Consulting

Providing options instead of answers is also critical to being a consultant. As I've mentioned in earlier chapters, if a client asks us how we'd recommend doing something, we want to try to avoid telling them "the way" or "the answer" because, as consultants, remember **we're not here to be right, we're here to get it right**. I recommend providing some options, and then we can steer them a bit if we have a preferred option.

If we pretend that our junior devs are our clients, we can reframe these interactions a bit so that when they come to us for help with something, we use the opportunity to practice our consulting chops. For example, if a dev is tasked with adding a date picker to the UI, and they've never done that before, we may have a quick exchange like this:

- | **Dev:** "Hey Sean, have you ever used a date picker?"
- | **Me:** "Too many times; what's up?"
- | **Dev:** "I need to add one to the UI on this page; I don't even know where to begin."
- | **Me:** "Well, typically, with components like that, if it's something that could be used on any one of our projects, we want to try to avoid building it. Are we using a component library for anything else?"
- | **Dev:** "Yes, but the date picker on the kitchen sink page doesn't look the same as the one in the comp. Should I use it and try to re-style it to match?"
- | **Me:** "Well, you have a few options. If the functionality is all there, see how much work it takes to restyle. Often, component libraries have themes, and we may get it very close very quickly by applying our app's theme to the date picker component. If the styling gets tricky, you can also ask the designer if it's ok if we have some flexibility in the look and feel of the component or if we can release it initially with a default theme and then update it later to match our branding better."
- | **Dev:** "Perfect, thanks!"

So notice I didn't tell the dev what to do, and I didn't tell the dev what I

would do because they didn't ask. I presented some options and some counter options, and now the dev has a playbook of next steps to continue work and see what they can come up with.

Practice Patience

Part of being a good mentor is remembering our early programming days when we struggled with things that may seem trivial today. Even in scenarios where "read the docs" is an obvious solution, sometimes docs can be intimidating to new devs because navigating shitty docs is a huge pain if we don't know what we're looking for, right?

So it's crucial that we practice and demonstrate patience with our junior devs. Most of the time, we actually want them to slow down because juniors tend to *go go go*, and they're sprinting in the wrong direction and breaking stuff. We need to slow them down, calm them down, and lay some groundwork for them. They may come to us five times for the same bug and still can't solve it or understand why something isn't working. We absolutely can not expect them to know what they don't know. **Nothing is obvious until it's obvious.**

If we feel that things should be evident and other devs are not seeing the answers we see, it says more about us than them. We're likely excelling and developing our intuition and building that gut we discussed in an earlier chapter. Answers become obvious, paths to success become apparent, and paths that stray from guaranteed success sound silly. We can't assume others on our teams are at that same level. Using empathy, as discussed in the "Communication Mastery" chapter, we need to believe that the situation is difficult for them, stand with them, and offer help and support because that's what good leaders do.

Practice Empowerment

To re-emphasize once more, the guidance we offer to our junior peers needs to plant seeds to get their minds working. In one recent project, I worked with a client who had an existing product in Java/JSP with Oracle. Part of our SoW was to train their engineering department in NodeJS with microservices and Amazon RDS using MySQL syntax, among other things. The entire project was a mentorship project. We had around fifteen devs from the client side who were all competent developers but brand new to the new stack.

Working with these devs, the mentorship process was often about using analogies between the two stacks. Still, fundamental differences were complex for even their design team to adopt. For example, their old system was synchronous, and the new system was asynchronous. So in the old system, when someone would upload a file, the user would see a spinner and have to wait upwards of several minutes while their system would scan for viruses, make multiple variants of the file, save each variant in Oracle as a blob, and write some records for order state.

The new system would upload the file and do all that processing behind the scenes so we could unblock the user, and they could upload their next file. Part of our job was to empower the client devs to understand the new system so well that *they* could work with their designers and help change mocks to reflect accurate flows.

Simple things like "We can't show a thumbnail on this screen anymore because we don't know if it's been created yet, but we also don't need to show a spinner anymore and can auto-close the modal" was utterly foreign to the designers who've only known that previous way for several years.

Empowering their devs to have those educational conversations meant that their engineering department could demonstrate value to the company. We weren't coming in to save the day; we were coming in to empower and collaborate.

The same mindset applies to junior devs who want your help. Learn to help them help themselves, and they may sing your praises in their status reports.

Recognize and Appreciate

Lastly, part of being a good mentor is recognizing and celebrating progress. We all feel good when our hard work is noticed, especially when trying to impress someone we admire. When we give our reports, we don't just want to say, "I worked with so and so yesterday on the date picker"; we want to say something like, "I worked with so and so yesterday, and we aligned on some next steps for the date picker. I'm really impressed how quickly they are ramping up, and I think we're in a great place for the next sprint."

Notice we didn't claim any credit for their work; we just helped steer their thoughts a bit. We gave credit where credit is due and planted a seed that they are doing a great job. They now have that in mind when delivering future stuff, enhancing their character. They will likely want to deliver their best work because other people have heard how great they're doing.

Becoming a mentor is one of the most fulfilling milestones in our career. After two decades of building software, I no longer care about making billionaires richer; it's just a numbing and unfulfilling experience. However, I will stop whatever I'm doing in a heartbeat to unblock a fellow dev on my team because, at the end of the day, we all need to find purpose in our careers. That's literally the reason I'm writing this book for devs just like you. So please, as you grow and excel, remember to pay it forward and appreciate being given a chance to mentor someone who looks up to you.

CHAPTER THIRTY-SEVEN

Conducting Interviews

Of all the tasks I'm responsible for during a given workday, interviewing candidates is near the bottom of my list of things I look forward to. I've interviewed hundreds of candidates in my career and developed a system that gives me what I need without all the nonsense that comes with typical tech interviews.

Let's start with a few unpopular ideas:

Ditch The Whiteboard

Let's rip the bandaid off this one. Whiteboards are *not* for code. Whiteboards are for capturing ideas for quick iteration in a collaborative setting. I can't count how many times I've seen devs struggle to write code on a whiteboard; devs with masters degrees in computer science struggling to execute simple tasks like "Imagine our language didn't have JSON `parse` and `stringify` support; create those two functions for me."

Super simple stuff; loop over an object or array, purge values of type function, ensure keys are double-quoted, and so on. Accomplishing this task in your everyday IDE or editor would be pretty trivial. We get code completion, and we can see the entirety of the function body in a few hundred pixels of screen real estate. It's just a natural and low-pressure task like creating any other utility function.

Trying to do that on a whiteboard, we don't know where to put closing curly braces; the literal size of the function is now measured in inches or feet vs. Pixels, so it would require us to keep walking back to see everything at once, and the whole process implies that we know precisely what the function is going to look like before we start writing.

Any career dev knows that we often iterate when writing a new function for the first time. We type, erase, rename, format, leverage code completion to give hints for native functions we may rarely use, and so on.

A clumsy whiteboard demonstration merely proves that the dev struggles to whiteboard, *not* that the dev struggles to code or comprehend the deliverable. It's an unfair practice, adds unnecessary pressure, and skews the candidate's capabilities. We might as well ask the candidate to sing the function to us and decide whether to hire them based on how well they stayed in tune. I stopped asking my candidates to do anything on the whiteboard years ago. In my own interviews for jobs, if I were asked to whiteboard, I would reply, "Actually, I'd prefer to type it in a REPL or an editor or talk you through how I would build the

function. I brought my laptop to use for this part of the interview." I'll cover this more in the "Interview Mastery" chapter.

If I don't get the job because I refuse to whiteboard, great, it's not a job I would want anyway.

Stop Holding The Keys

Taking my approach a step further, it also doesn't make sense to ask a dev to solve a tricky problem while we're watching the dev and already know the solution. Several years ago, I migrated away from this process when I noticed how devs were reacting to pressure. It wasn't the pressure of needing to deliver something quickly; it was the pressure of not wanting to make a typo, not wanting to try things, not wanting to look up patterns or examples, etc. This is silly because that's precisely how we code in our workday. We use references, we Google, and we iterate.

I would rather have a resourceful dev on my team who can quickly recognize a good resource and discard a bad one. I want a dev who can hop on Google, and within a few seconds, they already know what they're looking for, and they can scroll past all the garbage and narrow in on a good idea in a minute or less. I realized that I don't care if the dev has the syntax memorized or not--it ultimately doesn't matter, and I think the skills of being able to sift through BS to find hidden gems and "Aha!" moments are essential to have on our team.

We always have spike tickets like "Research this new tool" or "Research some options to solve this problem." I don't want a dev to come back five days later with pages of options because that puts that burden of sifting through the BS back on me. This becomes especially true on client projects where we work with legacy systems and deprecated code languages and libraries. We're not expected to be experts in 20-year-old pre-.NET ASP projects, but we should be able to Google that stack, get snippets in front of us, and gather our trust bearings nearly instantly.

So applying this to my interview process, I assign these "solve" tasks as take-home tasks. I tell them to email me their answer in the next day or two, to take their time, get it right, make it clean, and pretend it's something they'd be proud to merge into the `master` branch of a project. Then, I evaluate different things. Given they had extra time, no pressure, and access to infinite resources online, the quality should be *that* much better. It shouldn't be a chicken-scratch solution written

hastily to appease the interview, even if the solution works. If it's sloppy or has typos, or the formatting is all over the place, or if they're using nested loops to parse an array instead of cleanly mapping over it, then I know they're going to likely create more work for us than they'll deliver for us, and I'll pass on the candidate.

I have a coding challenge^{*} that I use for every candidate, which I'll reference below. See if you can solve it yourself. For context, the best answers have been one line of code, and the worst have been upwards of 50 lines of code:

```
const data = [
  {
    a: 'foo',
    b: 'bar',
    c: null,
    d: undefined,
    e: 0,
    f: {
      a: 'fuz',
      b: null,
      c: {
        a: 'biz',
        b: 'bz',
        c: '123',
        d: [
          {
            a: 'foo',
            b: 'bar',
            c: null,
            d: undefined,
            e: 0,
            f: false,
            g: 12,
            h: '13',
            i: {},
            j: [],
            k: [[1]]
          },
          {
            a: 'foo',
            b: 'bar',
            c: null,
            d: undefined,
            e: 0
          }
        ],
        a: 'foo',
        b: 'bar',
        c: null,
        d: undefined,
        e: 0,
        f: '-7',
        g: '3.14159265358979323'
      }
    }
  }
]
```

^{*} <https://runkit.com/seancannon/succeed-in-software-interview-coding-challenge>

```

    }
},
g : 123,
h : '456',
i : false,
j : {},
k : [],
l : [[]],
m : '3.14159265358979323'
};

// Challenge, refactor this cleanse function so it accomplishes
// the following criteria
// - data is not mutated
// - all `null` and `undefined` values are omitted from the returned
data tree
// - all stringified numbers are converted to numbers.
//     Example, '123' becomes 123.

const cleanse = o => o;
cleanse(data);

```

For context, here is the desired output:

```
{
  a : 'foo',
  b : 'bar',
  e : 0,
  f : {
    a : 'fuz',
    c : {
      a : 'biz',
      b : 'buz',
      c : 123,
      d : [
        {
          a : 'foo',
          b : 'bar',
          e : 0,
          f : false,
          g : 12,
          h : 13,
          i : {},
          j : [],
          k : [[]]
        },
        {
          a : 'foo',
          b : 'bar',
          e : 0
        },
        {
          a : 'foo',
          b : 'bar',
          e : 0,
          f : -7,
          g : 3.141592653589793
        }
      ]
    },
    g : 123,
    h : 456,
    i : false,
    j : {},
    k : []
  }
}
```

```
l : [[],  
m : 3.141592653589793  
}
```

The Interview

What does the actual interview entail if the code challenge is take-home and there's no real-time whiteboard demo? I ask typical status questions like "What projects have you been working on recently?" and "What's driving your incentive to move on from your current position?" I want to understand how seamless the onboarding effort will be or if the candidate will require additional training. For the transition incentive, I'm looking for answers like "There's no opportunity for growth at my current employer." A candidate who's outgrown their employer is not the same as a candidate who's grown tired of their employer. I also ask about any memorable wins or pitfalls with any particular tool that has made the candidate a better developer. I touched on this briefly in the "Experience" chapter.

After I finish my intro questions, I typically explain, "So there are two technical challenges, one I'll send with you, and you can do it on your own time and send me the solution, and the other we're going to do together." I'll ask the candidate to pull up StackOverflow.com, and we'll browse some questions that coincide with the stack of the respective project we have in mind for the candidate, and then we pair up on solving the question.

The candidate drives, I navigate, and it's a typical collaborative experience. I can gauge the candidate's ability to pair program and communicate and how well they take suggestions or corrective feedback. Once we solve the challenge (I've never had an interview where we couldn't at least offer next steps for the asker), I explain the take-home challenge and then ask if the candidate has any questions for me. I answer what they want to know, and we wrap up.

The entire interview rarely lasts even 30 minutes. I want the candidate to act and feel as natural as any day they would come into work so I know what we'd be getting. I want to know if we're a good fit because disruption is the last thing we need--a non-team player or dev that has one foot out the door on day one, for example. Our investments into the dev's growth are wasted when they suddenly leave for something that

aligns better with what they wanted in the first place.

When I'm interviewing candidates on behalf of a client, I redirect all salary and logistical questions to key stakeholders inside the org. That conversation is none of my business, and I don't want to know about those negotiations--candidates for my own projects, I pay for time and materials only. I want devs to feel they are paid for every minute they commit to my project, no wasted time, no hurried time. Just build it the right way from day one; things take as long as they take, and everybody is happy.

I recommend adopting my interviewing approaches for a much better success rate with new hires. The trick, however you approach it, is to get the candidate to understand they'll be working *with* you, not *for* you, so keep the interview process collaborative and engaging, and you'll have a better idea of who it is you're considering hiring.

CHAPTER THIRTY-EIGHT

Cross-Team Relationships

This chapter will cover common cross-team relationships and how to get the most out of each one. When I say cross-team, I mean teams with a stake in the project with you, where you may work directly with them on various tasks or deliverables, versus teams to whom you report as your seniors in the organization. We'll cover those in an upcoming chapter.

DevOps

Let's start with DevOps. When starting a new project with a new client, the very first thing I do is align with DevOps to ensure I have access to the tools I need to do my job thoroughly. It will be much simpler if this is an internal DevOps team I've worked with on previous projects. If it's the client's DevOps team or IT department, I'll typically first go through my internal DevOps point person and align on roles, so I don't overstep if they already have plans to try to secure access for our dev team.

For many projects, though, our internal DevOps team is not involved. We may be working on-prem at the client offices or building inside their infrastructure, so our DevOps team has no skin in the game. For the sake of this chapter, let's assume that's the case.

When I reach out to the client's DevOps team, my mindset is that I need the ability to quickly troubleshoot and test theories and new code in sandboxes that branch off the main deployment pipeline. This typically translates into read access for logs, monitoring tools, pre-production cloud services, and also the ability to deploy and test branches on a staging environment if possible.

For VCS, I typically want the ability to create repos, add team members, and configure branch rules for pull request quality control and automation.

How do I go about this? First, I align with a single point person and build rapport by demonstrating respect for their DevOps team and responsibilities. I want to convey to them that my requests are safe and expected and will **keep them personally out of the mud on bug triage**. Often, if DevOps insists on owning access and permissions for branch deployments or VCS settings, we will need to pull them in when we're assigned a bug that needs troubleshooting.

Framing requests like "I'm happy to take these responsibilities off your hands so you can focus on architecture and infrastructure and not get distracted all the time for one-off builds and whatnot" puts me on their

side of the fence in all this, essentially working with them as someone willing to do the grunt work so they can do the crafty stuff.

If the DevOps contact is hesitant to assign access and permissions to me, I'll follow up by asking for access to the bare minimum, like read-only log access. If the staging environment lacks actual customer data, PII exposure is likely a non-issue.

If they refuse me access to that, I make it a point to pull them into every ticket assigned to me that requires me to troubleshoot something on the cloud. These are polite, professional, yet excessive requests for them to check the logs for me, deploy branches for me, and so on. Then, every morning at standup, I'll mention that I'm blocked by DevOps on ticket 123 for whatever reason. This approach will start to spawn requests for access on my behalf by internal key stakeholders such as project managers, engineering directors, or executives who are paying for my time and want expedited results.

DevOps often sends me a private message like, "Hey, let's set you up with direct access. This isn't typical, so please keep it to yourself." This is precisely what I want. DevOps is still known to be the authority. Their team is not yielding authority to the engineering team. Instead, I'm building a bridge across the teams to troubleshoot on behalf of engineering tasks, and DevOps isn't bombarded all day. Then, when we *do* ask them for help with something, it's typically a much larger issue where they would need to get in there and do what they do best with cloud configuration tuning and whatnot.

SecOps

The next team on our list is SecOps. In every project in my career where I had to work with SecOps, they kept their heads in the sand while we were building stuff, and then at the eleventh hour, they suddenly cared about everything and wanted strict compliance. So, that's fine, but it also affects how we approach security in our projects. From day one, we want to have a security-first mindset on every piece of code we're touching, regardless if it's authored by our team or pulled in as an external dependency.

As I mentioned earlier in the "Security" chapter, we don't need to be security experts. It's not really expected of us that we know all the nuances of every exploit and threat. Still, we *do* need to have the most common and readily known exploits handled ahead of time, and we *do* need to structure our project so that it's easy to pen test and easy to add security middleware at gated access points or data scrubs.

For the SecOps team, industry standards, best practices, and well-known tools are the path to victory, meaning we want unanimous applause from that team when they audit our project. Most of the time, the pen test reports come back with some detected threats, but 9/10 times these threats are false alarms.

For example, in our unit tests, we name things like `FAKE_PASSWORD`, and the pen test audits often hate that we use the word "password" and hate that we store those as plaintext, unencrypted in memory. Of course, these are unit tests, and not only are they fake passwords for non-existent accounts or fake accounts we're literally creating and destroying as part of the spec, but they're also never run on production--only locally on our computers and in the CI build pipeline before the container is ever deployed to the cloud.

Having to explain that our threats are false positives sometimes sparks a back-and-forth where they want to save face and claim their tools are more intelligent than that. It's up to us to frame the conversation that we're not trying to establish that we know more than them about

security, but instead that the flag was triggered by the naming, not by an exploit. Often it helps if we can get them to narrow their scans to omit our test code completely.

These back-and-forth interactions can become very frustrating because these teams typically know nothing about programming. They see the alerts in their scans and want us to make them go away by doing what the scan tells us we should do. Suppose the scan's recommended solution makes no sense. In that case, they often can't wrap their heads around that, and often even if we get a verbal understanding on a call with them, a couple of weeks later, their next scan will trigger the same false positive, and they'll pretend we never had a call to talk about it.

You have to understand that even the industries most people think are the most secure, like fintech or banks, are decades behind. Bank login password fields often have penetrable and self-jeopardizing rules, like "Your password must be between 5-12 characters and contain at least one upper case letter, one number, and one special character from this list of 5 allowed special characters." Wow, way to give hackers an attack vector recipe and keep the string short enough that they can brute force it.

As much as I would love to tell this person, "Numbers and special characters don't directly help anything here. I recommend we force the minimum password length to be at least 20 characters, not limit the maximum length, and require multi-factor auth" I don't say anything to them. It's not my job in these calls to be a security consultant. I'm an engineer building the application and architecture to enable their business. So, typically, I bite my tongue and abstract the password verification into some middleware so it can be swapped out and enhanced later. The system will welcome easy patch deployment.

Legal

Next up in our cross-team relationship list is the legal team. This one is short and sweet. I rarely interact directly with legal on any projects. Still, they commonly come into play and can unexpectedly affect our velocity, so it's good to know what we're working with ahead of time. Most of the time, legal has to sign off on the obvious stuff like terms and conditions, privacy policy, acceptable use policy, and other static pages we're required to include in the application.

These pages can typically be punted very late, so it's not much concern to us. When things start to get interesting is like registration forms where we might have to disclose what we plan to do with the prospect's email or personal information. This disclosure likely resides in the privacy policy, but other laws may require us to notify the user *on* the page where we're collecting the data.

For example, I worked on a project a few years ago with an unnecessarily complex UX in place for the newsletter opt-in checkbox. The initial comp included a label next to the box "Yes, I want to receive emails for future offers and savings," and the box was checked by default.

Several days later, the legal team stepped in and said, "That's illegal in Europe. The box can not be checked on behalf of the user." So the marketing team had the devs change the label for European users, which read, "I am not interested in receiving emails for future offers and savings," and the box was unchecked. Legal returned days later, saying, "No, that's still the same thing. If the users take no action on this form, regardless of the country, they will receive newsletter emails. The European users need to check the box to opt-in to newsletters manually."

I was on these calls and wanted to shake the marketing team to stop acting so unethical and shady with their prospects, but my role on these calls was simply a technical consultant. Can engineering effortlessly implement the business ask? Of course. The marketing and legal teams

can go back and forth all day, and it's not my place to chime in.

Some clients want me to, and I'll try to offer a common sense or unbiased observation, but I'm not going to pretend to be a lawyer. Usually, these things line up, as the law is pretty good these days about protecting the interests of Internet users.

Marketing

This is an excellent segue to our next cross-team relationship, the marketing team. This is another short one, but not nearly as sweet. Marketing typically has its own agenda, and our project is often one slice of their pie which usually sits well outside the scope of our day-to-day.

The downside is that they often come to us with one-off requests to get our product talking to their marketing tools. In the best-case scenario, it's a five-minute task to add a Google Analytics or Mixpanel script to our product's loading page. The marketing tool does all it needs to do to track basic metrics like customer interaction times and geo proximity.

Sometimes they have a slew of sales funnels and want particular and custom event hooks wired up in our product to align with those funnels. They want to know when users are engaging and when they're dropping. Often they'll want to A/B test specific funnels to see which ones perform better. They will often run ads with specific entry points into our product, and our code may need to align the appropriate tracker with the marketing campaign. Different marketing teams may use different tools; I've been on projects in the past which have had three or four analytics libraries running, and it's just a mess.

I survive this by closing my eyes and breathing. I aim to isolate their bolt-on third-party code requirements as much as possible. I used an example earlier about leveraging the strategy pattern to isolate Amplitude vs. Mixpanel, and I mentioned in the "Testing" chapter about using the strategy pattern to support A/B testing, so hopefully, you can see how this is all coming together and *why* you would want to leverage that pattern so much.

These teams do *not* care about our code quality or day-to-day sanity dealing with code spaghetti. It's really up to us to keep our house clean and not let their false sense of urgency influence our standards. There are clean, safe, and organized ways to get their requests into our codebase rapidly, and resorting to third-party copypasta so we can put

the marketing team behind us is *not* the answer.

The good news is once you have these patterns working on a project, you can keep them wherever you go. They're just patterns. I'm not telling you to steal code from your clients. We never do that and I've already made that point. I'm telling you to approach solving challenges in ways that don't *require* you to steal code from your clients. If you can imagine saying, "This is how I make API calls no matter the project," you can also imagine saying, "This is how I organize third-party integrations, no matter the project."

Design

We are moving on to the next cross-team relationship, the design team. This one is often the trickiest for devs because when we first start out, most of us build our own small projects and handle the design elements ourselves. We could take a crack at designing our UI or try some off-the-shelf templates or a UI library. We may ask an artsy friend for help with some nice-looking things for our project.

Typically once we have our project working, whether it's a video game or a website, we can inadvertently develop a biased and skewed opinion of the visuals. We'll love it if it's something we love, or we'll hate it if it's something we hate. This contrasts with loving it because it's an objectively good design or hating it because it lacks conventions.

The primary reason is that if we're not trained designers, we can't recognize what we're looking at. For example, we may like the colors because we like the colors, not because the colors are appropriate or effective. An experienced and professional designer would understand that a healthcare or insurance application might consider blue tones because blue is associated with trust. For an environmentally friendly product application, they might consider greens because green is associated with nature and Earth.

What tends to happen between engineering and design teams is that engineers don't intuitively respect the designs because they don't understand the "why" behind them. The devs may think, "This is silly to have this button here." or "These two fonts should be the same." The designer, however, likely has a particular reason for making that artistic decision, and it's likely founded in customer engagement research from those pesky marketing teams we discussed earlier.

It might be a hard pill to swallow, but devs don't *need* to understand the "why" behind the designs. We just need to respect the design and the designer's role. If the design is terrible and everybody but the designer knows it, it'll eventually change, and we'll have to redo some stuff. That's ok. That's part of the process, and it's the client's or employer's

problem, not ours.

How can we consider all this in a way that helps us? For one, we don't want to write a bunch of nested spaghetti front-end code with in-line styles where one change request requires updating hundreds of files. I like to abstract as much as possible, so iterations are simple.

We want to go into design walkthroughs with a "no big deal" mindset. If the designer wants us to change the entire website from blue to green, "Sure, no big deal." If the designer wants us to change the font size for all headers from 14px to 1.7rem, "Sure, no big deal." If the designer suddenly shows us mobile-friendly views and we've only ever had desktop support, "Sure, no big deal."

So, how does that translate to our world? A responsive first mindset using CSS grids or Flexbox is a good start. Componentizing the UI elements for sure, so changing one file affects its representation everywhere in the project. Themes, for sure, where the first "designs" are the first theme, and then a design overhaul is just another theme that can live alongside the original. Themes are just the strategy pattern for the UI appearance when you think about it. WordPress actually got something right twenty years ago ;)

Similar to how we compartmentalize our services, we want to compartmentalize the front end as well because, ultimately, our job is going to be realizing the vision of the designer, and we do *not* want to do that in a way that requires us to have to unwind and potentially break stuff every time they change their mind. We want to make them look good, but we want to make engineering look better.

With this mindset in our back pocket now, we can encourage the designers to explore their ideas because our codebase is set up for it. We can use them to prove our patterns and make us look like we have our shit together. Once we're on the design team's good side and they enjoy working with us, we'll have much more influence on tricky nuance things.

For example, suppose they designed a date picker; imagine nothing off

the shelf will let us override certain behaviors to make it match. In that case, we can approach the designer from the point of vulnerability and say something like, "Hey, I love your date picker, but nothing in the public packages will let me reproduce what you have in the comp, and it's going to require several days of work to build a custom one. Date pickers are fragile with timezones, daylight savings, and whatnot, and we'd love to use something proven stable if possible. Would you mind browsing a few examples with me to see if we can maybe lightly customize a prefab one instead?"

That request will likely resonate with them because we're asking from a place of collaboration. We *want* to use theirs, but it's going to put us in a bad place, so we're asking them to do us a favor so we can both win. **This is entirely different from suggesting their date picker is inferior or unnecessary because so many prefab plugins already exist.** We're not devaluing their contribution in any way.

This approach works really well, and it'll help build a deep rapport with them, and I recommend you try it out.

Product

Next up in our cross-team relationship list: product. Product owners and project managers often get mixed up. On smaller projects, they can even be the same person. Sometimes there's even a separate product manager, because, why not? Whatever the case, we as engineers want to identify this person or team and ensure we understand who they are and what authority they have over our contributions.

Typically, the project manager answers to the product owner, and we answer to the project manager. Occasionally in sprint planning or roadmap meetings with the PM, we may want to recommend moving features or de-prioritizing work streams and prioritizing others.

To get our way with the product team, we need to respect the product and the product owner's vision in the same way we respect the design and the designer's contribution and role to get our way with the design team. Often on projects, I'll have very strong opinions about features I'm tasked to build. For every feature where I think, "Wow, this is a great idea" or "Wow, this is a clever alternative to other similar products out there," there are four features where I think, "This is so stupid; why are we trying to re-invent the wheel?"

You may be sensing a theme here, but I smile, bite my tongue, and remind myself that I'm brought in on the project to execute an idea--*their* idea. It's not my job to change their ideas to fit my worldview. It's probably one of the more difficult hurdles to overcome on projects because we often crave fulfillment when building software. We want to build great applications and feel good knowing that people are enjoying them or that they're helping make the world a better place.

It's a harsh feeling building something knowing it's a terrible idea and doomed to fail, but I can walk with you on this one, as many of my career deliverables have been thrown away. Products fail for lots of reasons. I'll tell you about a few random ones from my career:

- In the late 1990's I worked with my uncle on a surveying

website project that was going to leverage satellites and charge builders for online survey reports. Late in the project development cycle, my uncle learned that the US government offered the service for free, so it was no longer a profitable avenue, and the project was scrapped.

- In 2001, I worked on an e-commerce site for a client manufacturing and selling martial arts apparel. A couple of months before launching the website, 9/11 happened. Two of his three investors died in the attack, and the third investor backed out because the risk changed. He lost all his funding overnight and could no longer pay me for the project. I did offer to work on credit until he found new funding, but in the end, we all just accepted the unfortunate events and moved on.
- I built a few websites for the band DragonForce over the years, coinciding with respective album releases. I was developing their 2014 album website skin revamp, and the band told me their new label required website and marketing work to be done in-house, so they launched their own, and I scrapped mine.
- In 2015, I worked on a project to visualize some IoT sensors that would be sent to a broadcast booth during televised events. Mid-project, one of the client founders sadly passed away in his sleep, and the project was canceled.
- In 2014, two client partners tasked us to build a website for their start-up to compete with Houzz.com. A few months into the project, we were nearly ready to launch, and they had a big fight and walked away from the project entirely. The fully working site was paid for but never released.

So what's the point of these stories? These are merely a subset of projects where the rug was pulled out. These projects didn't even fail because they were terrible; they just failed. **We can't get attached to the deliverable.** We need to stay attached to *our* contributions. What can we take with us to the next project regardless of the outcome of the current

one? Patterns and experience, my friends.

When working with the product team, like with the design team, we want to organize our code to enable their vision and allow them to change their minds without us having to unwind and potentially break stuff: same patterns, same process. Trust me when I say the "Sure, no problem" approach is highly liberating.

QA

The last cross-team relationship I want to cover is QA. This is a fun one for me because, very early in my career, I looked at QA as a pain in my ass. They would find and make me fix the most seemingly insignificant things. Of course, as I grew, I developed a genuine appreciation for them kicking back work when they found bugs. It's saving us from a bug ticket down the road if a customer happens to experience the issue instead of the QA engineer. Nowadays, thanks to my OCD and test coverage, it's extremely rare when QA kicks anything back to me personally. I can't think of a single ticket of mine that's been rejected by QA in the past five years, for example. Not because I don't *make* mistakes but because I *catch my own* mistakes, which we'll cover in a minute.

Bug tickets tend to be treated differently on projects than feature tickets. They tend to be prioritized and justified as reasons to bypass the release train. I personally don't like the idea of scrambling to accommodate something. Anything done in haste and straying from the process risks introducing more bugs and causing more disruption, so I avoid it when possible.

Ultimately, we want to phase out subjective acceptance criteria. If a work ticket has vague acceptance criteria, where a dev thinks it's done and the QA thinks it's not, that's a problem. Try to identify this situation during the grooming process before sprint planning and work with the project managers who are authoring these tickets to please make measurably objective acceptance criteria. "What is the definition of done?" If it's ambiguous, help reword it. My preferred approach is the Gherkin style. If you have any influence over your PM and the structure of tickets, I *highly* recommend this approach.

Essentially, Gherkin is a way of writing acceptance criteria with these parts:

- | **Scenario** -- imagine a "describe" unit test block
- | **Given** -- imagine the state created in a "beforeEach"
- | **When** -- imagine the left side of the test assertion
- | **Then** -- imagine the right side of the test assertion
- | **And** -- imagine a way to chain assertions

So, acceptance criteria that may have initially read...

The new user can successfully register with the app.

...would instead look something like...

- | **Scenario:** User registration
- | **Given** the registration form loads
- | **And** the user is logged out
- | **When** the user populates the fields
- | **And** there are no validation errors in the inputs
- | **And** the passive email lookup shows no existing user in the system
- | **And** the user clicks the SIGNUP button
- | **Then** the application makes a POST request to the /api/v1/user API
- | **And** the API writes the record to the database
- | **And** the API responds with a 200 HTTP response

You can see here how the definition of success on this ticket is not only objective; it's *testable*. We can write our unit tests to verify every one of these assertions in the Gherkin statement here. Then QA can leverage a tool like Cucumber and write whatever automated tests they want that *also* align with the objective definition of success. When we deliver our ticket, our "done" claim can be backed up by a screenshot of our passing tests or something that conveys to QA, "We've done our due diligence; please verify on your end."

Bringing QA in very early while we're building the project so our *process* is aligned, and then deferring QA handoff until our work has been self-

verified has been an enormous success. It removes a ton of back-and-forth where the QA and engineering teams would otherwise debate subjective deliverables.

Everything in software can become objective, and it's up to us in early process meetings to hold the product team's feet to the fire to make our lives *and* QA's life easier. Product will often say things like "The app needs to be fast," "Scrolling needs to be smooth," or "The product needs to handle high traffic loads." Those types of claims are just asking for kickback. We need those claims to be more like, "The app needs to load in under two seconds on every client," "Scrolling needs to maintain at least 60fps," and "The load balancer needs to be configured to allow 100,000 concurrent users without peaking RAM and CPU past 60%."

If we take the lead on setting our project up to be easily testable from all angles, QA and engineering will have a fantastic working relationship.

CHAPTER THIRTY-NINE

Vertical Relationships

This chapter will cover common vertical relationships and how to make sure you're representing your best self for each audience. When I say vertical relationships, I mean people or departments above you in the organization to whom you report or take direction.

Project Manager

I mentioned earlier that the product owner and project manager might be the same on smaller projects. We covered the product team in the previous chapter, but now we're dealing with someone who may assign our workload for sprints, deciding which tickets get brought in, and so on. We discussed a scenario earlier in the "Communication Mastery" chapter involving a PM who didn't want to pull technical debt tickets into the sprints and how to handle the dialogue to get them to see the value in our request.

The primary goal with the PM is going to be to get them to truly understand that there will be engineering prerequisite work that needs to be done to support feature tickets and that if they want a consistent velocity from us to plan the sprint accurately, then the process also has to accommodate technical infrastructure, technical debt, and buffer for bug triage.

We must have a plan for high-priority bugs--how we deal with them when they come into the sprint after the sprint has started and how we merge and deploy them safely, so we don't introduce regressions. Imagine a team of fifteen engineers all merging code into the `master` branch over several days, and then a bug ticket is presented. The bug affects the current production build, which may be two-to-four weeks behind our development tier.

The process needs to be such that a dev can create a branch from that release tag, fix the bug, deploy that branch through an isolated release train, so QA knows they're reviewing a release patch branch, and then *also* merge the branch into our `master` branch to align with the current sprint work. This is not always a simple merge and forget. Our sprint often includes major refactors, and the bug fix may no longer be relevant or require an entirely different solution.

The point here is that we need to assume that production bug fixes require **twice** the effort because they're potentially two fixes for one bug. For example, imagine we have a bug in some validation logic in a

component. On production, that component lives in a file with another component. Imagine in the last sprint if we abstracted that component into a shared library now owned by another team. That code was tagged already, and it's on the staging tier. In this sprint, we no longer have that component in our repo, so the production bug fix requires that release patch we mentioned earlier, **plus** another branch off the staging release, and it needs to be deployed to two repos managed by two teams.

That's a whole lot of logistics and risk for one bug. It has to be fixed either way, but it should *not* be done in haste, so we need our project managers to understand that the abstractions which give us engineering wins also give us rollback and patch nightmares. We should plan way ahead for them, or our bug fixes risk introducing new bugs, and then every sprint, we're playing catch-up. We'll never get ahead of patchwork.

Tech Lead | Engineering Director

Often the PM doesn't want to hear about it and wants to assign us tickets and hear us say, "Sure, boss, no problem." But that only works with cross-team relationships, *not* vertical relationships. "Sure, boss, no problem" would be sacrificing the quality of our code for the deliverable in question because the compromise is the implementation. In cross-team relationships, we get to keep ownership of the implementation, but in the vertical relationship, the implementation is literally the work, so we need representation here.

If we're the tech lead or director, it's up to us to go to bat for our team. We shield our team from the nonsense. If we're just a dev on the project, we need to stop having these battles with the PM and ask our lead to do it for us. The tech lead or engineering director isn't just there to sign off on tooling and strategy. They are in meetings with the execs and represent the engineering department. Junior devs, **be vocal** with your concerns and get these items on their agenda for their subsequent calls with the PM and the product team because they will likely be involved in sprint planning and grooming, ensuring we have the capacity and velocity to stay on track.

Systems Architect

Let's segue a bit to the architect. Not every project has one, but if it does, we must respect this role and the direction that comes from it. Often the architect will be planning out infrastructure meant to support the product years ahead of schedule. The devs may be assigned tickets to wire up log streams we don't seem to be using, or the architect may get very strict with database schema design and naming conventions and act as a QA of best practices. Devs often won't understand or agree with the direction because they are in the weeds daily--the architect is sitting 10,000 feet above the project while also planting all the seeds to make it work.

It's critically important that the devs trust the architect and not only do the work as asked but also build a strong relationship with this person and ask lots of "why" questions. These questions can't be challenges; instead, they need to come from a place of wanting to learn.

A dev might ask, "Hey, I see you want us to use a NoSQL database for this new service, but we're using a relational database everywhere else. Can you help me understand why you want us to do this? This is a drastic deviation from the norm, so I'm curious where your mind is going with this."

This puts the architect in the role of being your mentor. I can almost guarantee you the answer will not be, "The database doesn't matter; I just want to try NoSQL." It's much more likely to be something in the spirit of, "We want a storage layer between our data lake and our search server because ingested third-party data is going to be a mix of JSON/XML/CSV. This NoSQL DB is our denormalized JSON raw storage for the search index process. A relational database will come soon, with a normalized schema containing common fields across all the third-party data."

Building a rapport with a systems architect can help solidify one of the most invaluable relationships in your career.

CTO | VP Of Engineering

Moving up the ladder, we get to the CTO and the VP of Engineering. At this level, we're well outside the implementation details. These roles are working with cloud providers signing multi-million dollar partnership deals, and helping support the CEO with technical jargon that may wow investors.

Decisions at this level are often press-release-worthy decisions. "Our company is excited to announce that we are rolling out company-wide support for multi-factor authentication to align with the security team's growing concern for outside threats. The integrity of your personal information is our number one priority, and PRODUCT_X will allow us to leverage the most advanced threat protection on the market."

That trickles down to product and project management and eventually engineering as work tickets to wire up APIs, leverage SDKs, and so on. Our sprints are aligned with roadmaps with promises made to the public, investors, or the board of directors by the CTO, so *they* care about our velocity because if we miss a deadline, they look bad publicly.

I wouldn't worry about trying to build a personal relationship with the CTO or VP of Engineering other than getting to a point where once you leave your company, you would feel confident that if you asked them to recommend you on LinkedIn that they would take time out of their day to do it. As for day-to-day influence, they have too much on their mind to micromanage.

Some smaller companies and start-ups have a 20-year-old self-appointed CTO, and they don't even count. I don't take them seriously, and neither should you. You could just as easily form your own company and call yourself CTO, which we'll get into in a later chapter, but the title is just a title; it's what you do with the title that matters.

CEO | VPs | C-Tiers

Moving up the ladder again, we reach the other VPs and the C-Tier folks, including the CEO. I'll give you the same advice as with the CTO, except to also consider that the larger the company, the more detached the CEO is from anything going on in the company.

Enterprise CEOs often act as if they walk on air. I don't have time for people like that. To me, they're just arrogant people in suits, so I don't even bother trying to establish a relationship with them. Smaller companies, agencies, and start-ups are different. The CEO often comes up with the idea that forms the company's backbone. Those individuals can be ambitious visionaries, are often very involved with everything, and can be fun to work with. With these people, we want to set ourselves up to be experienced advisors. We want to give them **options**, not **answers**.

Remember, the company is often their baby--something they built from the ground up, so the answers must be theirs. They don't want other people taking over the direction of their company. They want to be presented with options and make their own decisions so that the success or failure of the company belongs to them and them alone.

This is fine, and it's good for us. Unless our goal is to be co-owner of the company someday, let them have their fun and use their ambition as a conduit to suggest crazy new cool ideas of things we could try. They'll often approve colossal R&D budgets and support the innovation work streams because they understand that's what's going to get their company noticed as a market disruptor.

Human Resources

The last team in our vertical is Human Resources. They are the end-all-be-all. In the Army, my MOS was 95B Military Police. It was the only MOS where an E-1 private could give orders to a five-star general. It was the MOS that kept all the others in line so nobody could abuse their authority.

HR is the same way. They typically have stringent rules to protect the sanctity of the workplace, and nobody is above them. They often have a "We are HR, no matter where we are" attitude, which I love.

Build great relationships with the people on the HR team. They all want what's best for us as workers and want to protect us from executive abuse. Sure they may set some rules which seem overly protective, but if we remind ourselves that everybody has a different story and different hardships and that HR is trying to create a safe workspace that allows workers to work without worrying about anything *but* work, then how can we disagree with that?

Our office atmosphere may be fun, but we're not entitled to it. We may make friends, but we're not entitled to them. We may find romance, but we're not entitled to it. We may be able to decorate our office or cubicle, but we're not entitled to it. You see the theme here. If we appreciate the perks and don't try to bring our outside life into our work life, everything works out great.

CHAPTER FORTY

Developer Relationships

This chapter will cover the art of working with other devs. To simplify this, I'm going to leave titles out of it, so don't get caught up so much in Junior, Mid, Senior, Principal, and so on here, but rather let's group the devs into three primary buckets: devs who you feel are **less experienced than you**, devs who you feel are about **where you are**, and devs who you feel are **more experienced than you**.

Less Experienced

Let's start with less experienced devs. As I mentioned in an earlier chapter, the 80/20 rule suggests that 80% of the devs on any given project will be responsible for delivering 20% of the quality work, and so we consider the 80% to be the entry-level tier of devs. These are typically recent boot camp or college graduates, interns, or, even if they have experience at other companies, maybe they're new to your company or project stack and not contributing at your level.

If you're a brand new developer at the bottom of your respective totem pole, that's fine! There's only one direction for you to go: up, my friend! Take this lesson as future advice when you pass up your peers and want to extend your hand behind you.

Before we dive in too much, I want to split the less experienced group into two separate groups: **devs who recognize they are novices** and **devs who don't**. The former tend to take feedback well, ask intelligent questions, respond to direction, and typically want to collaborate *much* more than the latter. The latter tend to be stubborn and insist on challenging good with bad ideas because they're so caught up in **being right** that they forget we're here to **get it right**. If the team chooses a route better than their approach, they will often act like it's simply an alternative, not an improvement.

We ultimately want to champion the novice devs who understand their place. We want to give them opportunities to grow and shine. We want to pair up with them as much as possible and be there for them as the go-to resource anytime they are stuck or want advice on best practices. Why? Because the more devs on our team we can mold into doing things that align with our vision, the easier the team becomes to work with overall. We start creating ambassadors who echo our recommendations. And maybe it's not our vision or our recommendations. Maybe it's that of the CTO even, but cohesion and consistency far outweigh personal preferences almost every time.

I've mentioned previously that it's not uncommon for four or five devs

on a daily standup to say something like, "Yesterday, Sean helped me with ticket 123" or "Today I'm pairing with Sean on feature XYZ." Helping these devs not only helps the project succeed, but it'll ultimately come my way to make me look really good to the execs. Helping novice devs succeed makes us look like great leaders, and when it comes time to put devs in leadership positions, it'll be a simple decision for management to look to us as that resource.

Regarding that other bucket of devs who lack the skill and don't want to admit it, I tend to simply do what I can to ensure those devs get tickets that don't require the rest of the team to depend on them. For example, when we build APIs for clients, typically someone has to write the Swagger docs, or someone has to write smoke screen tests in Postman, someone has to write the load tests in JMeter, someone has to write the README instructions, and so on. For the front end of website work, someone has to create the React Storybook examples. For bug tickets, we can split them into two tasks--the first is finding and reproducing the bug, and the second is fixing it. We can assign the first part to the problematic devs and the second to the devs we trust.

Most devs don't want to do grunt work, and most don't want to work with novice and arrogant devs, so this is a perfect place for those devs to help out. Not only can we carve off work streams for them so they don't lower morale and introduce bugs in our code, but they can also start to grow and get a better understanding of all the moving parts of the system as they're authoring the documentation.

It's a win/win whether they realize it or not. If they're novices and arrogant to the point where they're making other devs want to quit, or if they're jeopardizing the quality or integrity of the project, I'll usually do what I can to transition them off the project entirely. These days it's easy for me because I can remove them myself. Even more likely, I won't hire them in the first place because they'll never make it past my interview screening process. Several years ago, when I was in the lower to middle regions of the dev pool, I had no say in who hires who, and I'd have to use the appropriate channels to voice my concerns.

Remember, complaining is a slippery slope. Focus on the **process, not**

the **person**, and, more importantly, accompany your concern with a recommendation for improvement. If you complain with no proposed solution, you can come off as entitled and make yourself look bad. If you complain about a particular dev, management often requires you and the dev to mediate and "work things out." Your concern needs to be about the velocity of the team's deliverables or the high amount of bug tickets being introduced that are pulling the team off features. Suppose you list a handful of bugs that should never have been created in the first place, and the engineering manager notices that they were all introduced by the same dev. In that case, the manager can have a direct conversation with that dev, and things will likely work out in your favor.

The more you grow and shine, the less you have to deal with these devs and the more influence you'll have on how you want to leverage them or not. You can lead by example and influence and inspire devs instead of insisting others change to accommodate your worldview.

Equals

Let's move on to equals. This group of devs will change as you grow, so you must know where you fit in the mix. For example, when you're junior, your equals are junior, and when you're senior, your equals are senior. Try to recognize other devs on any project who remind you of yourself when you work with them. These are devs whom, if you paired with them on a new feature, either of you could drive or navigate, and the velocity and quality would be indifferent.

Equals are my favorite devs to work with because alignment on any project is key to having a powerhouse engineering team. Devs who are equal in capability and experience and equal in process and approach are the best of the best. Regardless if you're both right or wrong about a particular issue or situation, you can both grow and fail together. You can treat these devs as an extension of yourself, where maybe they learn something cool over the weekend, and you get a free lesson on Monday or vice versa.

Having another version of you on a project also allows you to bounce ideas off someone without fear of embarrassment. Often if we ask less experienced devs what they think, we may feel like we're risking our image of appearing to be their senior, like a teacher asking a student for help or advice. If we ask a more experienced dev for advice, we may feel we're risking looking like we're not ready for that promotion we're hoping for. Most of the time, that's a personal thing, and we just need to get over it because any of us on any project will be smarter than another dev in *some* area or *some* tool and worse in others.

We all have different tools on our belts and have different levels of comfort with each one. For example, where I may be an expert in JavaScript, I'm an admitted novice in bleeding edge CSS. My competence in CSS is several years behind. If I happen to be helping with some front-end web work, I'll sometimes catch myself wanting to use Flexbox and then be reminded that the conventions have changed to use CSS grids. A novice dev who may have *just* learned about CSS grids and *only* knows about CSS grids will be better at that tool than me,

regardless if I have 25 more years of programming experience than them. So I have no problem asking for help or advice on a particular tool where I recognize someone has a competency that I lack.

Equals are also great because if we are brought in late on a project, and one or more of our equals was on the project from the beginning, our ramp-up will be much faster and more intuitive. They really can make our jobs easier. There will be an inherent risk that management will see us as interchangeable, where they can replace any of us with another, and we become commodities. It's up to us to recognize that technical competence is a mere slice of our value, so how we'll rise above them is all the other skills you're learning in this book that they are not.

As you grow and become a leader in your field, you'll commonly hear the phrase, "I wish we could clone you," because you'll be offering so much more value than anybody else. Regardless of where you are right now, imagine if you *were* cloned; who on your team right now most closely resembles that clone? That's your equal.

More Experienced

The last group of devs I want to address is the more experienced devs. These devs are the best thing to happen to you on a project. The first thing you want to do is buddy up with these devs and get them to be your mentors. You want to leverage their experience and learn from them as quickly as possible so they can help you deliver more wins than you could deliver by learning things the hard way, studying on your own time, and so on.

These more experienced devs may not even be your senior in the dev ranks. You may not report to them, and they may not even earn more money than you. You just have to recognize certain devs that make you say, "Wow, this person is *really* smart" or "This dev is so much more organized and ahead of the curve on these projects; their process is so tuned compared to mine," and then model after them as much as you can where it makes sense.

I may earn \$100,000 more than another dev on my project whom I look up to as an engineer, and I'm still latching on to them like an excited puppy to learn everything they offer. I may have twenty years on them as a programmer, but they may not have a family and may dedicate every night and every weekend to learning all the tools. Maybe they've only been a dev for five years, but all five years have been in an industry sector where I've never worked, and our new project is part of that sector. Where I can see certain problems coming a mile away that they may not recognize, they may be able to offer patterns or conventions for certain tools that we're using that are inherently better than my current conventions. They may be able to offer insight into certain compliance rules for governing bodies that I wouldn't even be aware of. Part of the journey to being the best is recognizing the value in the rest of our team and not letting our ego get in the way.

Keep in mind that more experienced devs may only have a handful of things to offer that you can't, and so if you can identify what those things are, ramp up on them quickly and become their equals, you'll grow faster than anybody else on your team. Once you become their

equal, find the next more experienced dev and start again. Climb the mountain of expertise one hill at a time by finding people ahead of you who've already accomplished what you want to accomplish, do what they did, and follow their path, just like this book outlines. I promise you it works.

Part Four

Growth Strategies

CHAPTER FORTY-ONE

Location Matters

This chapter will discuss how geo-proximity relates to brick-and-mortar developer wages--when to work on-site, when to work remotely, and where to prioritize your focus for your job searches.

Before we dive in, please understand I cannot speak for the local software markets in countries besides the United States. While the countries we typically use for our offshore teams tend to have relatively thriving markets, the resources that work with us often point out that it's much more lucrative for them to work on projects from the U.S., even though their hourly rate is often substantially lower than that of a U.S. resident. So for the rest of this chapter, I'll be referring primarily to projects from U.S.-based companies.

If you currently live outside the United States and want to take on projects in the U.S., please keep in mind that you're not likely to earn near the rates of developers with U.S. citizenship. For example, our typical SoW for projects will often separate onshore and offshore rates, where clients want to cut costs and optimize budget by requiring a percentage of our labor force to come from offshore teams.

The offshore rates are often 20-40% of typical onshore rates. To put it another way, on a project where a senior engineering contractor working in the office in San Francisco can get \$100 per hour or more, we would likely pay the same level senior engineer in Mexico City, Mexico

\$50, or Pune, India \$20-\$40 per hour. Those two city examples are also very different because Mexico City or Guadalajara will be aligned with U.S. time zones. Hence, real-time daily collaboration is much easier. In contrast, Pune or Chennai are half a day ahead of us here in the U.S., so the work we typically assign to offshore teams in opposite time zones will be much less collaborative, likely passive QA work, documentation, or technical writing, perhaps. Your U.S. software work options and income can significantly vary depending on where you live. You need to understand the variables at hand if you want to take control of your career potential.

We often don't set offshore rates here in the U.S. These are the requested rates from the offshore devs or the agency with whom we partner and extend these rates to our clients. So, it's not about exploiting a cheap labor force but recognizing opportunities to leverage resources from areas with lower living costs where the devs can still offer value to our projects.

Taking things back to the U.S., I want to divide the U.S. projects into two main groups: coastal and inland projects. The best-paying projects in the U.S. will typically be from clients and employers that reside in metropolitan cities either on the east coast, like Manhattan or New York City, or from the west coast, like the San Francisco Bay Area. Inland projects will likely pay much less out of the gate, but we can leverage some of the newer, thriving tech cities, such as Denver, Colorado, or Austin, Texas, to make up some of that difference.

There are a few reasons the coastal cities pay more. The primary reason is that the cost of living is so high that competitive wages for software developers need to be much higher to accommodate those living expenses. Also, many tech companies set up their headquarters in the coastal cities because staff talent is plentiful. Still, they need to offer competitive wages to keep engineers from walking two blocks in the other direction and working at a competitor.

I grew up in the Twin Cities of Minnesota. While we have some companies like Best Buy or Target with thriving E-commerce divisions, they aren't paying *near* the wages that a developer could get in San

Francisco working at practically any company with a website. Like, less than half. Walking to my office in San Francisco, I pass LinkedIn, PlayStation, Ubisoft, and several other companies that are constantly hiring and offering junior developers \$100,000 who just graduated from a coding boot camp and have never worked at a software company before.

What worked best for me was to *work* in the city and *live* outside of the city. For example, in my previous home, I lived in Brentwood, CA, about an hour from the city. It was a 3000-square-foot home with a mortgage under \$3,000 per month. Meanwhile, one of my coworkers had a 400-square-foot studio apartment about a mile from our San Francisco office and paid over \$4,000 per month.

Each situation has its pros and cons, of course. I had two hours of daily commuting, and he could walk to his apartment without needing a car. He also had easy access to everything else in the city that could offer a fun after-work life. I have a family who needs a garage for our vehicles and a backyard for our kids and dogs, so living in the city isn't a realistic scenario for me. The upside is that the city income was substantially more than anything I could earn from a company in Brentwood or a nearby minor tech city like Pleasanton, which had some offices for older enterprise companies like HP.

California also has tech towns like Mountain View, Cupertino, or Menlo Park, which house some of the biggest tech companies in the world like Google, Apple, and Facebook, if working for one of the FAANG companies interests you. None of those companies interest me in the slightest, and I'll elaborate on my reasoning in the "Work For An Agency" chapter.

So, the formula I recommend here is matching the highest income potential with the lowest cost of living and putting ourselves in a position where we can grow and prosper while not being locked into a geographical region of any one employer or client. For example, I work for myself and am the CEO of my own software company, Alien Creations. I do most of my work for an agency in San Francisco called Presence. When I lived in California, I'd ride the train into the city to

work in the same office as the other devs, and, depending on the project, I'd often work on-prem at the client office if it was a reasonable distance from the Presence office. In San Francisco, that's usually the case.

Once the COVID-19 pandemic happened, my family relocated to a quiet area in Arizona. I work remotely full-time, still taking projects from Presence to continue our partnership. I set my rates and align with Presence's rates where I can so I'm not putting strain on them, but I'm also remaining ahead of the income curve as much as I can to continue growing. If I had to work in an office in Arizona, my role and responsibilities would have to change drastically to continue earning what I'm earning now, or I'd need to ramp up my sales team to sign my own clients. I'd *still* pull those clients from the coastal cities, but those contracts would likely require me to fly into the city now and then for meetings and whatnot, which I'd like to avoid. I prefer developing software to selling clients and managing a business, so my arrangement with an agency like Presence is perfect for me.

Where do the inland cities come into play? Frequently, employers and clients will have a headquarters in a coastal city, and they'll have another office in one of the inland cities because it's much cheaper for them to find dev resources in those cities who can also work in an office and be managed in person. A typical example is a company that has set up its headquarters in a city like San Francisco. That office might house the design team, the marketing and sales teams, the executive teams, and a subset of the engineering team. Then, the bulk of the engineering workforce would reside in another office in a city like Boulder, Colorado. Those devs would teleconference into meetings with us to understand the tickets that they're being assigned to do the work on whatever features apply to that sprint.

If your goal is career growth, working for that example company will look very different to you if you are part of the San Francisco team or the Boulder team. Same company; with different teams, income brackets, and growth potential. This is why working for yourself as a contractor can protect you from these corporate divisions and the bureaucracy that comes with them.

* * *

Post-COVID pandemic, the question I hear asked most in our industry is, "Does it really make a difference working in the office or working remotely?" I believe there is a difference. As we discussed earlier, the best way to get alignment among engineers is pair programming, and the best way to pair program is to sit side by side with your pair. In-person collaboration also builds morale and rapport; when you want to leverage your team to help you grow your career, you'll have a much better chance of doing that in person. Building rapport through a teleconferencing app like Zoom is *much* more clumsy and less personal. It can be done, but it's nowhere near as effective.

Ultimately, find a working setup that makes you happy and where you can deliver and grow at the pace you want. I've worked with off-shore teams from Mexico, Poland, Sweden, India, and several other countries on countless projects, and I've always been able to build rapport and make friends with at least some of the devs on these remote teams. Not only are we not in the same office, but we're also often not even in the same time zone.

So, while getting what you want from working remotely *is* possible, please don't discard the idea that you may thrive substantially more in an office with other devs. Having a presence about yourself in an office setting where you can build a reputation and be introduced to other people from other teams in passing can't be replicated as an experience by a random Zoom call with someone you've never met before, at least not yet.

When technology improves with VR and AR, we may get to a point where the interactions with our colleagues are just as lifelike as if we were there in person. But for now, recognize what's working for you and what's not so you can make adjustments as you need to for your own career growth.

CHAPTER FORTY-TWO

Always Have A Mentor

My primary mentor when I was learning how to build websites was Eric Jordan, the founder of the digital creative agency 2Advanced. He didn't know me, and I didn't personally know him. Still, he shared his struggles, career path, and life lessons in several articles and interviews, and I figured if he could do it, so could I, so I modeled after him at every opportunity. 2Advanced was building some of the most incredible-looking, cutting-edge websites I had ever seen.

When we think of websites that existed over twenty years ago, we often think of websites that look something like this:

* * *

This website was for ICQ, an extremely popular chat client at the time. This was a very common look and feel for HTML-based websites because CSS hadn't evolved yet to the point of extensive customization and allowing a consistent creative expression on all the major browsers.

Now, let's take a moment to appreciate the 2Advanced company website from 2001. Remember, this site existed at the **same time** as the ICQ website:

* * *

Imagine a twenty-year-old me building websites as ugly as that ICQ example one day stumbling upon a 2Advanced site and seeing their portfolio chocked full of cutting edge web experiences that could hold their own still today even. I was blown away, and it changed my entire life. ***This is what I wanted to be doing.*** These websites were animated, fully interactive, and decades ahead of their time.

2Advanced became my standard of what I wanted to build. They used technologies like Macromedia Flash at the time to circumvent browser styling limitations, and they even developed a method for leveraging hashbang URL anchors to deep link in Flash projects. Hence, their websites behaved like HTML and weren't just interactive experiences that always loaded from the beginning like most other Flash sites.

As the CEO and Creative Director, Eric Jordan was not simply raising the bar for website quality standards--he was also getting huge clients like Ford and Tesla. I learned all the technologies they used, followed all the tutorials they put out, tried to reproduce specific experiences they were building, and really just tried to elevate myself to their level. It was a hefty goal that always seemed slightly out of reach because as I improved year after year, so did Eric.

Unfortunately, Steve Jobs announced that iOS Safari would never be supporting Flash, so the technology phased out completely. 2Advanced had adapted and started putting out HTML5 websites. However, the technology was still unable to accommodate the creative level of 2Advanced; ultimately, the company decided to close its doors, and Eric moved on to do solo creative work.

Earlier, we talked about failure and experience and being able to learn from someone else's failure, and the fall of 2Advanced will sit with me forever. Never trust that any technology will be around forever, and never trust that your industry will ever have your best interests in mind. Sometimes you must realize when the playing field is no longer allowing you to play your best game and move on.

In addition to web development mentorship, I also listened to several

life coaches like Tony Robbins, Brian Tracy, Zig Ziglar, and Robert Kiyosaki to better understand how they were able to pull themselves out of hardships and overcome struggles to achieve success in areas like personal development, finance, and sales. Not everything they wrote about applied to my own life, but there's no doubt in my mind that I would be where I am today if I hadn't applied their training to my career.

Let's get into the definition of a mentor and digest each part so you can have a game plan moving forward in *your* career. The definition I prefer is "**an experienced and trusted advisor.**"

Experienced

The first word in that definition is "**experienced**." We already talked a lot about experience, what it is, why it's essential, and how we can leverage it to offer value to our clients. If we put ourselves in the client role and recognize someone in our network with experience that can offer *us* value, we can come full circle to why experience is so critical in our line of work.

When we look up to someone in our network, whether a colleague or someone relatively famous in our industry that we may be able to reach out to and connect on a distant level, it can be enticing to fall for a sales pitch. Sometimes people put out videos and make a lot of promises, but how can we be sure we're listening to the right people?

I recommend following a process I mentioned in a recent chapter, where we climb the mountain of success one hill at a time. Find a mentor one hill ahead of you, not someone at the top of the mountain. Find someone who's recently accomplished what you hope to accomplish, or find the author of a library that you want to understand better. Your goal here is to expedite your learning curve and **leverage their experience**, so you don't have to **live their experience**.

Suppose we use a fitness analogy. If you're currently overweight and want to lose 40 lbs and get healthy, it may seem enticing to listen to the personal trainer with 6% body fat who's winning physique competitions. But if that trainer has genetically always been 6% body fat since childhood and has never experienced being overweight, how can that trainer help *you* on *your* path? You can try to model after that trainer, but that trainer likely has a fitness routine that works for *them* and will not work for you.

Someone in the gym instead who used to be overweight and has managed to achieve their health and fitness goals is likely a much better resource. At least for initial considerations, you can learn what they did to achieve those goals. If they got surgery, took drugs, or cut corners to take an unhealthy path, you can rule them out as a resource. Still, if they

put in the time and effort and can share things that worked for them and didn't, you can ultimately model after them for your *first* path as an *ideal* path with a much higher likelihood of success.

The same principle applies to software. Someone born into a family that owns a company and was given a C-tier title isn't someone you can model your experience after. Someone who worked their way up the ladder, however, you certainly can.

Trusted

The next word in the definition of a mentor is "**trusted**." It's crucial that the people we choose to model ourselves after are trustworthy and not trying to scam us or have ulterior motives. Sometimes people give advice that benefits *them* but doesn't benefit *us* at all. For example, many coaches online leverage the fear of missing out (FOMO) to get people to buy a product or course that delivers no helpful advice. Maybe the course sells more courses, or maybe we find out that the financial expert online we've been listening to has actually gone bankrupt and is just pretending to be a mentor to scam people.

This is where we want to use the gut instinct we discussed earlier. Learn to read people, assess all the moving pieces, and question people's motives. Most people in our circle will go out of their way to help us because that's what a collaborative working environment is supposed to be like. There's accountability attached to interpersonal working relationships. It's hard to screw someone over if you have to work with them the next day, for example.

People outside of our circles, we need to be careful. If they don't know us, they have no reason to help us, and we should **be wary of the kindness that comes with conditions**. I realize it's not the most healthy way to live, always being skeptical of people. Still, I've only ever been screwed over by people I blindly trusted and have never been screwed over by someone I kept at arm's length until rapport and trust were built.

Advisor

The last word in the definition of a mentor is "**advisor**." We want to ensure that our mentor is offering advice and not just using us. Sometimes when we put ourselves in a position of being a student, we can end up just doing work for them, and they'll frame the work as "experience." Still, if we're not learning or growing from the work, then the work is no longer mutually beneficial, and we should walk away. This happens a lot at hackathons, where a company will use hungry and ambitious devs as free labor to solve an internal challenge they might be having.

In the movie *The Karate Kid*, young Daniel wanted to learn karate, and the sensei, Mr. Miyagi, had Daniel do chores like painting his fence and waxing his cars for hours and hours. None of that work was of any value to Daniel until Daniel finally confronted Mr. Miyagi about it, and Mr. Miyagi explained that those repetitive movements were karate moves in disguise.

There are ways to teach karate without having students do house chores. It's up to us as students to communicate to our mentors if we feel we're being taken advantage of, or else we could be learning without comprehending. Having no frame of reference or mental association doesn't help our careers, even if it makes for a clever Hollywood story.

It's also important to recognize when we're simply displaying an infatuation with someone and not actually learning anything from them. If we're simply a fan of someone because we wish we were in their position, then we're just experiencing envy and not putting them into a mentorship role. In my example of Eric Jordan from 2Advanced, Eric wasn't a mentor to me until I read some personal interviews he had given where he talked about struggles of pulling all-nighters, which clients were causing burnout and which were lucrative for him, which clients tapped into his creative side and which didn't offer any fulfillment. Ultimately, I interpreted his advice always to push boundaries, do great work, and express yourself--the clients will come.

Summary

To summarize, find someone who's where you want to be at your next milestone. Once you reach your milestone, if the mentor has also grown and now resides at your next milestone, keep learning from this mentor. If, however, you are now *equal* to your mentor, they are no longer a mentor, and you need to find someone new who resides at your next milestone.

When you pass up your mentor, remember to keep your arm extended behind you and share your newfound expertise with this person. Mentorship is an extension of a relationship, so it's essential to preserve it and not cut off people when you've learned all you can from them. Fulfillment in life is often gained from helping others, not through achieving your own success. We often develop a teeter-totter mentorship model with someone--they pass us up, we pass them up, and the volley continues because we want nothing more than to see our friend or colleague succeed as we have.

The ideal mentor will sit with you and work with you directly. Maybe a colleague or a professor. These mentors allow you to ask questions and get real-time feedback. Remember this when it's your time to return the favor. The less ideal mentor is detached and doesn't even realize they're a mentor to you. It can work, as it worked for me, and hopefully, I can work for you through this book and my online course. Still, I highly recommend finding someone in your circle to whom you can attach yourself and learn directly from them to achieve what they have achieved.

Repeat this process indefinitely and redefine your goals as you achieve them, and you will achieve greatness far beyond your expectations.

CHAPTER FORTY-THREE

Your Personal Brand

Your personal brand is your professional identity. It's the image that resides in people's minds when they think of you in a public and professional forum. Creating your personal brand involves finding your uniqueness, building a reputation for the things you want to be known for, and then allowing yourself to be known for them.

Personal branding is the conscious and intentional effort to create and influence public perception by positioning yourself as an authority in your industry, elevating your credibility, and differentiating yourself from the competition. The goal is to ultimately advance your career, widen your circle of influence, and have a more significant impact in the industry.

What comes to mind when you think of established software engineers like Douglas Crockford, Robert C. Martin, or Paul Irish? Imagine their unique personality, career focus, the way they engage with their audience, and how they influence positive change in software. Those three developers have three distinct brands. How would you set yourself apart, and how would you align with them?

Think of some famous actors like Ryan Reynolds or Dwayne Johnson. They are highly consistent in how they present themselves and the types of roles they accept. They are very in tune with how they are perceived by the public and can market their personal brands outside of

Hollywood to gain professional equity in other industries like alcohol, where Dwayne Johnson is associated with Teremana Tequila and Ryan Reynolds is associated with Aviation Gin. Knowing nothing about either of these brands of alcohol, it's a safe bet that they are high quality and worth buying because we trust that these actors aren't going to jeopardize their brand image with something that's not successful.

If we take this back to software, the three developers I mentioned will associate themselves with successful, proven technical paradigms and processes. They will advise from experience and demonstrate their findings in their books and videos. If they recommend a pattern or process that's new to us, then there's a good chance that we'll want to adopt it because they're known for solving problems and not introducing them. This is their reputation. What do you want your reputation to be?

Traits & Practicals

Much of this book has outlined various traits and industry subject matter such as ethics and personal values, integrity, security, privacy, etc. These traits should be a part of every software developer's character. Given that we will collect and live by traits that make us good people, we can ultimately decide which traits we want to wear on our sleeves to represent us as a brand and which we can keep behind the curtain.

For example, as you start to market yourself, which we'll cover a bit more in the "Networking" chapter, you can decide your voice. Are you the ambassador of security in software? Are you a leader in the movement for test-driven development? Maybe you want to focus on being the dev that gets pulled in on legacy migration projects because you love to keep up with all the older technologies.

Ultimately, it's up to you.

Find Your Niche

The key is to find a niche that allows you to stand out. In the "Jack of All Trades, Master of Some" chapter, we discussed the 80/20 rule. Here is another opportunity to leverage this by carving off 20% of your traits to represent 80% of your project potential.

For example, my personal brand can be summed up as "I'm typically the most seasoned developer on the project, I'm extremely OCD with test coverage, standards, patterns, and consistency, and I present really well to clients." This translates to me landing many projects where I'm put in an engineering leadership position, directly interfacing with the client and owning accountability for the general infrastructure and development process. As vague as that may sound, consider the contrast: a developer who's new to the stack, accepting tickets from an internal team member like an engineering or project manager, and adhering to someone else's standards and conventions.

At the end of the project, the person who assumes the most risk and accountability will get the most credit. Often this is just the agency that signed the SoW and paid our invoices, but the person who has built the most rapport with the client will get the most referrals. The person who directly brings the most success to the employer or agency will get the most consideration for subsequent work. For me, it's about ensuring I'm irreplaceable and offering a unique value that's hard to replicate by simply adding more resources to the project.

This also means that I don't often take commissioned projects outside of my comfort zone. While I enjoy working on new technologies and new types of projects in my *own* time, I crave job stability and growth rate, and so for me, I would be taking a step backward in my career to work on a new type of project where I wouldn't be in a leadership position. Remember, fail privately and succeed publicly. Sure it's fun to learn new technologies, but there's always time for that in the future. I want to be the most dependable resource on any project I'm on because that's my character; that's my *brand*.

Your Brand

Ultimately, your brand is your brand. If you want to model after me to earn the income I do, that's great. I'm offering you the formula that works for me, and you can use it or not. People always say money doesn't buy happiness. This is 100% true. My annual income was around \$300,000 when the COVID-19 pandemic shut the world down. While I was thankful to be able to continue working 40 hours throughout the quarantine, the rest of my life was tossed into chaos. My jiu-jitsu school had to close, my gym had to close, and no restaurants in my neighborhood were open. It was miserable not being able to go out in the world and live my life. My entire life became work. Work the day, and wait for the next work day.

Had my work day been building enjoyable stuff with exciting technologies, I would have gotten that sense of fulfillment from my work day. Unfortunately, my projects at the time were in the fintech space, and building APIs for banks brought me no joy whatsoever. I did, however, have a sense of reassurance that all my bills were being paid and I wasn't at risk of losing my home, like many other people in the world. So that's the trade-off.

I'm excited to see what you choose for your brand. Are you going to hone the "jack of all trades" angle and be the dev who can work on any project at any moment? Or are you going to hone the "master of some" angle and offer yourself as the best-of-the-best Rust developer to get all the Rust projects in your network?

Whatever you decide, I'm confident you can make a name for yourself and build an instructional course or write a book like me if you ever choose to do so. Build a reputation and get to the point where you don't ever have to look for work because work opportunities are constantly knocking on your door, and you can pick and choose the projects you want.

CHAPTER FORTY-FOUR

Goals Mastery

This chapter will talk about the importance of setting measurable goals and tracking them like we do any other task in our projects.

Throughout this chapter, I'll use goal examples in the present tense because that's how I learned to master the art of setting goals. It's a bit of self-affirmation, but it works. So instead of saying, "I want XYZ," we say, "I have XYZ." Let's dive in and understand each component of this.

Measurable

The first thing we want to align is that **goals need to be measurable**. Simply put, we need a way to identify if we have achieved the goal; how else can we recognize when we are growing or stagnant? For example, avoid setting a goal "I want to be rich" because that means different things to different people, and, as you start to earn more money, you'll likely accrue more expenses as well, and being "rich" won't be as apparent as you think it is.

We need to convert "rich" to a measurable alternative. Maybe it's a dollar amount, or maybe it's home ownership; ultimately, it's up to you, but hopefully, you can see how "I earn \$100,000 per year" is something that you can objectively measure. You either earn that much, or you don't. It's objective, not subjective.

Trackable

The next thing we want to align is that **goals need to be trackable**. We don't want our goals to die on the island known as Someday Isle; "Someday I'll earn \$100,000 per year" or "Someday I'll own my own company." We need to attach a deadline to our goals to set a course of action for achieving the goal and recognizing if we are on track.

For example, in 2007, I earned \$68,000, and I set a goal "I earn \$100,000 by January 1, 2010." That's both measurable and trackable, giving me the timeline that I needed to either get raises of certain amounts by specific times or move to other companies to get those raises I needed. The steps required to reach the goal by the deadline are discussed a little later.

Identify Long Term Objectives

While our goals need to be realistic, our long-term objectives do not. For example, in 2007, if I set a goal, "I earn \$20,000,000 by Jan 1, 2008", I would not have a realistic opportunity to achieve that goal. The success of that goal depends on circumstances out of my control. In our industry, that may come as an acquisition or buyout by a big company, but that still means I would need to create a company with enough disruption in the market to attract big enough competitors that they want to buy me out. All that is possible, but it takes much longer than a few months.

Long-term objectives, however, do not have to seem realistic because they can be pushed several years or decades into the future and introduce an enormous range of possible avenues to achieve them. If people like Elon Musk can exist, so can our belief that anything is possible with enough time and enough commitment.

Identify Milestones

With an understanding of measurable and trackable goals with realistic expectations, the next step is to detach the deadline from the long-term objectives and then set a series of milestones between where we are now and where we want to be. If we currently earn \$68,000 and our long-term objective is to have a net worth of \$20,000,000, consider these example milestones:

- \$100,000
- \$200,000
- \$500,000
- \$1,000,000
- \$5,000,000
- \$10,000,000
- \$20,000,000

From there, we can work our way up the milestone ladder and assign loose deadlines to each one. Maybe they're a few years apart, and as we reach our first milestones, our habits and circumstances change such that we can adjust subsequent ones as necessary. It may not take nearly as long as we think it will. Once we have some liquid capital, we can start to invest.

Maybe part of this path is forming a company, and now we have staff helping us achieve our milestones, or maybe we realize that the way to achieve this objective is to branch out into multiple industries and apply our skills wherever we can, even if it's not in the software field. These longer-term milestones don't need to be mapped out on day one. They should be identified and tracked on the radar so we know whether we are moving toward them.

Identify The Steps

Between each milestone, we finally have our goals. Each goal should be realistic, trackable, and measurable. We need to have a game plan of how we plan to achieve each goal. That game plan can also be broken up into a series of steps, which can also be tracked as goals. Instead of vaguely wandering towards a pie-in-the-sky dream, we're making daily progress which might involve setting appointments with some key stakeholders, enrolling in a course for a required certification, or maybe filling out the paperwork to establish your own corporation.

This hierarchy aligns nicely with our typical agile project hierarchy:



Consistent, daily, measurable progress will bring your milestones closer and closer every single day, and you will crave that productivity and progress. You will thrive for that forward movement and never want to stand still. Standing still is the death of progress. Always recognize when you haven't made progress and **do everything you can to act**. Even one step towards a goal is still progress.

Eat That Frog

The absolute best way to make the most progress on your goals is to tackle the least desirable work before doing the fun work and, of course, doing any work before enjoying your leisure time. Brian Tracy, who is a personal improvement coach I have followed for years, uses an analogy he calls "eat that frog," which symbolizes the scenario that if you had to eat a live frog every day, the best way to do it is first thing in the morning as fast as you can. Spending all day doing other things while staring at the frog will only distract and depress you and make you want to quit. If you do it right away when you start your day, the rest of your day will seem like a breeze by comparison.

Of course, no animals were harmed in this analogy. The idea is simply that procrastination will get you nowhere. Once you get into the habit of doing the hard stuff first, you will crave that sense of accomplishment and look for more challenging and fulfilling tasks. You can see this in people who run marathons or do obstacle course races. People don't participate in a Spartan race in the freezing rain for several hours, exhausting themselves and getting bruised up for a large prize at the end. Only the top 1% do that because they're competing to win, not competing to experience. The 10,000 *other* people participate to experience the feeling of overcoming the difficult challenge. That sense of accomplishment is a **very** real motivator and can make the rest of your life seem well within your control.

Adjust As Necessary

Lastly, adjust your goals as necessary. If you realize your current goal is not taking you in the direction you want, adjust and restructure your goals. If your circumstances change and you realize the long-term objective is no longer for you, change it up. Live your life and do what makes you happy.

Before I got married, I was a performing musician in a heavy metal band. Playing live music was one of my favorite things in life. One day I realized that the environment was beginning to conflict with my other goals. My goal of marrying my fiance and starting a family was unrealistic, considering I was surrounded by alcohol and groupies until 3:00 AM every weekend.

As much as I wanted to be a professional musician as a career, I realized that I wanted a family more. I adjusted my goals so that the music part of my life would be composing and recording music instead of playing live with a band, and I directed my career focus toward software. I still have music in my life, but my goals have become "practice every day" so I don't lose what skills I have. **Careers become hobbies, and hobbies become careers.**

My circumstances may change in the future, and I may want to go back to playing live shows, but being able to adjust and not feel like I failed is all I need to keep moving toward something that offers me fulfillment. My current long-term objective is actually to open my own jiu-jitsu school once I earn my black belt. BJJ gives me more fulfillment than software ever has, but I'm only a blue belt and have bills to pay. The long-term objective is technically on Someday Isle, but with every BJJ class I attend, and every year my kids get closer to graduating and moving out, the dream comes closer to reality. So please remember that your priorities may change as you start to achieve your goals. You don't need to make drastic decisions and abandon your dreams--just make small adjustments and reprioritize. We do it daily in agile software workflows and deliver software just fine.

CHAPTER FORTY-FIVE

Compensation / Title Correlation

This chapter will talk about the importance of your title and how it can attract or repel employment opportunities. Before we dive in, I want to disclose that your mileage may vary with any of these titles I mention. Different sectors of our industry may respond differently to different titles depending on the appeal or stigma surrounding any particular title. I'm going to offer my own opinions based on my own experience and working relationships with people in my network who've had any of the titles I'll cover in this chapter. You can apply that information to how you feel appropriate for your own career goals.

Right off the bat, I'm a sucker for common sense titles that intuitively represent a role and convey professional responsibilities. For example, the title "Tech Lead" implies you are a leader of technical decisions or implementations on your project. It's pretty straightforward. Typically, the tech lead sits right between the engineering workforce and engineering management.

This person will likely be a principal developer with some oversight and leadership responsibilities, managing developers in their technical direction but will likely *not* oversee PTO requests or handle their promotions and raises. It's a particular type of leadership position, and it's used all over our industry.

In contrast, I once interviewed for a startup building a social and

marketing presence for musicians. One of the people who interviewed me had the official title of "JavaScript Ninja." When he told me his title, and I realized he wasn't joking, I lost the ability to take him or the company seriously; I closed any intention of ever accepting the position. I did, however, politely finish the interview, passed the coding challenges and whatnot, and then simply declined their offer in a follow-up email, stating I found another position that was a better fit for me.

In my experience, gimmicks only ever cause problems in our field. What does the title "JavaScript Ninja" even mean? Would I be reporting to the "JavaScript Ninja"? Would he be reporting to me? Would the JavaScript Ninja have any authority on any other technology besides JavaScript? What if the project switched to Python or Go? Would his role have to change to "Python Ninja" or perhaps "Go Ninja Go Ninja Go"? Or would he no longer provide any value to the project because his career is his syntax, and JavaScript is all he can do?

We ultimately want a title that warrants respect from the people who pay us and those who report to us. We also want a title that conveys to future potential clients and employers that we can integrate with their culture seamlessly and offer value from day one. For example, even if I were running a NodeJS project with a React front-end and needed to staff some resources, I would never even interview someone who put JavaScript Ninja on their resume. That title tells me they will come in acting like they know everything and will think of themselves as a rockstar. They're not taking themselves seriously, so how can I trust they'll take my project seriously? Remember, projects don't need rockstars; they need team members who can collaborate, communicate, prioritize, and demonstrate the traits we covered earlier. Even if my assumptions are incorrect, I'm unwilling to take that risk because the risk-to-reward ratio doesn't justify it.

While we want to keep our professional career professional, we don't always *have* to use canned titles that everybody else has. Nine out of ten times, we won't get to decide our title anyhow because our employer has a salary tier directly associated with titles, and the job descriptions are already written up, approved, and managed by HR.

* * *

Some companies choose *not* to offer promotions with raises. I once worked for a company where all the devs who wrote code had the exact title of "Developer," and the concept of a "junior" or "senior" dev was vague and subjective within the culture. Some developers got paid more than others and correlated more with how long the dev had been with the company, and not really if they were a high performer.

Consider a typical LinkedIn profile. Seeing someone with the title "Developer" at a company for six years conveys that they just did daily grind work. It doesn't convey that the dev was ever given more responsibility or was allowed to lead project efforts or manage small teams or anything. Even if the dev got six raises over six years, the LinkedIn resume doesn't communicate that at all.

Now imagine that six-year job was split up such that every two years, there's a new title: two years as "Developer," two years as "Platform Developer," and two years as "Senior Back-End Developer." What a difference these titles make. They tell a story of accomplishment and growth. We can assume that each title change also came with a salary increase for this dev. Regardless if that's true or not, the dev has much more leverage negotiating salary at a new company because there's been an upward trend in the titles given to the dev thus far in their career.

In my personal experience, I had the title "CEO" before any engineering title was given to me by an employer. I created my own company in 2001 called Alien Creations, Inc. Up until then, I had been doing freelance work as an individual, but in 2001 I landed a client that required invoices to be paid to a corporate entity. I incorporated, assumed the role of "President and CEO," and built their website project just like I would have otherwise. Shortly after that, I got a job building industrial PCs and control systems, and the owner knew I built websites, so he asked me to take over the company's website and digital marketing efforts.

At the time, I referred to myself as the "Webmaster" because that was a very common title 20 years ago. Still, I ultimately changed it to "Senior

Architect and Head of Digital Product." That was literally my job: architecting and building their inventory management system and E-Commerce website experience, designing and sending out emails and newsletters, and building digital experiences like a "control system builder" where customers could configure their PCs using an app that I created with Macromedia Flash and AutoCAD. At the time, I had a \$30,000 annual salary, but when it came time to move on to my next employer, which title do you think would give me better leverage when negotiating my new wages: "Webmaster" or "Senior Architect and Head of Digital Product"? **I never tell prospective employers what I earn at my current or previous position. It's none of their business.** I tell them my target rate at the new position, and we negotiate from there.

The idea is that while we don't ever want to lie or deceive our clients and employers, often, our role as a software developer is a tiny afterthought of their business, and they aren't going to do us any favors in helping us market ourselves and demonstrate value to the outside world of prospects and future opportunities. Suppose we cannot self-label our rank in a pool of devs all sharing the same title, like by attaching "Senior" or "Lead" to our title because the employers never offered leadership opportunities for us. In that case, we may still be able to add some flare to our titles by providing some context to what we actually did.

Consider three devs, each with the respective title "Developer," "Platform Developer," and "UI Developer"; they could all be the same *level* of developer in the organization getting paid exactly the same, but when it comes time for those devs to move on to new opportunities, those titles will likely present **different** opportunities with **different** income potentials.

For example, in almost every project I've worked on, the back-end devs get paid more than the front-end devs. This is just the reality of the industry. Front-end work like UI and mobile application development is a more saturated labor force right now. It's also a much more tangible entry point into software development. Devs just starting out can *see* what their code is doing immediately, so all the behind-the-scenes complexity with APIs and databases doesn't really apply because they

can take a static comp from a tool like Figma or Zeplin and write some code to make it an interactive experience. If you consider cost allocation, the front-end devs may have to share the budget with the design team. I only mention this because if you are a front-end developer, your income potential is drastically limited until you become a full-stack developer or more back-end focused.

Let's go through some of my milestone titles and compensation. For some of these, I was paid hourly, and for some, I was paid a salary (bolded, respectively), but I'll provide both numbers just for context:

Year	Hourly Rate	Annual Salary	Title
2000 - 2006	~\$15 - \$20	\$30,000 - ~\$42,000	Senior Architect and Head of Digital Product / Contractor
2010	~\$33	\$68,000	Senior Developer
2011	~\$50	\$103,000	Senior Developer
2012	~\$54	\$112,000	Technical Director
2014	~\$63	\$130,000	Senior Software Engineer
2015	~\$77	\$160,000	Principal Software Developer
2017	~\$89	\$185,000	Head of Product Engineering
2017	\$120	~\$250,000	Lead Engineer / Architect
2018	\$135	~\$281,000	Lead Engineer / Architect
2019	\$140	~\$291,000	Lead Engineer / Architect
2021	\$150	~\$312,000	Lead Engineer / Architect
2022	\$160	~\$333,000	Lead Engineer / Architect

Before I explain this table, I want to remind you that I'm not sharing these numbers to impress you. I'm sharing them to **impress upon you the income potential you can have** if you model after my career approach. I also recognize that I'm severely limiting myself by not striving to climb the executive ladder. I have a colleague who is a CTO earning a salary well over \$700,000 and is awarded over \$1,000,000 in stock options yearly. So my numbers are just numbers. You can strive for them, or you can strive to surpass them. I merely want to disclose what's possible, so you have a frame of reference for your own career goals.

* * *

In the table above, I want you to take notice of three things. First, from 2000-2006 I was definitely a junior to mid-level developer as far as skills go. I could figure things out and get them working, but I wasn't elegant yet and didn't have a solid process. I wasn't leading teams and was still accruing experience that would offer value years later. However, my projects at the time didn't assign me a Junior Developer title, so there's no reason for me to display that or assign it to myself. So while I got lucky in this regard, you can take on your own projects and assign yourself your own title on your resume, being "Head of Whatever" or "Lead Whatever Developer." Words that convey leadership and accountability go a long way, so use them when you can.

The second thing I want you to notice is the wage jump in 2017. That is when I decided to become a full-time contractor and ditch my salary altogether. Salary positions often come with benefits and pay less per hour, and we can end up working nights and weekends essentially for free if the project falls behind schedule. My philosophy is I should be paid for every hour that I realize my employer's dream instead of my own, and in 2017 I made that switch and never looked back. I set my own rates, and then I own the responsibility of ensuring that I deliver value that surpasses it.

For example, \$ 160/hour is a lot of money for a dev if we compare dev wages laterally. Still, when I continuously deliver high-quality code at the velocity of two or more devs who would charge \$90/hour each, my client will pay *less* for the outcome even though they're paying *me* more. It's a lot of pressure on me to deliver top-tier code on every project continuously, but that's where all the experience, patterns, and conventions come into play.

The third thing I want you to notice is that I haven't changed my title in the last five years. It's doing great for me and conveys the job I have and love. I lead engineering teams and engineering efforts and design and help implement systems. There's really no reason or benefit that I can see to changing my title, so I'm leaving it as is until that need presents itself.

As far as career ladders go, if I wanted to work for a company again as a

salaried employee, I'd likely enter at an executive director position or possibly VP of Engineering, depending on the company. From there, it's CTO or bust. Personally, I have no desire whatsoever to work in the executive tier of any company. I attend too many meetings as it currently stands. Plus, I have my own dreams and life goals, and I can't fathom being that attached to any company or product that would require me to dedicate years or decades to any one brand. That's just me, but I want to mention the path so you know where you could go if you followed my path as a model.

I hope you have some context and insight into how titles can help or hurt your career goals. If nothing else, look onward and upward to your mentors and try to earn the titles they have so you can earn the wages they earn. Once you reach that goal, rinse and repeat up the ladder.

CHAPTER FORTY-SIX

Continuing Education

This chapter will discuss the importance of continuing education and how to stay current and relevant without overwhelming yourself. I'm going to assume you have at least a baseline education in the software world if you're reading this book, regardless if you're self-educated, completed a coding boot camp, or have a master's degree in computer science. **It doesn't matter nearly as much in our industry where you learn what you know, just that you can apply what you know.**

I realize that's a hard pill to swallow for some of us. It can seem downright unfair and unjustified if your hundred thousand dollar student loan debt doesn't hold any weight for a job opportunity where you're considered equal to other candidates who taught themselves programming by watching YouTube. All I ask is that you direct that anger and frustration at the education system, navigate the industry the best you can, and realize that certain sectors in our industry demand those prestigious degrees while others don't.

Decades ago, when software engineers were required to write code in Assembly or even with punched cards, these engineers needed to understand all the intricacies of the computers executing the code. Back then, a bug could literally be an insect interfering with the card reads, and a patch was literally tape placed over an incorrectly punched hole. There was a lot more pressure to get it right the first time and perform due diligence before shipping the software.

There's a famous story of a tragic event from the Gulf War in 1991 where a US Patriot Missile battery failed to track an incoming Iraqi Scud Missile. The Scud struck an American Army barracks and killed 28 soldiers, wounding around 100 other people. An investigation determined that the legacy Patriot code, written in Assembly, used 24-bit floating point numbers and had to accurately perform calculations between battery uptime and radar pulses to determine ballistic missile speed and position. The US military updated this software six times during the Gulf War, including an update to support 48-bit floating point numbers for improved accuracy.

Unfortunately, that refactor didn't get applied everywhere it needed to, so at least one subsystem in the code was using the legacy 24-bit calculation. One Patriot Missile battery had a very long uptime and its represented hours in the calculation exceeded the bit range, so the calculation was rounded. The system failed to track the Scud on the radar, and several people died. The Army played it off as an extreme edge case because they had performed "thousands of hours of testing." Still, the bug only presented itself after a certain period, so if every test was followed with a reboot, the testing process would never reveal this particular bug.

For projects like that, the engineers working on extremely low-level code very close to the metal will likely have college degrees or military training specific to computer science and electrical engineering. The reality, however, is that the disaster was caused by the failure to track technical debt and test the system edges appropriately. Understanding the system's capabilities and intricacies versus applying best practices and processes when building, testing, and delivering software are very different skills. Colleges fail to teach that sort of thing.

These days, these low-level projects present themselves more in the IoT space, where we can write code on an Arduino or Raspberry Pi, interact with servos and batteries, and prove concepts for large projects or even build some fun projects to create the ultimate smart home. We don't need college degrees to build even the most complex contraptions with IoT technology because that information is at our fingertips with a quick

Google search. We don't need a college to teach us the material. We only need a college to potentially vouch for us in a particular industry sector, like if you want to be directly involved in IoT's production and manufacturing process and not just do it for client POC work or as a hobby.

For example, the closer the code is to the metal, the higher the risk that something physically will go wrong and users can actually get hurt. For example, software on vehicles that will engage traction control or anti-lock brakes has to work, or people can die. That risk becomes exponentially higher these days with self-driving technology because the code isn't making objective conditional decisions; it's making subjective decisions based on interpretations of what it thinks sensor input represents. This opens up a conversation, like if the car is in a trolley problem situation. The trolley problem is a thought experiment in ethics about a fictional scenario in which an onlooker has the choice to save five people in danger of being hit by a trolley by diverting the trolley to kill just one person. Self-driving technology will inevitably encounter this scenario, like if the car can't stop, should it kill a pedestrian or the driver? Imagine a bug in that code. Imagine *writing* that code.

Companies are well aware of the potential of bugs that can cause harm to living beings, and when they staff resources to work on these projects, the resources need to offer a trusted conduit, and a scapegoat should something go wrong. This is similar to OAuth, where we don't have to worry about trusting the user that their credentials are valid because we trust Google. If Google responds with, "Yep, this person is who they say they are," then we can trust the user because Google trusts the user. That scenario applies to candidates in high-risk positions; we trust MIT, and MIT gave this candidate a degree, so we trust the candidate by proxy. If something goes wrong, it's easier for a company to deflect the blame on MIT than to own it for using poor judgment in hiring a candidate.

When we start putting abstraction layers between the hardware and the runtime, we start getting into lower-risk applications: websites, mobile apps, video games, marketing kiosks, visual effects in movies, and so

on. Bugs can still have consequences, but the risks are much lower. Physical risks become extreme edge cases, like if our visuals cause a strobing effect and give someone a seizure, for example.

The amount of freely available education online for this type of software development is endless. It's extremely common for self-taught individuals to demonstrate competency *far* surpassing anything a college could teach. Colleges are also very slow in rolling out curriculum, and odds are any software you learn in a college will be either an old unpopular language or a potentially deprecated version of a newer one. Graduates from college aren't easily able to apply their recent learnings in the software job market because the market has been running circles around the colleges, and they can't keep up.

This is why coding boot camps do so well. They offer hyper-focused short-term training that teaches all the latest and greatest tools of the day. The problem is that boot camp graduates have no industry skills outside of their syntax. Just because someone can speak a language doesn't mean they can write poetry or a screenplay, for example. Just because someone learned ExpressJS over the past three weeks doesn't mean they will know how to use NodeJS any other way or know how to find their bearings when Express releases a version update.

They won't likely know how to recognize patterns across technologies, like finding routes and controllers when tracking data flows in a Python or PHP project, for example. They won't understand that there are twenty other ways to build a project other than the way they were taught. If we have to extend a client's existing codebase, there's a slim chance that it'll line up with the boot camp graduate's skillset, so that value becomes exceptionally short-lived.

I learned the hard way over twenty years ago. I enlisted in the Army as an MP in 2000 to leverage the GI Bill for college. I enrolled at Brown College in Mendota Heights, MN, and attended the E-Commerce degree program. It was nothing more than an extremely expensive slap in the face. My technology classes included Visual Basic 6, ASP 3.0, "Intro to Java," C++, Database Design (using Microsoft Access), and DHTML; I've used none of it in my career.

* * *

DHTML was only one of two classes that applied slightly, and I already knew more than the class taught. The other class was called "E-Commerce," which at the time talked about concepts like reverse online auctions and understanding merchant accounts and payment gateways. The rest of my classes were all filler nonsense like 20th Century Literature, US History, and some generals like English and Math. \$60,000 later, I had enough disdain and frustration that I became the most self-motivated, self-educated version of me I've ever been, and I just decided to pull up my bootstraps and learn what the jobs demanded on my own.

Web hosting in 2001 was either an expensive Microsoft IIS plan, around \$50 per month, or a cheap Ubuntu plan, around \$10 per month, which was often a shared server with several other customers. Of course, none of the stuff I learned in school would allow me to build common sense websites on an affordable Linux server, so I went out and bought a book called "PHP & MySQL: Web Development" because a lot of the Flash tutorials I was reading had some back-end examples in PHP. This was PHP 4 at the time, so it didn't yet have an actual object model. It was very procedural, but wow, what an easy and intuitive language. I understood and applied everything in that book to build websites with dynamic back-end functionality in a couple of months.

From there, you know the rest. I learned things the hard way, failed, tried again, and succeeded over and over for years. PHP has a slew of problems, which I also discovered the hard way, and it became a fun challenge to try out PHP alternatives at the time, like ColdFusion, JSP, ASP, and Perl. I started to recognize commonalities across all these technologies and realized one would have a benefit over the rest and fall short in another area--pros, cons, and options. College taught me none of this. None of my clients cared, and none of my employers required anything from me for which my college could take credit.

Let's consider that the technology we depend on to do our jobs is constantly evolving and updating, and our baseline education likely stops when we start our careers. It will be critical for us to keep up with changes on our own. Initially, that can seem overwhelming and

intimidating. As our toolbelt grows, we can feel the pressure that all the tools are updating around us, and the amount of continuing education required is escaping our grasp. How can we ever keep up?

I recommend focusing on the LTS updates and dealing with any other updates as the need presents itself. Tools, libraries, and frameworks are constantly releasing bleeding edge stuff, but at least in the web development space, 90% of the time, we can't leverage it all anyhow. For example, CSS constantly releases new sexy features like animations and masking and whatnot, and browsers are typically months or years behind. It's getting better these days where most web browsers have auto-update features, but they often require a user to re-launch the browser, and the way I typically work, I'll go weeks or months without rebooting my laptop or re-launching Chrome. Even as browsers update, they may not adopt all the new CSS features, and some browsers may adopt different features at different times, so if we assume three visitors to our website are on three different browsers, it becomes a challenge to build an experience that accommodates all of them and still utilizes newer features.

Back in the day, we had to download new browser versions manually. It was extremely common for CSS features to work on Chrome and not on Internet Explorer, or they'd work for a percentage of users on a particular browser, but there was no fallback support. Features would often behave differently on Safari and Firefox. We had to adopt patterns and approaches like feature detection to offer a cascading experience for users with layers of fallback so users with the newest browsers would experience the newest effects. Users on older browsers would see something static instead to avoid a wholly broken experience.

Another example is that gRPC has been out since 2016, and no browsers support it. It's substantially faster than HTTP, but browsers haven't fully leveraged all the HTTP 2 features, so we're limited to using gRPC on the back end as a transport layer between microservices. There's the technology we build, and there's the technology we use. We need to recognize that our clients ultimately want stable, common sense, well-adopted options that offer value to their customers--options that don't introduce niche bugs, quirks, and problems into their ecosystem. We

need to accept that **stability is better than volatility** almost every time.

Years ago, I stopped caring about all the fancy new stuff and focused exclusively on my tools' LTS versions. LTS stands for Long Term Support. NodeJS, for example, tags their *even* major versions (12.x.x, 14.x.x, 16.x.x, etc) as LTS and then uses their *odd* major versions (11.x.x, 13.x.x, 15.x.x, etc) for all the latest fun stuff that the developer community can try out and report bugs. The odd versions are highly volatile, and the even versions are not. Why in the world would I ever use an odd version of Node on a client project? Why would I bother learning about a bunch of features on an odd version of Node that may not ever make it to the LTS version?

I don't have time to volunteer as a beta tester to an open-source project, and I'm certainly not going to volunteer my clients to be beta testers either. I'm happy to let the community work out all the kinks in the tool, and then I'll update it to the next LTS if it makes sense to me. Remember, I don't attach myself to any particular tool. My tools are not my career. If the Node community does something stupid and ruins the language, I'll pivot to another tool like Go or Rust, or whatever makes sense for the project of the moment.

If you're curious and love pushing the boundaries of technology, absolutely have fun and play with all the new tools. We discussed that in the "Jack Of All Trades, Master Of Some" chapter. Remember not to play with them on client projects unless you're on an R&D project, and that's the job--to see if something is possible with all the new stuff.

So with the idea that our baseline education got us into the industry and 20% of our toolbelt will likely be used on 80% of our projects, then if we only focus on LTS updates, our continuing education efforts become much more manageable. What about certificates? Getting certified in a particular tool is another short-lived win, as these tools update and deprecate just like any tool. For example, I earned a Brain Bench PHP 5 Certification in 2010 and immediately realized it didn't matter to anyone but me. Certifications are a way for those education bodies to earn money by presenting publicly available documentation and testing that you read it. If you can read the docs and learn the tool, then the

certification doesn't change anything.

Some employers, however, do ask for certifications when hiring resources. They may want to put vendor logos on their website to support their sales efforts. It's another trust proxy situation. If I'm trying to hire a company to build my software, and I've never heard of "Unisol Data Systems" but on their website they display an official "AWS Certified Solutions Architect" badge, then I can likely trust that Unisol will follow all the best practices and build something for me the way AWS recommends. I trust AWS, so I don't need to trust that Unisol has figured out how to do it independently. I only need to trust that Unisol is a legit AWS Certified Solutions Architect, which is pretty easy to verify.

I recommend waiting to obtain these types of certifications until you need them. Putting one or two certifications on your resume can fill that gap if you're struggling to find work because you don't have a portfolio that conveys competency. Then when you build a couple of projects and prove yourself as a skilled engineer, you won't need to display a certification if you don't want to. Certification badges are cool until they aren't. Imagine reading a resume for a candidate, and they mention they are a certified jQuery developer. That can inadvertently hurt their resume because nobody uses jQuery anymore on large projects. A developer flexing a certification ten years too late tells me they are out of touch with technology trends and standards. They'll likely be a liability on the project.

There is one scenario I want to mention where certifications can benefit you. Suppose you are a DevOps professional and want to go all-in on one cloud provider. In that case, I'd recommend loading up your resume with as many certifications from that cloud provider as possible because, with enterprise systems, the DevOps infrastructure is really about orchestration using best practices and widely adopted paradigms. It's not necessarily about creative problem-solving.

So, where a software developer should be advertising themselves as someone who can be agile, flexible, and help the client overcome logical challenges, a DevOps engineer should delegate risk as much as possible

to the cloud provider. Essentially, "Given all the tools the cloud provider offers, this is how they recommend we orchestrate things."

We don't want creative, unique, proprietary ways of implementing load balancing or a CI pipeline. We want proven, intuitive, and recognizable ways. This may vary from cloud provider to cloud provider. Still, in DevOps, there's less "figuring it out" because the cloud provider has essentially already figured it out, and that's what the certification conveys. Suppose a DevOps engineer wants to be more marketable as a consultant. In that case, it's about repeating that certification approach to all the major cloud providers like GCP, AWS, Azure, IBM, etc. Another approach is to get a collection of certifications on a cloud-agnostic stack like Hashicorp.

I recommend picking one cloud provider like AWS to start out and then waiting until you notice some opportunities leaning towards another provider. Then, spend a week studying and loading up all the vendor's certs. Unlike software libraries and languages, where we can lock in a version behind the latest version if we don't want to use the latest releases, cloud providers typically don't version their services and don't roll out breaking changes nearly as often as software tools. They may update and version their CLI or SDK, but they will be highly stable and years apart.

To close out this chapter, learn through practice. In earlier chapters, I mentioned making some *hello world* apps with new tools on the side to learn through failure in your own time and to diversify your portfolio. That's precisely when you want to apply continuing education to your career. Maybe your next *hello world* project is in a new version of a tool you already use. That's perfectly fine and typically all you ever need to do.

CHAPTER FORTY-SEVEN

Interview Mastery

In this chapter, I will teach you the process I use when interviewing for new positions or projects. These interviews are specifically for when an employer is hiring to fill a specific position or an agency has already sold a project. It is resourcing staff to align with specific project needs.

Know The Role

Before we can prepare for our interview, we first need to understand the position we're trying to fill. This is important because how we interview for a lower-tier developer position will be much different than how we interview for a management position.

Lower-level position interviews are going to be much more focused on technical competency, where we need to prove to them that we can integrate seamlessly into their project ecosystem and we won't introduce unreasonable risk for them by slowing down the project's velocity or introducing bugs if we happen to be a novice with the respective stack.

Management position interviews will focus on leadership scenarios and interpersonal communication skills. They want to know if we are a people-person and how we can communicate technical direction to our team. How do we handle an uncooperative or low-performing team member? This is a very different type of interview than "Show me how you traverse a binary tree."

Understanding the position ahead of time can help us prepare for questions in this space. If we're making a relatively lateral career jump, this interview should be a breeze because they'll just be asking things we encounter daily. If we're interviewing for a promotion, the interviewer will likely challenge us with things we may not be familiar with, which is ok. Instead of trying to study for all that stuff, we can prepare ourselves for how we will best answer questions in general which involve things that are new to us.

Dress To Impress

When I arrive at my interviews, I'm always fashionably early and dressed to impress. It doesn't matter to me if it's a startup culture and the interviewers are wearing t-shirts and flip-flops. I will wear a nice button-up with a tie to convey that I'm taking this position seriously and that I'm interviewing *them* as much as they're interviewing *me*. I want the interviewer to be borderline intimidated by my professionalism.

I also want to convey that I have my shit together, and this interview is a choice for me. I'm here because I'm moderately interested in what this position can offer me to support my intended career growth. I don't care if the interviewer is a junior developer or the CTO; they need me more than I need them, and how I dress and present myself needs to send that message if I hope to have any leverage in the interview when it comes time to negotiate compensation.

I've heard an argument that dressing professionally is a way to mask incompetencies--a seasoned developer should have a strong enough reputation to wear whatever they want to an interview. I don't see it this way. My philosophy is to represent myself as a complete package. I'm professional in all areas of my career, and I want to be taken seriously on my first impression.

If I arrived at the interview in a t-shirt and shorts, I would have to work much harder to compensate for my attire and convince them that I'm a competent engineer and software **professional**. What's the point in adding unnecessary difficulty for myself in the interview? Just like we format our code and organize our projects to make our development lives easier, we should similarly dress professionally to make our professional lives easier. Leading by example goes a long way.

Smile And Model

My interviews are typically multi-tiered. Sometimes 8-10 people interview me before discussing logistics and compensation. For these types of interviews, I need to win these people over so that when they convene behind the scenes, they all agree that they want to work with me. At this stage, it's not about impressing them with smarts or competence. It's about reassuring them that I would be a great team player and add a positive vibe to their work day.

In most jobs, we don't end up working directly with everybody who interviews us. Often we will have various directors in the room that we may never see again. Still, their department may depend on the engineering department. Maybe the engineering department hasn't been delivering well, and these directors want high-level process improvement. If they like me and I present well, that's usually all it takes.

I always smile first and try to get them to model after me. If they're overly serious and professional, I'll casually bring my energy down to match theirs. Sometimes these interviewers don't want to be there interviewing me at all--I might be the last candidate of the day in a multi-day interview grind, and they're just exhausted. If that's the case, then acting like an eager puppy will irritate them, so matching their energy and modeling after their subtle body language is critical. Remember, **people tend to like people like them**, so if we can present ourselves as someone who will align with them personality-wise and won't disrupt their day, that may be all we need to get a sign-off from this person.

Demonstrate Value

The goal of our interview is to demonstrate value. Different interviewers are looking for different qualities that they associate with value. Earlier, we talked about finding the underlying motivation. This is where I try to read between the lines from their questions, and I might ask follow-up questions or reflect back to them what I hear to make sure I understand their questions correctly. Once I have an idea of what I believe to be the **why** behind their question, then I can answer to address the **why**.

For example, earlier, I mentioned that for a management position, we might be asked how we would handle an uncooperative or low-performing team member. Sure, this might be a hypothetical question, but there's no reason for us to assume that. It's better if we assume they're asking because there's currently an uncooperative and low-performing member of their team who would potentially report to me. Maybe they want this person gone or to get their act together and start contributing more. This person is likely making the interviewer's life hell, and this question might be a plea for help.

So the value here isn't necessarily being able to deal with the low performer without hurting the low performer's feelings, but rather, being able to sit between the low performer and the interviewer, so the interviewer no longer has to *deal* with the low performer. My answer then would be to perform an initial assessment to see where the gaps are between the low performer and the rest of the team, identify strain points, and then address each gap and strain point in a unique way that makes sense.

For example, if the low performer has a bad attitude and challenges ideas without contributing better ones, I would first stop including this person in meetings and sit in on their behalf so I can improve the meeting morale and then communicate direction directly to the low performer. I can intercept the negative feedback, so the team doesn't have to.

* * *

If the low performer happens to be committing sloppy code or introducing lots of bugs, I would sync up with one of the high performers and set some expectations that I'm looking to this person to be a mentor. Then I would pair the high performer up with the low performer indefinitely until I see some positive change.

We have several options on how we may want to approach these coaching opportunities and how we can adjust and pivot as needed. Ultimately, suppose the low performer proves unfit and has no intention of accommodating the team and company's needs. In that case, I will terminate the dev, and we can find an alternative resource to fill that role.

Notice here I didn't tell the interviewer exactly what I would say to the low performer. I'm not acting out the scenario. I'm giving a very vague playbook of things I would try first. That's all these interviewers typically want--someone willing to take action, assess, adjust, and improve the company overall.

Don't Patronize

Whatever we do, we don't ever want to patronize our interviewers. We don't want to come off as condescending, and we don't want to talk over them as if we're giving canned responses to earn the "best answer" award. For example, one common question is, "What would you say is your worst quality?" This is actually an insightful question, yet I hear self-proclaimed experts online suggest responses like, "Sometimes I work *too* hard and forget to take breaks!"

While that answer may work on a teenager interviewing you for a position at McDonald's, a director considering paying you a quarter million dollar salary will just be insulted or roll their eyes and remove you as a candidate for the position.

A much better answer is a realistic answer. It's ok to show vulnerability and be candid with the current state of our growth as a professional. We can't be experts at everything, so we typically want to **frame these answers in a way that conveys a weakness caused by a strength**.

For example, I've given one answer: "I often struggle to find the right balance when explaining technology, system capabilities, or limitations in meetings. For example, I tend to start walking through low-level and intricate implementation flows, and then I'll notice I'm losing the execs. If I try to keep the explanations high-level and product or UX-focused, I find some devs on the call can't translate that into implementation details. So I often find I need to have two sets of meetings, one that's more product-focused and one that's more technology focused, and I'm still trying to learn how to find that balance to appease multiple audiences at once."

That's an entirely true answer. It conveys to the interviewer that I'm analytical and self-aware--I recognize how I affect other team members and consider their time and needs. I aim to be the best team player I can be, and being candid about this self-limitation gives the interviewer a better picture of what working with me will be like, and maybe this is a non-issue for them. Maybe they've found that balance and can work

directly with me to help me improve, and then it's an easy win for both of us. Worst case scenario, I'm in two meetings instead of one, and maybe instead of a combined one-hour call, we can have two 30-minute calls, and no time is lost.

Provide Options Not Answers

While there are inevitably going to be closed-ended questions like "Solve this formula" or "Fix this bug," there are also potentially going to be several open-ended questions like "How would you approach this?" or "What would you recommend to a client needing this?". For these questions, we want to provide options, not answers.

I've used this phrase a few times, but I really want to emphasize that **we're not here to be right; we're here to get it right**. Interviewers aren't likely looking for someone to take over the team and enforce authoritarian law on the process and syntax. When being interviewed, I consciously want to avoid a "Sean's way or the highway" type of answer.

Different challenges present different options for different solutions that yield different risks and rewards. If we're joining an existing company with an existing team, we must understand that we share accountability with everybody else. We need to be able to work with them and work with the company's limitations and corporate bureaucracy.

Some companies have strict security mandates that severely limit our flexibility in the engineering department. It slows the pace and requires us to get sign-off on several things like libraries, frameworks, patterns, vendors, and so on. If our proposed solutions don't align with or accommodate the security needs, then we don't really offer any value to the company.

We must present multiple options, including the best-case scenario and other scenarios that accommodate blockers and hurdles. Then we point out any risks associated with those hurdles and let the team and organization assess whether the risks are acceptable. Teams like SecOps have goals like "Don't allow the company to be hacked" and "Don't allow PII data to be leaked." *How* we achieve those goals should be a conversation.

There's likely already a solution or process in place. Still, if we need to

build new features so the company can remain profitable, and if one or more of the security policies prevents us from doing that, then we need to be willing to work with SecOps to see if we can accomplish the same goal through another paradigm that will allow us to achieve *our* goal as well. If we can communicate that to our interviewers, they will typically want us to fill the position as soon as possible.

Control Coding Challenges

Lastly, I want to talk about how I handle coding challenges in interviews. You already know where I stand with writing code on a whiteboard, and if an interviewer asks me to do it, I politely decline. If the challenge is super simple, I'm not going to die on that hill, and I'll write it out on the whiteboard to appease them. But if it's something where I need to think or work through it a bit, I'll communicate to them, "I haven't written code on a whiteboard in years. Would you be comfortable if I typed it out?"

I bring my laptop to every interview, so there's really no reason not to accommodate this. If they say no, then that's my cue that I absolutely don't want to work here. I'll say, "I understand. I think my working style might not be a good fit then, so let's call it a day, and I'll follow up with HR this afternoon. Thanks for your time." I'll then email HR, "Thanks for the opportunity today. Unfortunately, I wasn't comfortable with the style of one of the interviews, so I'll be considering another opportunity. Best of luck filling the position."

I realize it's a power play, but as I said, I'm interviewing the company as much as they're interviewing me. It's important to recognize if the culture is a good fit from the start. I want to work where the process is flexible and open to improvement. For that interview, is it important that I can solve the problem, or is it important that I can use a whiteboard? Insistence on the whiteboard tells me that the interviewer can't distinguish between the means and the end. It's ok not to be able to make that distinction, but to insist and refuse to consider it is where they would lose me, personally, and I will go out of my way to avoid working with them.

Assess Your Needs

If you are desperate for the job and willing to jump through whatever hoops they put in front of you, then do what you need to do, friend. I sincerely mean that. Maybe it's not even about a dream gig. Sometimes our situation demands that we secure immediate income for our family and aren't able to pick and choose.

In these circumstances, do what needs to be done, but don't overlook the reality that in our industry, these companies really don't care about us or our life situation, and we should always be looking onward to bettering ourselves. **The moment we no longer make financial sense to the company, they fire thousands of us with plans to hire new devs later on when it makes financial sense again.**

Consider Mark Zuckerberg's open letter on November 9, 2022:

"Today I'm sharing some of the most difficult changes we've made in Meta's history. I've decided to reduce the size of our team by about 13% and let more than 11,000 of our talented employees go. We are also taking a number of additional steps to become a leaner and more efficient company by cutting discretionary spending and extending our hiring freeze through Q1."

He essentially made his problem 11,000 people's problem. How many of those people were prepared for the layoff? Some of them had likely worked at Meta since Facebook was a startup. Loyalty means nothing because these companies answer to their top shareholders, not their staff.

This is why we should ultimately work for ourselves. **We should not tie our value to any language, stack, project, or company.** When we get hired or take on a client project, we don't necessarily need to keep one foot out the door at all times, but we should certainly keep our ear to the ground and **always be ready** to accept a better opportunity when it presents itself. We must be able to bounce back from an unexpected layoff or project cancellation as if we were ready for it the whole time because we *should* be ready for it at all times.

CHAPTER FORTY-EIGHT

Work For An Agency

I've emphasized already that our career should not be our syntax. We don't want to associate our value with any language, pattern, stack, project, or company. The last item in that list is the most critical of all of them. When we work for a company helping to build its product, whether a B2C or B2B product or service, we become integrated with that company's culture. The longer we work there, the more integrated we become. The company has made decisions on things involving innovation and growth versus stability and reliability. The primary goal of any company is to make money; most of the time, if something is working and profitable, the company doesn't want to touch it.

Change in these companies has to be justified. It has to come from industry and market demands or customer feedback. Sometimes a company falls behind in market share and invests in innovation to launch a disrupter to hopefully gain back some of that market share. How this all translates for us as engineers is that the tools and processes used to build these products and services *rarely* change. **Companies want to leverage established processes to build products that earn money.** When it comes time to try to build something new, the engineering staff at the company has often worked on the same tools for so long that the technology has evolved around them, and they cannot lead that innovation effort.

Companies often resort to hiring an outside contracting agency to

spearhead this effort. Sometimes it's a team augmentation arrangement where the agency will help train and work alongside staff engineers while the new product or feature is built. Still, sometimes it's a solo project effort built in a silo to replace an existing system. If you are working at this company, it can seem like it doesn't trust your abilities and doesn't want to spend any money on you to help you learn and grow. If the new product or feature is released, the company and agency get the credit, but *you* don't get the credit. It isn't easy to put on your resume unless you work directly with the agency.

One genuine reason companies hire agencies is that they can fire agencies. These days, it's becoming increasingly difficult to fire employees because employees can come back and sue the employer for discrimination claims. It's an expensive and cumbersome process with lots of paperwork to deal with that scenario. When the company hires an agency, the devs, designers, and project managers are just contractors. If one contractor isn't working out, the company tells the agency, "We don't like this resource; please use someone else," and then it's just a simple project reassignment.

Another fundamental reason companies hire agencies is because it saves a lot of time and money to retain contractors who can work with the new technology from day one, which will be used to build the new product or feature. Suppose a company wanted to train its engineering department and ramp them up on new tools and technologies. In that case, it could take several weeks or months, and they would likely have to hire an outside consultant to oversee the training. That's potentially tens or hundreds of thousands of dollars to get to the starting line.

Imagine a scenario where you're working at a company building and maintaining their product, and the next thing you know, fifteen years have passed. Sure, you were a junior dev when you started, and maybe now you're a senior dev or engineering manager, but the last fifteen years have been in a bubble of one company's way of doing things. If you get laid off or decide you'd like to work somewhere else, your fifteen years of experience are *not* going to transfer seamlessly into the job market to be applied to a new position, regardless of your title. You may get interviews and consideration, but you will undoubtedly put

yourself in a position where you will struggle with unfamiliar technology and processes, and your perceived value will not align with your title.

For example, I have a close friend who worked at HP for twenty years in the San Francisco Bay Area. He worked his way up the ladder and even transitioned from engineering to project management roles. One day, the company told him that they were closing that office and that if he wanted to keep his position, he'd have to relocate to another state where they had an office needing that role; otherwise, he had 30 days to look for another job. That "other job" also included job openings within HP since they were going to make him apply for internal positions instead of simply reassigning him.

So, he started interviewing for PM positions in the Bay Area and soon realized that none of HP's paradigms for project management followed any industry-standard other companies were using. HP had its esoteric interpretation of agile, just like they had their esoteric interpretation of lots of things. I won't judge HP here. They were doing what they felt was best for HP, and most enterprise companies take years to adopt new processes. There are almost always compromises and accommodations, so this situation doesn't really surprise me at all.

The unfortunate reality here is that my friend's twenty years of experience meant nothing to the job market. He had to decide if he wanted to start from scratch and learn all the new agile buzzwords and processes or relocate his family to another state. Ultimately he decided on a career shift and found a webmaster position for a local city because he always dabbled in code in his spare time. It was a significant pay cut for him, but he was able to stay in his home. The moral of this story is that two decades of growth in one company sabotaged his marketability. This is where an agency solves everything.

Working at a software agency is incredibly liberating, even though it offers its own set of challenges. The agency will retain a client (often an enterprise company, as I described above) and then assign some resources to the project. In my own experience, if I'm a good fit, I'll be assigned. Depending on the project, my role may vary. Sometimes I

build the entire project deliverable by myself. Sometimes, I lead a team of engineers and oversee processes and patterns. Sometimes I'm entirely hands-off, architect the system, and communicate process updates and approaches with the client. It's ultimately up to the client, the respective budget and timeline, and the available resources we have on hand.

Some projects are standalone greenfield experiences where we get to decide the stack, and in some projects, we need to extend a legacy system or migrate it to a newer technology, so we have to work with the stack that's in place. This means every project requires a comprehensive analysis of requirements and technology options. The language may change, the version of the language may change, or the framework built on the language may change. Some projects are small and static, like for a marketing experience, and some are large and dynamic, like a SaaS product.

Considering that you get a chance to level up your abilities and exposure to new things every few months (even if those new things are old legacy), imagine how your value as a consultant and programmer can grow. Instead of spending ten years working on a single Java/Wildfly application, you spend ten years building 30-40 completely unique applications on completely unique stacks with different teams for different clients that all have their own unique constraints and industry demands. Those are **incredibly** different decades that populate **significantly** different resumes.

Suppose you are a video game developer. In that case, your game studio also benefits like an agency because every new game you work on will likely have some upgrades and variance to the code, the game engine, and the process. After all, the hardware technology is improving so quickly that if studios want to stay current, they have to be able to publish games on new technologies as they are released to the market.

Even in mobile game studios, you will experience a variety because iOS and Android are constantly updating and offering new APIs. Depending on the game, it might make more sense to build it in Swift, or it might make more sense to build it in Unity or Unreal Engine. These skills and capabilities will allow you to jump around the job market to

different studios and offer value all the same.

Consider the earlier chapter where we talked about continuing education. How amazing of an opportunity where our employer includes a continuing education spike each time we start a new project. Our jobs are not to be experts on the client's stack before we start the project. Our jobs are to be experts in **discovery** and **ramp-up** so we can provide options and design a plan of attack for how we want to integrate or migrate their existing system to meet their needs.

So while I'm certainly biased and insistent on an agency gig being exponentially more beneficial to us as engineers, I want to address some less-than-ideal scenarios that can present themselves so you can recognize them if you encounter them. Regardless of the client our agency retains, we typically have to move at the client's pace, which can be slow for an enterprise client.

It's common for an enterprise client project to last a couple of years or more, and it can feel like we're stuck working *for* the enterprise company as one of their staff as the project scales and extends. It's up to us to communicate to the execs at our agency whether or not this aligns with our career goals. If you like the consistency of your day-to-day and don't mind the work and the stack, you can ride out these projects, and they become easy money. Be careful, though, as you absorb the same risks as being one of the stagnant employees, as I mentioned earlier.

However, if you're like me and crave progress and growth, you may want to do what I do. I communicate that I don't want to become stagnant and want project rotation at least once per year, if not every few months. If the agency can accommodate that, great, and if they can't, then I work with them where I can, so we're helping each other out. We need to be careful communicating project roll-off demands, though, because depending on the agency, there may not be a considerable backlog of client work, and we can risk being transitioned off a project and experiencing a pay gap between projects.

To address this, I recommend having multiple sources of work on hand at all times. If your primary agency has a work gap and they don't have

any internal projects for you to work on, you need to have a way to get small side projects you can start and finish quickly to keep your income stream active. I recommend building some relationships with salespeople from previous projects. These are typically client partners who handle initial client acquisition and write SoWs and whatnot. Often agencies and design firms will turn down smaller projects because they aren't lucrative enough. Still, maybe they're perfect for a solo side project for us, so if we can offer our services on the side for them as a simple referral opportunity, we can potentially both win.

To summarize this chapter, **we want to differentiate our career from our career growth potential**. If you love a particular company and they have a growth model with a solid career ladder, and you can work your way up and retire with a pension, that may be the perfect career path for you. I just want you to remember that your position at a company is never set in stone.

Layoffs, downsizing, and cancellations are genuine risks, no matter how much time you invest with your employer or how high on the ladder you climb. My philosophy is that we are in control of our destiny, and nobody should control the fate of our career but us. If we put our career and livelihood in the hands of our upper management and executive team, we're setting ourselves up for failure because they will never care about our growth and success as much as we will.

CHAPTER FORTY-NINE

Sales

Humbly speaking, I'm no Zig Ziglar or Grant Cardone. I'm not too fond of sales, and I regret that this book needs a chapter on it, but I would be doing you a disservice by omitting it. Teaching sales or learning sales commonly involves memorizing tactics, tricks, and pitches, which never sat well with me. I've been scammed a couple of times by fast-talking salespeople, and I've made conscious efforts to steer my character away from that profile when selling my products or services.

I believe that there's an ethical and practical approach to sales that we can take. If we sift through all the shady stuff, we're left with some core principles that can be applied to almost any sales scenario. We can walk away with our integrity and core values intact--and we've already learned some of the foundational elements, so this shouldn't take you too far out of your comfort zone.

When you're finished with this book, I highly recommend reading "The Win Without Pitching Manifesto*" by Blair Enns.

* <https://www.winwithoutpitching.com/the-manifesto/>

Why Sales

Before we get into the weeds, why do we have to talk about sales at all? We're software professionals, not salespeople, right? Well, yes and no. If you're a staff engineer at a company, you're not necessarily selling your company's product to its market. Still, you will likely be selling your syntax-improvement recommendations to your colleagues and your process-improvement suggestions to the project managers.

Once you start working for yourself as a contractor or even if you start your own business, you'll be selling yourself or your company's services to your prospects. Ultimately, the process of getting someone to trust you and accept that the value you can deliver far exceeds their contribution cost is virtually the same regardless of the transaction.

Build Rapport

So, we already covered some general approaches for pre-established interpersonal relationships in the "Communication Mastery" chapter, but what about building that initial rapport with someone we don't know? Maybe we're reaching out to a lead on LinkedIn, or maybe we have a list of prospects who signed up for marketing emails and calls. I'm not too fond of cold-calling leads or receiving cold calls from salespeople. They're disruptive and put people on the spot, and there's a very unfriendly and off-putting vibe with them, so I don't do them anymore.

I lean towards a different approach to establishing an introduction with an unknown prospect. My goal is to **build relationships, not leads**. I typically start by discovering what skills or knowledge the other person has that I don't; then, I'll temporarily put them in a mentorship role and ask *them* for help with something that I need. While that rapport is being built, I'll passively demonstrate my own value, and if there happens to be alignment on collaborative needs, great. If not, no worries.

Remember, **offering to help someone does not build rapport nearly as well as asking someone for help**. Getting someone to help us allows them to feel fulfilled and justified in their career. I'm not suggesting using people for our own gains here. I'm suggesting building genuine, actual relationships with people--starting by **displaying vulnerability instead of arrogance**. Offering to help someone before rapport is built can make them feel inferior or invalidated. It can imply that we know more than them and that they need us, but really how could we possibly know that if we've never met before?

Consider this cold contact:

"Hi Sarah, is your website underperforming? I can help look under the hood to detect bottlenecks and offer some refactor suggestions to improve your site load times by up to 300%! When is a good time next week for us to have a quick call?"

My gut tells me there is a 0.01% chance that she would accept my

proposal to chat, let alone reply at all. Several things are wrong with that message, and we're making blind assumptions, which is rarely a good idea.

Let's try another approach:

"Hi, Sarah! I saw on your LinkedIn profile that you specialize in Salesforce. I'm stuck on something stupid, and I think I'm missing an obvious detail. I usually integrate Salesforce APIs on websites I build, but I never configure stuff in the UI, so I'm a bit out of my element. Google isn't helping, so I thought I'd try pinging a LinkedIn contact. If you think you might be able to help point me in the right direction, I would be forever grateful! Let me know, and I'll elaborate on my blocker. Thanks!"

This message is a 180-degree approach from the first one. I'm not wasting Sarah's time here or assuming she has a problem I can fix. I'm giving Sarah the opportunity to feel important and helpful. Most people really do enjoy feeling like they're helping others around them. I know I do. Sometimes in our careers, we can feel like we work so hard, but nobody appreciates what we do or how hard we work.

If Sarah can feel appreciated in a five-minute chat, what a win for both of us, right? Obviously, we don't want to lie about our Salesforce problem if we don't actually have one, so swap out the variables with something that makes sense in your own life. There are always tools or skills upon which we could significantly improve. If you're struggling to think of one, search farther away from your stack and your career. It could be as simple as reaching out to a LinkedIn contact from another country or culture and asking them for some insight into their culture so you can be more considerate and accommodating to new colleagues from that culture in the future.

Imagine a message like this:

"Hi, Meera! My name is Sean, and I'm going to be leading a team of engineers from India for the first time, and I'm pinging some of my Indian LinkedIn contacts because I'm hoping to become more educated on your culture. I want to be respectful and accommodating to my new colleagues, but here in the US, we can be ignorant sometimes, and I'd like to be better. Would you mind sharing a few tips with me? Maybe some common cultural differences in our countries' work styles or something I should be aware of so I'm not putting unrealistic expectations or stress

on the new team? Thanks in advance--I really do appreciate the help!"

Cross-culture collaboration is an instrumental skill, and it's an ever-evolving skill as we collaborate with more and more cultures. For example, teams from India, Poland, and Mexico are all very different. Teams from California or New York differ from those in Texas or Colorado. Culture is often isolated to the environment and upbringing.

People tend to emulate the people around them naturally. The way the world is fragmented by different governments regulating different geographical regions in different ways, we will see significant gaps across various cultures. When we start working with these people, it's crucial that we respect their cultures and not assume every team member thinks or behaves exactly as we do.

Sending a message like my example to Meera shows vulnerability on my part as a lack of expertise in the Indian cultural upbringing. Still, it passively demonstrates my expertise in the industry space since I mentioned I'm leading the team. It just plants a seed. I'm not pinging Meera to offer her engineering consulting. I'm just creating a brief profile for myself so that when she thinks of me, she thinks of a "culturally considerate engineering leader" and not a "salesperson."

Another scenario could be we want to start making websites for authors to help them market and sell their books. If we have any interest in ever writing a book, we could ask them something like this:

"Hi, Frank - I was hoping to ask you a quick question since you're a well-established author. Would you recommend that a first-time author self-publish or try to get a major publisher? I'm a career web developer, and I'm ready to write an instructional manuscript on how to build websites, but I don't know where to start. Google is telling me twenty different ways, and I'd prefer to model after someone successful like yourself. Any tips are appreciated, sir. Thanks so much for your time!"

You can see the theme here. **Show vulnerability and humility in their space, demonstrate passive value in our space**, and make them feel like we need them. If the prospect is in the engineering space, we can get a little nerdier with the dialog and a little more specific with our request, like asking for their opinion on a particular tool or cloud service:

* * *

"Hi, Johanna! I saw you have the skill GCS on your LinkedIn profile. Quick question: Would you recommend GCS over AWS, or are they pretty much the same? I have a new client who's currently on GCS, and they're not sure if they want to stay with Google or switch to Amazon when we do the v2 rebuild of their product. I've only ever worked on AWS, so I wanted to get a real-life opinion, and you seem to know your stuff. Any advice is appreciated; thanks!"

This example conveys that Johanna and I are probably equally capable of building stuff and deploying to a cloud provider just fine. Still, I'm showing that she can help me because she's a real person with a real opinion who's really worked with the tool I have to consider. That's valuable, even if I'm not asking her to help me solve a problem.

Underlying Motivation

Let's imagine now that we've established rapport with the people in our network, and now we're in a position where we see that they are struggling with something, or we wholeheartedly believe that they could benefit from what we're offering. How do we get to that level of engagement?

The "Communication Mastery" chapter discussed peeling back the onion layers to discover the underlying motivation. In sales, this is a critical prerequisite before we even consider closing. We can't sell something to someone until we truly understand what *they* perceive as valuable. Imagine trying to sell a bike to three different people:

- **Julie** needs a bike to participate in her local triathlon. She's already a skilled swimmer and runner and wants to win.
- **Brad** needs to lose some weight, and his doctor recommended starting casual exercise by biking around the park daily.
- **Chris** is having a midlife crisis and wants to resume a childhood hobby of BMX biking.

A single sales "pitch" won't work on all three of these prospects. If all three of them walk in the door to our bike shop and we know nothing about why they came in, how are we supposed to connect with them in any meaningful way to close the sale? We have to ask and ask and ask some more. It's not prying if it's a genuine conversation where we show interest and curiosity with the ultimate goal of helping them find fulfillment. Our highest profit margin bike on the showroom floor will not be both a BMX bike and a triathlon bike, just like we can't write a one-size-fits-all software application.

Our goal can not be to close the sale. Our goal must be to help the prospect get what they want.

Someday Isle

Consider this famous analogy about two islands; Island A and Island B:

- **Island A** is barren; no food, no water, and no way off the island. Our prospect is stuck on Island A and is lonely, hungry, thirsty, and depressed.
- **Island B** is a tropical paradise; with plenty of food, water, entertainment, and the prospect's entire family. Our prospect wants to believe that Island B exists but has never seen it and has no idea how to start looking for it, let alone how to get there. It's just a dream at this point.

Our goal is to convince the prospect that we can deliver Island B. If we're trying to sell a boat to the prospect, we need to very clearly understand that **the prospect does not care what kind of wood the boat is made of**, if the boat comes with intricate carvings, or if the mode of transportation is even a boat at all. We need to stop trying to sell the boat because it's simply a means to an end. Instead, we need to sell Island B.

When we're talking to the prospect and learn that the prospect is lonely, hungry, thirsty, and depressed, we need to realize that a boat literally solves none of those things, and Island B solves all those things. We need to change our perspective and frame the conversation so that we can promise that if the prospect trusts us, we can replace hunger and thirst with satiation, loneliness with companionship, and so on.

If we replace some variables of this story, imagine our prospect is someone who happens to have an excellent idea for a mobile app. Island A is the idea stuck in the prospect's head. It can't get out because the prospect is not a technologist. Island B is a deployed, popular app earning lots of money, but our prospect knows nothing about that aside from rumors that people can get lucky and make a lot of money with mobile apps.

* * *

So if we remember not to focus on the boat, then we can infer here that our prospect does not care if the app is built with Objective C, Java, Kotlin, React Native, Cordova, Expo, Unity or Swift, or even if it's a native app or a web app. None of these details matter because any of these tools can build a production-worthy app, and none make any sense to a non-technologist. At best, the prospect has a smartphone and wants to use their own app on their phone, but they'll likely say things like, "I want the app to work on all phones." We can follow up with "Understood. We'll likely roll the app out in phases to cover the most popular phones first, and then we can add support for less popular phones while the app is making money."

Once we understand this, our conversation with this prospect revolves around getting them excited about the reality of the product. We want them to genuinely believe that the idea doesn't have to be stuck in their head and that we can make it real for them. If we haven't already demonstrated value, we can show some apps we've deployed to the app store and convey, "You're in good hands; we've done this before; let's focus on your app now." Our prospect will likely want to know income potential and profit margins. "How much money do you think I can make with this app?" is a very real question the prospect may ask. Suppose we have some objective sales numbers from previous projects or even publicly-available app store metrics. In that case, we can give some ballpark estimates and even form a game plan for targeted marketing and upsell strategy.

Whether the app should be a one-time fee, a subscription-based model, freemium, and so on are essentially customizing the Island B experience for our prospect. We can answer those questions in the sales call, but our responses should provide options, not answers. We would want to say something like, "We can release different versions of the app under different pricing models and see which one performs the best and then use that for the long term." We want to stay out of the weeds and avoid mentioning *how* we plan to do that--like mentioning which design patterns can accomplish this or which tools we may want to use, for example.

We want to avoid talking ourselves up about how great we are at

writing code or how organized our database is, or how modular our UI components are. We can summarize that by saying, "We'll follow best practices to ensure your app is fast and stable so your customers can stay happy from day one." That's all we need. Ideally, if we've built a strong enough rapport from our initial relationship, the prospect will already see us as industry experts and trust our judgment.

Imagine your neighbors got a beautiful new pool installed. It's got everything you've ever wanted in a pool, and now you're inspired to add a pool to your own backyard. You ask your neighbor for the pool builders' contact information and set up a meeting to discuss your dream pool. In this call, you don't really care to hear their sales pitch about how long they've been building pools, how they lay the plumbing down, or which brand of pumps they use, right? You know they build great pools because you witnessed and experienced it firsthand. Your neighbor has a fantastic pool, and that's all you need to know. At this point, you want to talk about your *own* pool and the experience you want in your backyard. **You want to talk about Island B.**

If the pool company ignores your enthusiasm and keeps returning to a canned sales pitch, it's potentially going to rub you the wrong way. You may start thinking, "They build great pools, but several other companies also build them. They aren't listening to me right now, so how can I be certain they'll listen to me moving forward to ensure I get the pool I want? How can I be sure they won't just build a pool *they* like or a replica of my neighbor's pool?"

Back to our role as the seller: when our prospect is talking to us about the product they want, what they're really telling us is the **experience** they want, the **response** they want, and the **feeling** they want. They're not telling us the technology or the process they want. Almost every time, they just don't care. I firmly believe the key to sales is to listen to the prospect, develop a deep understanding to find out what they genuinely want, try to understand *why* they want the thing, and then if we fully believe that our solution can deliver what they want, communicate that message to them by **proving the value we can deliver far outweighs the cost of our services.**

Cost vs. Value

The last concept I want to cover in sales is cost versus value. Almost inevitably, we will get to a place where we think we're fully aligned on what the prospect wants, and we know we can deliver it, and then the prospect asks, "So how much is this going to cost?"

This is often the most challenging question to answer. I think the reason is that typically in sales, there's a markup on the item, and sometimes we don't intrinsically believe that the item is worth the sticker price. We're just *hoping* the prospect will pay the sticker price because it positively affects our profit margin or commission, and if the prospect responds with hesitation or a lower price, it becomes a whole thing. Again, if our goal is the sale, then we will lose the sale. Our goal should be to ultimately convince the prospect that the value far exceeds the cost. Double, at least.

For example, I took my daughter shopping this weekend, and we were at a boutique at the mall. She picked out these silly socks with funny quotes on the ankles; each pair was \$18. These socks were low quality and probably cost \$ 1-2 to manufacture. Including shipping and packaging, the store probably could have made a hefty profit with an \$8 price tag. **In my mind, the cost far outweighed the value.** My daughter wasn't thinking about any of that stuff. She was thinking about the reactions from all her friends when she showed off her socks to them. To her, \$18 wasn't paying for the materials and labor associated with the socks. The \$18 was paying for the attention and laughs she'd get at school, so **in her mind, the value far outweighed the cost.**

We need to understand this when we start talking about pricing with our prospects because the only way they will ever reject our offer is if we haven't associated enough value to justify the cost. I can't emphasize this enough--**the value has to come from all the effects derived from the end deliverable, not from the deliverable itself.** This might be an emotional response, or it might be a monetary ROI. Again, each prospect will associate value with different things. In reality, some projects will never offer enough value to the prospect to justify the cost,

and that's ok. We'll cover that more in a bit, but our goals don't change either way.

Consider expensive cars. Why do people spend \$200,000 on a Lamborghini? My \$10,000 Yamaha R1 can go 0-60 in 1.9 seconds on the Dyno, and I've hit 194 MPH on an empty late-night freeway before I had to exit. Lamborghinis are fast but aren't twenty times faster than my street bike. People spend \$200,000 for the attention that comes with owning a Lamborghini: the constant head turns from pedestrians and other drivers on the road pointing at the car and taking photos, wishing they owned one themselves. My bike could never compete with that experience.

Owning a Lamborghini is a lifestyle milestone. It's a symbol of achievement and wealth for a lot of people. Aftermarket body kits for import tuner cars like you might see in the Fast and Furious franchise look just as slick, and several of those cars can go just as fast as a Lambo. Still, the Lamborghini brand holds so much credibility that the company can get away with the steep price because the value outweighs the cost.

Bringing this back to software sales, if we recall the bicycle analogy, **the value will be different for different prospects**. This is why we can't try to associate value with our products or services. Our products and services are valuable to **us** because they help us get subsequent work, and we can feel accomplished and have a lucrative career from them.

When our prospect asks us the cost, we tell **them the cost with confidence and shut up**. Let them process it and wait for them to respond. The first person who talks next will typically lose the negotiation. I realize this is a sales tactic but try to look at it as a communication skill more than a sales skill. I learned this trick from the movie Boiler Room (which I highly recommend) and decided to try it on my personal training prospects, and it surprised me how well it worked.

We communicate our needs to them (which, in this case, is a financial need), and we show respect by letting them respond. In their head,

they're most likely trying to work out the cost-versus-value associations. If they figure it out, we'll close the sale, and if they can't, they'll come back with a counter or some hesitation, and we can iterate on building value. We don't want to budge on our price. Our price is *our* value that meets *our* needs.

As I mentioned earlier, some prospects aren't ready for Island B, and that's ok. I've had tons of sales calls where the prospect really wanted a \$300,000 website, but their budget was \$5,000. The sales conversation was just them realizing their fun website idea wasn't worth taking a second mortgage on their home and making a hard pivot on their full-time job. They thought it would be a cheap side project that could earn passive income magically, and that's not their fault. That's just a non-technologist misunderstanding of technology. The value of their product vision can only be extended so far without tapping into manipulative and unethical sales approaches, which I won't do.

This is why I don't budge on my price. How much could I have budged for that deal to have worked? Neither of us would really have been happy about it anyway. The client would have paid exponentially more than they had been financially prepared for, and I would have earned exponentially less than I needed to feel justified in my efforts.

Most of the time, I lean on salespeople to do sales stuff as much as possible because I value my time and mental energy. I'd rather direct those resources to architecting systems and developing top-tier software; if you're more of a people person and love the process of closing deals, more power to you. I genuinely admire the skill so long as it remains ethical and nobody gets scammed. That said, the foundation I outlined in this chapter has proven invaluable in my career. The concepts work across so many different scenarios it sometimes feels like I unlocked a cheat code to the human brain. I can't take credit for discovering these principles, of course. I learned by reading and listening to some fantastic mentors in the sales world. At this point, I encourage you to read some material by Zig Ziglar, Brian Tracy, Grant Cardone, or any of the thousands of industry leaders so you can add this skill to your toolbelt and keep it in the 20% that earns you 80%.

CHAPTER FIFTY

Networking

Part of maintaining your personal brand and expanding your influence is going to involve networking. We have several platforms and approaches at our disposal, each with its own pros and cons. We must recognize these early on so we don't waste our time or attract the wrong kind of attention. This will be a short chapter; these platforms come and go, and your mileage may vary.

Technical Networking

STACK OVERFLOW

StackOverflow.com is my favorite website for building a reputation for competence in our field. It's clear as day how we approach and solve problems. People can see how quickly we can accurately respond to a question and how we word our feedback to the original poster (OP). There's something about the vulnerability of putting ourselves out there with the sole purpose of helping another dev. We may get the answer wrong or offer an answer overshadowed by another.

I've improved immeasurably on StackOverflow when I've suggested a solution I thought was ideal, only to be schooled by another user who offered a substantially better answer than mine. These times, I'll humbly delete my answer and remember the better answer moving forward in my career--either when I encounter the issue in my work or when I see a similar question asked on the website. How we improve is irrelevant.

The website can't be used to network directly. There's no option to connect to other users or directly message them. I use it as a portfolio of competence. I add my profile link to my resume, and potential clients or employers can quickly verify that I've proven myself in public. It reduces the need for a complex technical challenge during the interview process. I've had interviewers tell me they saw my StackOverflow profile, and it was all they needed to pick me among other candidates for the interview.

As a bonus, it helps us improve as engineers. Not that the questions on the website are all that challenging, but more so that we can train ourselves to quickly problem solve. Answering questions on StackOverflow is a race. If our answer is picked, we get the points and the recognition. If someone beats us to the answer, it's all the more reason to try harder next time.

This skill comes in handy during our workday when junior devs ask us

for help with something. If we're well practiced in immediate problem solving, we will naturally problem solve quickly at work too. Create an account if you don't have one already, and start answering as many questions as possible in your typical stack. Gain some points and a reputable score, and add your profile link to your resume or social profile. It's a great resource to have your skills speak for themselves.

GITHUB

Github.com is another website I look for when considering candidates for a position. I first see if they help others on StackOverflow, and then I see what their typical code commits look like on GitHub. Some employers want devs who contribute to multiple open-source repos or showcase multiple side projects. I never hold a lack of that against a candidate. I understand sometimes our entire GitHub portfolio is locked behind private organizations or in private repos. If I can see some source code, great, but if not, no worries.

Where StackOverflow may show a snippet or two of syntax, GitHub typically offers a more accurate representation of the type of code I'll see if I bring the dev on my project. I can get an idea of what patterns the dev prefers. I can get a feel for how sloppy and buggy the dev allows their code to be when committing it to the `master` branch. This is important because I don't want to hire a dev who claims to adhere to strict conventions while their GitHub profile says otherwise.

I recommend including your GitHub profile for networking purposes, regardless if you have something cool to show off. So many projects use GitHub, and you can establish yourself as a team player before your interview by having an account ready. If you have some solid portfolio work, pin your top repos for easy reference. I recommend not to pin a forked repo unless it's a well-known project and you're a major contributor. Otherwise, it just litters your profile with other people's work.

CODEWARS

* * *

CodeWars.com is a nice hybrid between StackOverflow and GitHub. It's an excellent website for practicing canned challenges regardless, and if you earn badges worth showing off, it makes sense to add a link to your resume or social profile. I recommend waiting until you feel like a badass on the website before sharing your link unless you're sharing it for fun among your peers as a way to challenge each other in some friendly competition. If your goal is to earn respect from a potential employer or client, consider how more or fewer badges can make you look, similar to sharing your StackOverflow profile if you only have a few hundred points.

CodeWars is a double-edged sword. I've only used it briefly when trying it out as an option to help my son improve his coding skills, so my profile hardly reflects my career or actual skill set. If I put a link to it on my resume or my LinkedIn page, it would hurt me more than it would help. People would be confused because my LinkedIn says I've been coding for over 25 years, but my CodeWars profile makes it appear like I only know basic JavaScript.

You can see how a profile full of badges in all the languages that convey a wide range of syntax mastery could be an exceptional asset on a resume and how the opposite would also be true. If I'm reviewing a candidate and see a CodeWars link, I'll visit briefly and be impressed if it's impressive, and if I'm not, I'll move along to the next resource in the candidate's resume. I understand not everybody has hours of free time to complete online coding challenges, but don't assume every employer or interviewer thinks like me.

Blogging

MEDIUM

The only blogging platform I'll cover in this chapter is Medium.com. You could make a WordPress or Blogger website, but Medium is the defacto hub for thought dumps, so I recommend starting there and only migrating if it's not working for you. I find there are two distinct types of articles on Medium from authors in the engineering space: subjective op-ed and objective instructional.

I recommend writing several instructional articles to make a name for yourself as a reputable resource before authoring any op-ed piece. Building credibility and establishing a reader base before offering an opinionated article can help ensure that the initial batch of reader responses is positive. They may not all agree with you, but if they are a fan of your work and you've helped them in the past, they'll likely be more polite in their commentary. This is in contrast to anonymous or random troll commentary.

People typically only leave negative comments when they're not afraid of being held accountable. I call them drive-by haters. Suppose you don't yet have a strong following of readers. In that case, you'll not only risk negative comments overshadowing positive ones but also be in a position where you'll have to moderate or defend your position. Once you get a fan base, your fans will defend your article on your behalf so that you can focus on writing and programming.

I've tried blogging on occasion over the years and never enjoyed it like I thought I would. I wrote a couple of op-eds early on, and nobody knew who I was, so I was just some guy with an opinion, and I didn't get any fulfillment from the time spent blogging, so I spent my time elsewhere. Ironically, I abandoned one blog post about how to succeed as a career software professional. It got rather long and needed some structure, so I wrote this book instead. Go big or go home, right?

The articles I typically read and enjoy on Medium are primarily instructional--quick how-to articles for everyday things, like getting a plug-in to work or a brief overview of a SAAS tool. In our work, it's handy to get a cliff-notes introduction to a tool instead of visiting the tool's documentation and sifting through all the sales clutter. As it happens, Google often puts Medium articles at the top of my search results, above official docs, so I'm not the only one who appreciates the format.

Regarding networking, I've met a handful of very talented engineers on Medium--their articles impressed me, so I reached out to connect with them. Again, it's essential to establish credibility with instructional articles before writing an op-ed. Otherwise, you risk coming off as an ignorant noob with baseless opinions. One author has an article from September 2015 titled "Becoming a software engineer is hard" and mentions she is starting Hack Reactor (coding boot camp) the following month. She has another article from June 2016--**eight months later**--titled "Coding Is Over" and drops gems like this:

Coding is super dumb

Coding allows for typos. More than that, it allows for infinite "creativity", also known as code smells. Most code is pretty rank. Engineers spend countless hours dealing with syntax, typos, indentation, linting, errors, arguing over style and best practices, and making shortcuts to try to coax some of the code to type itself. It's absurd. And it's a waste of time.

This person might as well be a teenager, acting like she has everything figured out after only eight months on the job. I'm not exaggerating when I tell you **Junior devs who act like they know everything about everything are almost always the ones who add all the bugs to all the code**. The sad part is I agree with her assessment. She's not wrong. Spending countless hours on syntax *is* a waste of time, but she's failing to understand that this is not new. Tools like Microsoft Frontpage and Macromedia Dreamweaver tried to solve this with abstraction layers between the developer and the code for years since the late 1990s, and the results were horrid. Lower-level IDEs like Microsoft Visual Studio for Visual Basic can apply a more modular approach to coding. Unity allows code to be dragged onto the scene and into an artifact. Apple Xcode allows dragging behaviors and scripts to objects when coding

Swift projects. In reality, someone still has to code these tools. Someone still has to code these behaviors and scripts. Regardless if she has one point, she's coming off as a know-it-all, and that only works if you actually know it all.

Be humble. Be honest. Be vulnerable. It's ok not to know things. It's ok to write an article and describe a helpful feature in a library that might be buried in the docs. It's ok to write an op-ed about your personal experience with something. Instead of saying, "Coding is super dumb," consider saying, "Coding is racking my brain." It's the difference between making a bold claim in the absence of experience vs. telling a story with opinions of your own experience.

Hackathons & Meetups

Depending on where you live or work, you may be invited to participate in a hackathon or an open meetup. These are typically different experiences with slightly different objectives.

MEETUP

Meetups are like happy hours. Sometimes they're even called "mixers." When I worked in San Francisco, tech companies had meetups all the time as a way to extend an open invitation to people in the tech space to come to hang out, see what they do, meet some folks, and have a nice evening.

I've attended several meetups as a representative of my employer at the time. A few of us from the office would attend and wear our company shirts and name tags. We would pass out our business cards throughout the evening, collect resumes and contacts, and then the next day at the office, we'd go through our new leads like Halloween candy after a night of Trick-or-Treating.

Several contacts we'd meet at the event would be salespeople trying to establish B2B connections. Usually, we'd sift through those (we don't need a personal contact at CircleCI, for example) and go through the engineers and recap the evening to decide if any of them were someone we'd want to work with daily. Personality goes a long way in an office setting--remote work, not as much. If any of the engineers were interested in considering work options, we'd typically set up an in-person interview that week and get the ball rolling.

What this means for your networking efforts is that if you attend meetups, you'll likely meet some scouts from potential employers who need resources for a project. Your goal is to be a fun version of your professional self. Don't get blackout drunk and make a fool of yourself; you'll become that person everybody talks about for the rest of their career, and you could sabotage your employment potential. Also, don't

be a quiet mouse in the corner. You want to be remembered the next day by as many people as possible for all the reasons *you* decide.

The meetup is just as important an opportunity for us to interview potential employers. It's no fun spending all day in interviews and getting a job somewhere, only to realize several days into your position that the culture is not a fit. You can use the meetup as a first impression gauge to decide if you even want to pursue a career at a particular company, solely based on how they represent themselves in a setting when their guard is down.

HACKATHON

Hackathons are not the same as meetups. Sometimes they're called "codefests." The purpose of the hackathon is usually to solve a company's particular challenge or reach an open-source milestone of some kind. The host of the hackathon may or may not even serve food or drinks. I've attended hackathons with a full buffet and a \$15,000 prize, and I've attended a hackathon in a dimly-lit cellar with a "contributor credit" incentive, and we weren't even sure if we were in the right place.

While I've attended a few of these, I've never participated in them--I always accompanied a colleague who wanted to go. Hackathons can be a way to get the crowd to help solve an internal challenge because the company or org couldn't do it themselves. I know this because I worked at a company that didn't have the resources to write a complex adapter for a third-party service to be used unconventionally, and the CTO at the time said, "Let's host a hackathon next month and pay some kid a few grand to build this for us."

Sometimes security companies will host hackathons to encourage ambitious engineers to exploit their upcoming release. The company will host food and a nice evening; in return, the devs try to gain access and find loopholes. The company benefits through cheap beta testing and can itemize patches for the loopholes exposed that evening. The cost of paying the first prize is negligible compared to losing millions of

dollars through a production data leak.

From the company's perspective, hackathons can be the perfect solution. They can remove the risk and overhead of scouting and onboarding a contractor or employee that can't deliver in a timely fashion, and the "Wisdom of the crowd" theory has prevailed over individual expertise throughout history.

One hackathon I attended with a colleague turned out to be a last-ditch effort to get help finishing a library advertised to release that month, and the founders were weeks behind schedule. The group basically wanted free labor and offered beer tickets for the bar next door. I understand that situation, but full disclosure would have been better than masking a "Help, open source friends!" evening as a "Come to our hackathon!" event, especially when there's no hacking involved.

If you attend a hackathon, go to have fun and see what it's like. Keep your expectations low and your eyes open to avoid getting exploited, and if you go with a friend or colleague, you'll enjoy the evening either way.

Virtual Resumes

Hired

Of all the resume websites online, I found Hired.com to have the best overall engagement from potential employers. Like all resume websites, there's no magic way to get a resume loaded into the experience without having to go through each job and project and touch up little details, so I recommend doing it manually and taking the opportunity to polish the details and reword some things to align with current trends and market direction.

My favorite thing about Hired is that they really did get my resume in front of hiring managers within hours. Every time I've "activated" my resume on Hired (vs. deactivating it when I'm employed, so I don't advertise that I'm looking for work), they show me job openings within my desired pay range that I can click on to convey interest. I've had three to five daily calls from hiring managers and recruiters telling me about positions and wanting me to come into their office to interview immediately. It's very active in that sense.

My least favorite thing about Hired is, ironically, that same experience. Several times I've spoken with prospective employers and gone through interviews only to learn their budget for the role in question is nowhere near the advertised compensation on Hired. As I'm writing this chapter, I wish I had the foresight to ask why this was the case, but instead, I politely declined those offers and moved on. I don't know if the employers were lying or were uncoordinated with their budgets and HR or if Hired.com had a bug. Regardless, while some employers were off by several thousands of dollars in their advertised compensation, some were spot on, and I found projects aligned with my desired income.

I recommend using Hired.com for short-term job search efforts. This website is excellent if you get laid off and need a job quickly. For long-term career growth, I don't see much value.

LINKEDIN

LinkedIn fills the gap for long-term career networking and growth efforts. I keep in touch with several of my former colleagues on LinkedIn. Seeing what they're working on as the years go by can be a deciding factor if I want to reach out to them for a future project. This goes back to the saying, "It's not what you know; it's who you know."

I aim to have a respectable and professional presence on my LinkedIn profile, and I've gone out of my way to get recommendations from colleagues when I transition off projects. This is important in the same way people tend to believe customer product reviews over manufacturer claims. I could toot my own horn on my profile bio, but it'll never hold as much weight as a recommendation from a colleague.

Consider these newest recommendations on my profile:

* * *

Sean Cannon



Kaushik Acharya · 1st

Senior Director Of Engineering & Product Development at Xolv
July 7, 2022, Kaushik was Sean's client

I worked with Sean on an extensive project porting a legacy java application running in a stand-alone data center to a modern, redesigned Node / React application running on AWS.

I was impressed by the depth of Sean's knowledge and his ability to justify architectural decisions. He is equally comfortable evaluating and adjusting design decisions from his peers, or designing from the ground up. At Business Wire, he provided us with best practice conventions for both Node / React and AWS development. Based on his guidance, we adopted and adhered to several software design patterns that proved highly beneficial not only in later use, but even during the initial build.

Sean is easy to work with and does not stand on superiority, despite being at a senior lead / architect level. He is highly productive, approachable, and a pleasure to be around.



Ben Luu · 1st

Principal Software Engineer at Business Wire
June 30, 2022, Ben worked with Sean but they were at different companies

I had the honor of working closely with Sean. He came in to mentor and guide our team towards a more modern technology stack. He has vast domain knowledge and always ensured that the best practices and patterns were always at the forefront. Cutting corners were not allowed and principles were enforced to warrant that velocity was spent in developing new ideas and not applying band-aids.

Working alongside Sean for the last few years has shown me how unique he is. Sean is capable of being a one man shop and drill down on any issues by himself, or he can lead a team towards a common goal. He can communicate up or down within a company to ensure everybody is on the same page. I have the upmost respect for Sean and enjoyed being his colleague.



Gregg Giles · 1st

Growth Product Management @ Disney Streaming | Disney Media and Entertainment Distribution
June 22, 2022, Gregg was Sean's client

Sean is an exceptional engineering leader and collaborator, absolutely one of the best I've ever worked with. As a Product leader, my job is to dream up the next big thing to tackle, and Sean was always right there helping figure out how we'd turn that germ of an idea into meaningful solutions that make lives better for customers. During discovery, Sean left no stone unturned, and would surface options that I didn't even realize were possible, which is a fantastic position to be in, which opened avenues for making the product even better than I'd hoped for. His communication is top-notch and he's always ready to explain even the most complex issue in a way that's tailored for the recipient to easily grasp.

I can't sing enough praises about Sean, he's an absolute pleasure to work with and an indispensable asset to anyone that wants a pragmatic and realistic partner that will help you get the job done right and with zero drama. Just an all-around great partner!

Colleagues in different roles will have different recollections of how we worked together, so their reviews offer different perspectives. Depending on who's looking at my LinkedIn profile, one of these reviews may resonate more than another. I can focus on my work and deliver the most value possible in all areas, and then my results become self-evident.

The way to get recommendations on LinkedIn is to give them. As of this writing, I've given 44 and received 37. Imagine if I had only given 10, or 5, or none. What we put out in the Universe comes back to us. Unlike building rapport, where we would want to ask someone to do a favor for us, reviews like this are often an afterthought. A missing review on LinkedIn does not block us, and if we're transitioning off a project, helping us may be a low priority to them. Writing a positive recommendation for someone out of the blue is a nice gesture. If they

happen to write us one as well, great, but if not, it's okay.

LinkedIn also has a social media component, but proceed with caution. I recommend reserving this tool for self-promotion by demonstrating competency. Showcase your work, recognize others, and congratulate everybody in your circle when they achieve something. If you are tempted to offer an opinion in a post or comment, remember that opinions are invitations to conflicting opinions. Even the friendliest debates can cause division among peers. *Professional* opinions on something that has worked well for you, like an endorsement, is typically a safe avenue. On a platform meant to keep as many doors and opportunities open as possible for us and our careers, *personal* opinions on social topics should be saved for another platform.

Social Networks

I have a shared approach to all the major social media platforms--Facebook, Twitter, Reddit, Instagram, TikTok, Snapchat, etc.--I firmly believe it's in our best interest to draw a hard line between our personal and professional lives. These days, employers often think they have the right to browse our social media accounts and snoop on our personal lives, and it's frankly none of their business.

First and foremost, I recommend never friending or following colleagues on these platforms while we work with them. If they or we transition off a project, I may accept a friend request if I have built a close relationship with a colleague. If that happens, I add them to a restricted list to keep in touch with them in a filtered setting. I don't want to have to filter myself on my personal social media account, and I don't want to risk something personal being seen by a "friend of a friend who is not my friend."

Friends go in my main list, and acquaintances are restricted. Some people put their entire lives on social media for the whole world to consume. I respect their choice to do so but knowing what I know about phishing, catfishing, cyber security threats, etc., I don't recommend being so open about everything. When someone posts, "I bet you can't name your second-grade teacher!" the first thing that comes to mind is, "That's a security question."

I'm not suggesting that you should be paranoid or become a recluse. Just understand how easy it is to forget that you have a colleague in your list of friends when you decide to rant about your work day or ask if anybody is hiring. Imagine if your colleague sees that, mentions it to another colleague, and word gets to your employer. They decide to start looking for your replacement, assuming you'll be resigning soon, and the next thing you know, you're replaced. All because you needed to vent on your personal social media account, and that information was never meant for your boss's ears. **Always assume everything you post online will make its way around to the people you least want it to.**

For example, I have multiple Twitter accounts:

- I have a professional account where I tweet about technology or work things, and not very often.
- I have a personal account where I tweet primarily comments to other people's tweets that have nothing to do with much. Typical twitter nonsense.
- I have a jiu-jitsu account where the only people I follow are fellow BJJ athletes, and I'll share photos and clips from training sessions.

Separating these accounts helps me isolate my networks. I don't want my BJJ training partners to know much about my personal life, and I don't want to know about theirs. I don't want to know their political stance, whether or not they supported the COVID-19 vaccines, or any other divisive identity politics. I want to go to class, focus 100% on my training, and then return to my personal life afterward. Similarly, I want to go to work, focus 100% on building top-tier systems with competent colleagues, and then return to my personal life when work ends.

If an active colleague asks to follow me on Facebook, I'll politely tell them, "I'm more active on LinkedIn. I'll send you an invite." and leave it at that. **If you're serious about your career, you need to be serious about this boundary.** Colleagues can eventually become friends, but it's a slippery slope to establish close friendships with colleagues while working with them. It can affect your job performance and judgment, and you may find yourself in a leadership position over them, where they may want special treatment. You'll be a great leader if you have boundaries set from day one.

CHAPTER FIFTY-ONE

Fulfilling Your Dream

Well, we made it. My goal with this book was to offer you tools, perspectives, guidance, and frames of reference; advice for paths to consider and paths to avoid; wins and pitfalls to recall the next time you're in a situation that can benefit from it. I've poured over 25 years of experience into this book. While I realize you, as the reader, will have your own path, goals, dreams, and preferences, I believe applying what you've learned in the last fifty chapters will set you far above your peers so you can more easily get to where you want to be in your career.

That brings us to the topic of this final chapter: where do you want to be?

Earlier in the "Goals Mastery" chapter, we discussed ways to define and track measurable goals. Fulfilling your dream is the long game of that process. How will you know when you've "made it"? How will you know when you've fulfilled your dream?

Regardless of the actual dream, the key word in that question is "fulfilled." The question then becomes, "How can we identify fulfillment?" We can certainly identify when we are *not* feeling fulfilled, right? We can feel stressed, depressed, detached, and unmotivated. Working exceptionally hard toward something we don't want to do manifests as stress. Working exceptionally hard toward something we love manifests itself as passion. People who are passionate about

something never seem to get tired. They don't let little things bother them because there's an innate drive to stay in the feeling.

I remember as a kid when I was first learning how to play the guitar. I would spend countless hours in my room listening to Metallica songs on cassette and trying to play what I was hearing. I didn't have access to music lessons or books, and my family didn't have the Internet yet, so it was just my music, my guitar, and me. The first thing I would do in the morning before school was practice my guitar. All I could think about all day was getting home from school and practicing some more. It was passion and gave me more fulfillment than anything else in my life at the time. I didn't need video games or a girlfriend or lots of money. I was ten years old and just wanted to experience being able to play like Kirk Hammett could.

Fast forward a few years, and I discovered Ultima Online with some friends. It was a magical experience for me, and I would play for hours raiding enemy guilds, and camping spawns. This was the late 1990s, and I was notably good at playing the guitar by that time--I played Van Halen's Eruption for one school talent show and played a shred version of The Star-Spangled Banner behind my head for another. My level of fulfillment practicing alone had faded but playing live and in a circle of friends filled that void. I realized the connection with close friends where we were all learning and improving at something was the puzzle piece. It didn't matter if it was playing in a band, playing Ultima Online, or taking karate lessons. It was incredibly fulfilling to share vulnerability and recognition for accomplishments with genuine people I could trust.

Around that time, a couple of my friends sparked an interest in building websites. I was a novice and still relied on community hosts like GeoCities and Angelfire, but I could code some HTML and had experience slicing Photoshop documents for custom UI stuff. I put myself in a mentorship role in my circle of friends in this space and helped them learn as much as possible. I felt significantly more fulfilled when my friends succeeded than when I did. Helping my friend launch his Ultima Online character's website was more fulfilling than when I built and launched my entire guild's website. I didn't realize it at the

time, but teaching was my dream. Helping others overcome struggles is my underlying motivation. It explains why I wanted to teach martial arts, become a personal trainer, and stay out of management positions to remain at the top of the engineering workforce. It explains why I love rescuing and fostering animals and why I spent hundreds of hours writing this book.

I tell you this story because I want you to understand that had I not put that puzzle together in my life, I'd likely be trying to make it to the top of my career and earn as much money as possible. I did that for several years, as outlined in the "Compensation / Title Correlation" chapter, but when we unpack the details, my work is the same today as it was in 2004. I still take direction from a product owner, write code, collaborate with other teams, and aim to establish a work/life balance. I write software because it makes financial sense, not because I love it; I do not enjoy building enterprise systems to make billionaires richer. I love figuring out solutions, and I love helping others. No matter what I do in this life, that element must be there for me to feel fulfilled. Learning jiu-jitsu with a training partner offers the exact same fulfillment: figuring out why we get stuck in a position, studying and learning a counter or escape, and applying that new tool when it makes sense.

With all the knowledge I'm hoping you obtained from this book, I want to leave you with this: whatever dream you decide is for you, remember just that - **it's for you**. As long as you work for someone else, you are fulfilling *their* dream, not your own. You may feel fulfilled or content at your job, but your employer controls it and doesn't owe you anything. They can reassign you, change your work setting, or lay you off, and everything changes in a flash. The more you can control your destiny, the happier you will be. I challenge you to form a company and offer your services as a consultant instead of an employee. Don't wait until your employer downsizes. Be your own boss, rule your own life, and fulfill your own dream.

Best of luck, friend. I wish you the happiest adventure possible. When you reach your destination, remember the people behind you and try to help them as much as I've tried to help you. You may find that helping others is your calling too, and we're more alike than we realize.

Part Five

The End

Glossary

accessiBe - accessiBe is a technology company working to solve the problem of web accessibility through AI. The company has raised \$58 million in two rounds of funding.

ActionScript - an object-oriented programming language initially developed by Macromedia Inc. HyperTalk, the scripting language for HyperCard, influences it. It is now an implementation of ECMAScript, though it originally arose as a sibling, both being influenced by HyperTalk.

Active Record - In software engineering, the active record pattern is an architectural pattern. It is found in software that stores in-memory object data in relational databases. It was named by Martin Fowler in his 2003 book Patterns of Enterprise Application Architecture.

ADA - The Americans with Disabilities Act of 1990 is a civil rights law that prohibits discrimination based on disability.

Akamai - Akamai Technologies, Inc. is an American content delivery network, cybersecurity, and cloud service company providing web and Internet security services. Akamai's Intelligent Edge Platform is one of the world's largest distributed computing platforms.

Alt Text - The alt attribute is the HTML attribute used in HTML and XHTML documents to specify alternative text that is to be displayed in place of an element that cannot be rendered. The alt attribute is used for short descriptions, with longer descriptions using the longdesc attribute.

Android - Android is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets.

API - An application programming interface (API) is a way for two or more computer programs to communicate with each other. It is a software interface offering a service to other pieces of software.

AppKit - AppKit contains the objects you need to build the user interface for a macOS app. In addition to drawing windows, buttons, panels, and text fields, it handles all the event management and interaction between your app, people, and macOS.

AR - Augmented reality is an interactive experience that combines the real world and computer-generated content. The content can span multiple sensory modalities, including visual, auditory, haptic, somatosensory, and olfactory.

Assembly - In computer programming, assembly language, often referred to simply as Assembly and commonly abbreviated as ASM or asm, is any low-level programming language with a very strong correspondence between the instructions in the language and the architecture's machine code instructions.

Async/Await - The `async` function declaration declares an `async` function where the `await` keyword is permitted within the function body. The `async` and `await` keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to configure promise chains explicitly.

AutoCAD - AutoCAD is a commercial computer-aided design and drafting software application. Developed and marketed by Autodesk, AutoCAD was first released in December 1982 as a desktop app running on microcomputers with internal graphics controllers.

AWS - Amazon Web Services, Inc. is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals, companies, and governments on a metered pay-as-you-go basis. These cloud computing web services provide distributed computing processing capacity and software tools via AWS server farms.

Azure - Microsoft Azure, often referred to as Azure, is a cloud computing platform operated by Microsoft for application management via around-the-world distributed data centers.

Backlog - Within agile project management, product backlog refers to a prioritized list of functionality that a product should contain. It is sometimes referred to as a to-do list and is considered an 'artifact' within the scrum software development framework.

Bash - Bash is the GNU Project's shell--the Bourne Again SHell. This is an sh-compatible shell that incorporates useful features from the Korn shell (`ksh`) and the C shell (`csh`). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.

BDD - In software engineering, behavior-driven development is an agile software development process that encourages collaboration among developers, quality assurance experts, and customer representatives in a software project.

BigQuery - BigQuery is a fully managed, serverless data warehouse that enables scalable analysis over petabytes of data. It is a Platform as a Service that supports querying using ANSI SQL. It also has built-in machine-learning capabilities.

BitBucket - Bitbucket is a Git-based source code repository hosting service owned by Atlassian. Bitbucket offers both commercial plans and free accounts with an unlimited number of private repositories.

BJJ - Brazilian jiu-jitsu is a self-defense martial art and combat sport based on grappling, ground fighting, and submission holds.

BLE - Bluetooth Low Energy is a wireless personal area network technology designed and marketed by the Bluetooth Special Interest Group aimed at novel applications in the healthcare, fitness, beacons, security, and home entertainment industries.

Blue/Green - In blue-green deployments, two servers are maintained: a "blue" server and a "green" server. At any given time, only one server handles requests (e.g., being pointed

to by the DNS). For example, public requests may be routed to the blue server, making it the production server and the green server the staging server, which can only be accessed on a private network. Changes are installed on the non-live server, which is then tested through the private network to verify the changes work as expected. Once verified, the non-live server is swapped with the live server, effectively making the deployed changes live.

BMX - BMX, an abbreviation for bicycle motocross or bike motocross, is a cycle sport performed on BMX bikes--either in competitive BMX racing, freestyle BMX, general street, or off-road recreation.

BS - Bullshit

C# - C# is a general-purpose, high-level, multi-paradigm programming language. C# encompasses static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines.

C++ - C++ is a high-level general-purpose programming language created by Danish computer scientist Bjarne Stroustrup as an extension of the C programming language, or "C with Classes."

Cassandra - Cassandra is a free and open-source, distributed, wide-column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

CDN - A content delivery network, or content distribution network, is a geographically distributed network of proxy servers and their data centers. The goal is to provide high availability and performance by distributing the service spatially relative to end users.

Checkstyle - Checkstyle is a static code analysis tool used in software development for checking if Java source code is compliant with specified coding rules.

Chrome - Google Chrome is a cross-platform web browser developed by Google. It was first released in 2008 for Microsoft Windows, built with free software components from Apple WebKit and Mozilla Firefox. Versions were later released for Linux, macOS, iOS, and also for Android, where it is the default browser.

CI - In software engineering, continuous integration is the practice of merging all developers' working copies to a shared mainline several times a day.

CircleCI - CircleCI is a continuous integration and continuous delivery platform that can be used to implement DevOps practices.

CLI - A command-line interpreter or command-line processor uses a command-line interface (CLI) to receive commands from a user in the form of lines of text. This provides a means of setting parameters for the environment, invoking executables, and providing information to them as to what actions they are to perform.

Clojure - Clojure is a dynamic and functional dialect of the Lisp programming language on the Java platform. Like other Lisp dialects, Clojure treats code as data and has a Lisp

macro system. The current development process is community-driven, overseen by Rich Hickey as its benevolent dictator for life.

Cloudflare - Cloudflare, Inc. is an American content delivery network and DDoS mitigation company, founded in 2009. It primarily acts as a reverse proxy between a website's visitor and the Cloudflare customer's hosting provider.

Cloudfront - Amazon CloudFront is a content delivery network operated by Amazon Web Services. Content delivery networks provide a globally-distributed network of proxy servers that cache content, such as web videos or other bulky media, more locally to consumers, thus improving access speed for downloading the content.

CloudSearch - Amazon CloudSearch is a fully managed service in the cloud that makes it easy to set up, manage, and scale a search solution for your website or application.

Cloudwatch - Amazon CloudWatch collects and visualizes real-time logs, metrics, and event data in automated dashboards to streamline your infrastructure and application maintenance.

CMS - A content management system (CMS) is computer software used to manage the creation and modification of digital content (content management). A CMS is typically used for enterprise content management (ECM) and web content management (WCM).

CMYK - The CMYK color model (also known as process color, or four color) is a subtractive color model, based on the CMY color model, used in color printing, and is also used to describe the printing process itself. The abbreviation CMYK refers to the four ink plates used: cyan, magenta, yellow, and key (black).

Cocotron - Cocotron is a developer SDK that implements a usable amount AppKit and Foundation for Windows and Foundation for Linux/BSD in Objective-C. You need to install cross-compilers and cross-compile the frameworks using Xcode on Mac OS X.

CodeCommit - AWS CodeCommit is a version control service hosted by Amazon Web Services that you can use to privately store and manage assets (such as documents, source code, and binary files) in the cloud.

CodeIgniter - CodeIgniter is an open-source software rapid development web framework, for use in building dynamic websites with PHP.

ColdFusion - Adobe ColdFusion is a commercial rapid web-application development computing platform created by J. J. Allaire in 1995. (The programming language used with that platform is also commonly called ColdFusion, though is more accurately known as CFML.) ColdFusion was originally designed to make it easier to connect simple HTML pages to a database. By version 2 (1996), it became a full platform that included an IDE in addition to a full scripting language.

Confluence - Confluence is a web-based corporate wiki developed by the Australian software company Atlassian. Atlassian wrote Confluence in the Java programming language and first published it in 2004.

Cookies - HTTP cookies (also called web cookies, Internet cookies, browser cookies, or simply cookies) are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer or another device by the user's web browser. Cookies are placed on the device used to access a website, and more than one cookie may be placed on a user's device during a session.

Cordova - Apache Cordova (formerly PhoneGap) is a mobile application development framework created by Nitobi. Adobe Systems purchased Nitobi in 2011, rebranded it as PhoneGap, and later released an open-source version of the software called Apache Cordova. Apache Cordova enables software programmers to build hybrid web applications for mobile devices using CSS3, HTML5, and JavaScript, instead of relying on platform-specific APIs like those in Android, iOS, or Windows Phone. It enables the wrapping up of CSS, HTML, and JavaScript code depending on the platform of the device. It extends the features of HTML and JavaScript to work with the device. The resulting applications are hybrid, meaning that they are neither truly native mobile applications nor purely Web-based.

Core Data - Core Data is an object graph and persistence framework provided by Apple in the macOS and iOS operating systems. It was introduced in Mac OS X 10.4 Tiger and iOS with iPhone SDK 3.0. It allows data organized by the relational entity–attribute model to be serialized into XML, binary, or SQLite stores.

CORS - Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

Couchbase - Couchbase Server, originally known as Membase, is an open-source, distributed multi-model NoSQL document-oriented database software package optimized for interactive applications. These applications may serve many concurrent users by creating, storing, retrieving, aggregating, manipulating, and presenting data.

Couchbase Lite - Couchbase Lite is a developer-friendly, full-featured embedded NoSQL database for apps that run on mobile, desktop, and custom embedded devices.

CouchDB - Apache CouchDB is an open-source document-oriented NoSQL database, implemented in Erlang. CouchDB uses multiple formats and protocols to store, transfer, and process its data. It uses JSON to store data, JavaScript as its query language using MapReduce, and HTTP for an API.

COVID-19 - Coronavirus disease 2019 (COVID-19) is a contagious disease caused by a virus, the severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2). The first known case was identified in Wuhan, China, in December 2019. The disease quickly spread worldwide, resulting in the COVID-19 pandemic.

CPU - A central processing unit (CPU), also called a central processor, main processor or just processor, is the electronic circuitry that executes instructions comprising a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program. This contrasts with external components such as main memory and I/O circuitry, and specialized processors such as graphics processing units (GPUs).

Cross-Cutting Concern - In aspect-oriented software development, cross-cutting concerns are aspects of a program that affect several modules, without the possibility of being encapsulated in any of them.

CSS - Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML or XML (including XML dialects such as SVG, MathML or XHTML). CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.

CTO - A chief technology officer (CTO), also known as a chief technical officer or chief technologist, is an executive-level position in a company or other entity whose occupation is focused on the scientific and technological issues within an organization.

Cucumber - Cucumber is a software tool that supports behavior-driven development (BDD).

Cypher - Cypher is a language originally designed by Andrés Taylor and colleagues at Neo4j Inc., and first implemented by that company in 2011.

DBA - Database administrators use specialized software to store and organize data. The role may include capacity planning, installation, configuration, database design, migration, performance monitoring, security, troubleshooting, as well as backup and data recovery.

DevOps - DevOps is a set of practices that combines software development and IT operations. It aims to shorten the systems development life cycle and provide continuous delivery with high software quality. DevOps is complementary to agile software development; several DevOps aspects came from the agile way of working.

DigitalOcean - DigitalOcean, LLC is an American cloud infrastructure provider headquartered in New York City with data centers worldwide. DigitalOcean provides developers, startups, and SMBs with cloud infrastructure-as-a-service platforms.

Docker - Docker is a set of platform-as-a-service products that use OS-level virtualization to deliver software in packages called containers. The service has both free and premium tiers. The software that hosts the containers is called Docker Engine. It was first started in 2013 and is developed by Docker, Inc.

Docker Hub - Docker Hub is a hosted repository service provided by Docker for finding and sharing container images with your team.

DNS - The Domain Name System is a hierarchical and distributed naming system for computers, services, and other resources in the Internet or other Internet Protocol networks. It associates various information with domain names assigned to each of the associated entities.

Dreamweaver - Adobe Dreamweaver is a proprietary web development tool from Adobe Inc. It was created by Macromedia in 1997 and developed by them until Macromedia was acquired by Adobe Systems in 2005. Adobe Dreamweaver is available for the macOS and

Windows operating systems.

Dropbox - Dropbox is a file hosting service operated by the American company Dropbox, Inc., headquartered in San Francisco, California, U.S. that offers cloud storage, file synchronization, personal cloud, and client software.

DRY - "Don't repeat yourself" is a principle of software development aimed at reducing repetition of software patterns, replacing it with abstractions or using data normalization to avoid redundancy.

Dyno - A dynamometer or "dyno" for short, is a device for simultaneously measuring the torque and rotational speed (RPM) of an engine, motor or other rotating prime mover so that its instantaneous power may be calculated, and usually displayed by the dynamometer itself as kW or bhp.

EC2 - Amazon Elastic Compute Cloud (EC2) is a part of Amazon.com's cloud-computing platform, Amazon Web Services (AWS), that allows users to rent virtual computers on which to run their own computer applications. EC2 encourages scalable deployment of applications by providing a web service through which a user can boot an Amazon Machine Image (AMI) to configure a virtual machine, which Amazon calls an "instance", containing any software desired.

Echo Nest - The Echo Nest is a music intelligence and data platform[clarification needed] for developers and media companies. Owned by Spotify since 2014, the company is based in Somerville, MA. The Echo Nest began as a research spin-off from the MIT Media Lab to understand the audio and textual content of recorded music. Its creators intended it to perform music identification, recommendation, playlist creation, audio fingerprinting, and analysis for consumers and developers.

ECMAScript - ECMAScript is a JavaScript standard intended to ensure the interoperability of web pages across different browsers. It is standardized by Ecma International in the document ECMA-262.

ElastiCache - Amazon ElastiCache is a fully managed in-memory data store and cache service by Amazon Web Services. The service improves the performance of web applications by retrieving information from managed in-memory caches, instead of relying entirely on slower disk-based databases.

Elasticsearch - Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents.

ELB - Elastic Load Balancing (ELB) automatically distributes incoming application traffic across multiple targets and virtual appliances in one or more Availability Zones (AZs).

ELK - "ELK" is the acronym for three open-source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch.

EOL - An end-of-life product is a product at the end of the product lifecycle which prevents users from receiving updates, indicating that the product is at the end of its useful life. At this stage, a vendor stops the marketing, selling, or provisioning of parts, services, or software updates for the product.

ESLint - ESLint is a static code analysis tool for identifying problematic patterns found in JavaScript code. It was created by Nicholas C. Zakas in 2013. Rules in ESLint are configurable, and customized rules can be defined and loaded. ESLint covers both code quality and coding style issues.

ES5 - ES5 is a shortcut for ECMAScript 5. ECMAScript 5 is also known as JavaScript 5. ECMAScript 5 is also known as ECMAScript 2009.

ES6 - ECMAScript 2015 was the second major revision to JavaScript. ECMAScript 2015 is also known as ES6 and ECMAScript 6.

Expo - Expo is an open-source platform for making universal native apps for Android, iOS, and the web with JavaScript and React.

FAANG - FAANG is an acronym that stands for five major, highly successful US tech companies: Facebook (now Meta), Amazon, Apple, Netflix, and Google.

Firebase - Firebase provides detailed documentation and cross-platform SDKs to help you build and ship apps on Android, iOS, the web, C++, and Unity.

Fixed Bid - A fixed-price contract is a type of contract such that the payment amount does not depend on resources used or time expended by the contractor.

Flake8 - Flake8 is a popular lint wrapper for python. Under the hood, it runs three other tools and combines their results: pep8 for checking style. pyflakes for checking syntax. mccabe for checking complexity.

Flash - Adobe Flash (formerly Macromedia) is a multimedia software platform used for the production of animations, rich web applications, desktop applications, mobile apps, mobile games, and embedded web browser video players. Flash displays text, vector graphics, and raster graphics to provide animations, video games, and applications.

Flexbox - CSS Flexible Box Layout, commonly known as Flexbox, is a CSS 3 web layout model. It is in the W3C's candidate recommendation stage. The flex layout allows responsive elements within a container to be automatically arranged depending on viewport size.

Flow - In positive psychology, a flow state, also known colloquially as being in the zone, is the mental state in which a person performing some activity is fully immersed in a feeling of energized focus, full involvement, and enjoyment in the process of the activity.

FOMO - Fear of missing out is the feeling of apprehension that one is either not in the know or missing out on information, events, experiences, or life decisions that could make one's life better.

FP - Functional programming (FP) is a programming paradigm where programs are constructed by applying and composing functions.

FrontPage - Microsoft FrontPage is a discontinued WYSIWYG HTML editor and website administration tool from Microsoft for the Microsoft Windows line of operating systems. It was branded as part of the Microsoft Office suite from 1997 to 2003.

FTP - The File Transfer Protocol (FTP) is a standard communication protocol used for the transfer of computer files from a server to a client on a computer network. FTP is built on a client–server model architecture using separate control and data connections between the client and the server.

G-CORE - G-CORE is a research language designed by a group of academic and industrial researchers and language designers which draws on features of Cypher, PGQL and SPARQL.

GCP - Google Cloud Platform, offered by Google, is a suite of cloud computing services that runs on the same infrastructure that Google uses internally for its end-user products, such as Google Search, Gmail, Google Drive, and YouTube.

GCS - Google Cloud Storage is a RESTful online file storage web service for storing and accessing data on Google Cloud Platform infrastructure. The service combines the performance and scalability of Google's cloud with advanced security and sharing capabilities.

Gherkin - Central to the Cucumber BDD approach is its ordinary language parser called Gherkin. It allows expected software behaviors to be specified in a logical language that customers can understand.

GitHub - GitHub, Inc. is an Internet hosting service for software development and version control using Git. It provides the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project.

GitKraken - GitKraken is a suite of developer tools: GitKraken Git GUI, a Git client available on Mac, Windows, and Linux, GitKraken Issue Boards, Kanban style boards, and GitKraken Timelines, an online timeline maker.

GitLab - GitLab Inc. is an open-core company that operates GitLab, a DevOps software package that combines the ability to develop, secure, and operate software in a single application. The open-source software project was created by Ukrainian developer Dmitriy Zaporozhets and Dutch developer Sytse Sijbrandij.

Google Analytics - Google Analytics is a web analytics service offered by Google that tracks and reports website traffic, currently as a platform inside the Google Marketing Platform brand. Google launched the service in November 2005 after acquiring Urchin.

Gremlin - Gremlin is a graph traversal language and virtual machine developed by Apache TinkerPop of the Apache Software Foundation. Gremlin works for both OLTP-

based graph databases as well as OLAP-based graph processors.

GSQL - GSQL is a language designed for TigerGraph Inc.'s proprietary graph database.

Gulf War - The Gulf War was a 1990–1991 armed campaign waged by a 35-country military coalition in response to the Iraqi invasion of Kuwait.

Hashicorp - HashiCorp is a software company with a freemium business model based in San Francisco, California. HashiCorp provides open-source tools and commercial products that enable developers, operators, and security professionals to provision, secure, run and connect cloud-computing infrastructure.

Haskell - Haskell is a general-purpose, statically-typed, purely functional programming language with type inference and lazy evaluation.

Headless Browser - A headless browser is a web browser without a graphical user interface. Headless browsers provide automated control of a web page in an environment similar to popular web browsers, but they are executed via a command-line interface or using network communication.

Heroku - Heroku is a cloud platform as a service supporting several programming languages. One of the first cloud platforms, Heroku has been in development since June 2007, when it supported only the Ruby programming language, but now supports Java, Node.js, Scala, Clojure, Python, PHP, and Go.

Higher-Order Component - a higher-order component is a function that takes a component and returns a new component.

Higher-Order Function - In mathematics and computer science, a higher-order function is a function that does at least one of the following: takes one or more functions as arguments, and returns a function as its result. All other functions are first-order functions. In mathematics, higher-order functions are also termed operators or functionals.

HR - Human resources is the set of people who make up the workforce of an organization, business sector, industry, or economy. A narrower concept is human capital, the knowledge, and skills that individuals command. Similar terms include manpower, labor, personnel, associates, or simply: people.

HTML5 - HTML5 is a markup language used for structuring and presenting content on the World Wide Web. It is the fifth and final major HTML version that is a World Wide Web Consortium recommendation. The current specification is known as the HTML Living Standard.

Hudson - Hudson is a discontinued continuous integration tool written in Java, which runs in a servlet container such as Apache Tomcat or the GlassFish application

IDE - An integrated development environment is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools, and a debugger.

InnoDB - InnoDB is a storage engine for the database management system MySQL and MariaDB. Since the release of MySQL 5.5.5 in 2010, it replaced MyISAM as MySQL's default table type. It provides the standard ACID-compliant transaction features, along with foreign key support.

IntelliJ - IntelliJ IDEA is an integrated development environment written in Java for developing computer software written in Java, Kotlin, Groovy, and other JVM-based languages. It is developed by JetBrains and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition.

iOS - iOS is a mobile operating system created and developed by Apple Inc. exclusively for its hardware. It is the operating system that powers many of the company's mobile devices, including the iPhone.

IoT - The Internet of things describes physical objects with sensors, processing ability, software, and other technologies that connect and exchange data with other devices and systems over the Internet or other communications networks.

IP (business) - Intellectual property is a category of property that includes intangible creations of the human intellect. There are many types of intellectual property, and some countries recognize it more than others. The best-known types are patents, copyrights, trademarks, and trade secrets

IP (technology) - The Internet Protocol (IP) is the network layer communications protocol in the Internet protocol suite for relaying datagrams across network boundaries.

ISP - An Internet service provider is an organization that provides services for accessing, using, or participating in the Internet. ISPs can be organized in various forms, such as commercial, community-owned, non-profit, or otherwise privately owned.

IT - Information technology (IT) is the use of computers to create, process, store, retrieve, and exchange all kinds of data and information. IT forms part of information and communications technology (ICT). An information technology system (IT system) is generally an information system, a communications system, or, more specifically speaking, a computer system -- including all hardware, software, and peripheral equipment -- operated by a limited group of IT users.

Jasmine - Jasmine is an open-source testing framework for JavaScript. It aims to run on any JavaScript-enabled platform, to not intrude on the application nor the IDE, and to have easy-to-read syntax. It is heavily influenced by other unit testing frameworks, such as ScrewUnit, JSSpec, JSpec, and RSpec.

Java - Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible.

JavaDoc - Javadoc is a documentation generator created by Sun Microsystems for the Java language for generating API documentation in HTML format from Java source code. The HTML format is used for adding the convenience of being able to hyperlink related documents together.

JavaScript - JavaScript, often abbreviated as JS, is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. As of 2022, 98% of websites use JavaScript on the client side for webpage behavior, often incorporating third-party libraries.

Jenkins - Jenkins is an open-source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat.

Jira - Jira is a proprietary issue-tracking product developed by Atlassian that allows bug tracking and agile project management.

JMeter - Apache JMeter is an Apache project that can be used as a load-testing tool for analyzing and measuring the performance of a variety of services, with a focus on web applications.

jQuery - jQuery is a JavaScript library designed to simplify HTML DOM tree traversal and manipulation, as well as event handling, CSS animation, and Ajax. It is free, open-source software using the permissive MIT License. As of Aug 2022, jQuery is used by 77% of the 10 million most popular websites.

JSDoc - JSDoc is a markup language used to annotate JavaScript source code files. Using comments containing JSDoc, programmers can add documentation describing the application programming interface of the code they're creating.

JSON - JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays. It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

JSP - Jakarta Server Pages (formerly JavaServer Pages) is a collection of technologies that helps software developers create dynamically generated web pages based on HTML, XML, SOAP, or other document types. Released in 1999 by Sun Microsystems, JSP is similar to PHP and ASP but uses the Java programming language.

JVM - A Java virtual machine is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation.

JWT - JSON Web Token is a proposed Internet standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims. The tokens are signed either using a private secret or a public/private key.

Kanban - Kanban is a lean method to manage and improve work across human systems. This approach aims to manage work by balancing demands with available capacity, and by improving the handling of system-level bottlenecks.

Lamborghini - Automobili Lamborghini S.p.A. is an Italian brand and manufacturer of luxury sports cars and SUVs based in Sant'Agata Bolognese. The company is owned by the Volkswagen Group through its subsidiary Audi.

LED - A light-emitting diode (LED) is a semiconductor device that emits light when current flows through it.

Linter - Lint, or a linter, is a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs. The term originates from a Unix utility that examined C language source code.

Lisp - Lisp is a family of programming languages with a long history and a distinctive, fully parenthesized prefix notation. Originally specified in 1960, Lisp is the second-oldest high-level programming language still in common use, after Fortran.

Lodash - Lodash is a JavaScript library that provides utility functions for common programming tasks.

LTS - Long-term support (LTS) is a product lifecycle management policy in which a stable release of computer software is maintained for a longer period of time than the standard edition.

MailChimp - Mailchimp is a marketing automation platform and email marketing service. "Mailchimp" is the trade name of its operator, Rocket Science Group, an American company founded in 2001 by Ben Chestnut and Mark Armstrong, with Dan Kurzius joining at a later date.

MediaConvert - AWS Elemental MediaConvert is a file-based video processing service that transcodes content for broadcast and multi-screen delivery at scale.

MediaNet - Originally founded in 2000 as "MusicNet," MediaNet was established by the Major labels and RealNetworks as a digital music platform for business. Since that time, MediaNet has become, and remains, the only business-focused digital music platform to unify catalog services and rights administration under one roof.

Memcached - Memcached is a general-purpose distributed memory-caching system. It is often used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source must be read. Memcached is free and open-source software, licensed under the Revised BSD license.

Mercurial - Mercurial is a distributed revision control tool for software developers. It is supported on Microsoft Windows and Unix-like systems, such as FreeBSD, macOS, and Linux.

Metallica - Metallica is an American heavy metal band. The band was formed in 1981 in Los Angeles by vocalist/guitarist James Hetfield and drummer Lars Ulrich, and has been based in San Francisco for most of its career.

MFA - Multi-factor authentication (MFA; encompassing two-factor authentication, or 2FA,

along with similar terms) is an electronic authentication method in which a user is granted access to a website or application only after successfully presenting two or more pieces of evidence (or factors) to an authentication mechanism: knowledge (something only the user knows), possession (something only the user has), and inference (something only the user is).

Middleware - Middleware is a type of computer software that provides services to software applications beyond those available from the operating system. It can be described as "software glue".

Minecraft - Minecraft is a sandbox game developed by Mojang Studios.

Mixpanel - Mixpanel is a tool that allows you to analyze how users interact with your Internet-connected product. It's designed to make teams more efficient by allowing everyone to analyze user data in real-time to identify trends, understand user behavior, and make decisions about your product.

MongoDB - MongoDB is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. MongoDB is developed by MongoDB Inc. and licensed under the Server Side Public License which is deemed non-free by several distributions.

Mono - Mono is a free and open-source .NET Framework-compatible software framework. Originally by Ximian, it was later acquired by Novell, and is now being led by Xamarin, a subsidiary of Microsoft and the .NET Foundation. Mono can be run on many software systems.

Morpheus - The opencypher Morpheus project implements Cypher for Apache Spark users. Commencing in 2016, this project originally ran alongside three related efforts, in which Morpheus designers also took part: SQL/PGQ, G-CORE and design of Cypher extensions for querying and constructing multiple graphs. The Morpheus project acted as a testbed for extensions to Cypher (known as "Cypher 10") in the two areas of graph DDL and query language extensions.

MOS - Enlisted soldiers are categorized by their assigned job called a Military Occupational Specialty (MOS). MOS are labeled with a short alphanumerical code called a military occupational core specialty code (MOSC), which consists of a two-digit number appended by a Latin letter. Related MOSs are grouped together by Career Management Fields (CMF). For example, an enlisted soldier with MOSC 11B works as an infantry soldier (their MOS) and is part of CMF 11 (the CMF for infantry).

MPH - Miles per hour (mph, m.p.h., MPH, or mi/h) is a British imperial and United States customary unit of speed expressing the number of miles traveled in one hour.

MVC - Model-view-controller is a software architectural pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted by the user.

MyISAM - MyISAM was the default storage engine for the MySQL relational database

management system versions prior to 5.5 released in December 2009. It is based on the older ISAM code, but it has many useful extensions.

MySQL - MySQL is an open-source relational database management system. Its name is a combination of "My", the name of co-founder Michael Widenius's daughter My, and "SQL", the acronym for Structured Query Language.

NDA - A non-disclosure agreement is a legal contract or part of a contract between at least two parties that outlines confidential material, knowledge, or information that the parties wish to share with one another for certain purposes, but wish to restrict access to.

NebulaGraph - A graph database self-described as "The open source graph database built for super large-scale graphs with milliseconds of latency."

Neo4j - Neo4j is a graph database management system developed by Neo4j, Inc. It is implemented in Java and accessible from software written in other languages using the Cypher query language through a transactional HTTP endpoint, or through the binary "Bolt" protocol.

New Relic - New Relic is a San Francisco, California-based technology company that develops cloud-based software to help website and application owners track the performance of their services.

nGQL - nGQL is a declarative graph query language for NebulaGraph.

Nginx - Nginx is a web server that can also be used as a reverse proxy, load balancer, mail proxy, and HTTP cache. The software was created by Igor Sysoev and publicly released in 2004. Nginx is free and open-source software, released under the terms of the 2-clause BSD license.

ngrok - ngrok is a simplified API-first ingress-as-a-service that adds connectivity, security, and observability to your apps with no code changes

NodeJS - Node.js is an open-source server environment. Node.js is cross-platform and runs on Windows, Linux, Unix, and macOS. Node.js is a back-end JavaScript runtime environment. Node.js runs on the V8 JavaScript Engine and executes JavaScript code outside a web browser.

NoSQL - A NoSQL database provides a mechanism for the storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but the name "NoSQL" was only coined in the early 21st century, triggered by the needs of Web 2.0 companies.

npm - npm (originally short for Node Package Manager) is a package manager for the JavaScript programming language maintained by npm, Inc. npm is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called npm, and an online database of public and paid-for private packages called the npm registry. The registry is accessed via the client, and the available packages can be browsed and searched via the npm website. The package manager and the registry are managed by npm, Inc.

OAuth - OAuth is an open standard for access delegation, commonly used as a way for internet users to grant websites or applications access to their information on other websites but without giving them the passwords.

OCD - Obsessive-compulsive disorder (OCD) is a common, chronic, and long-lasting disorder in which a person has uncontrollable, reoccurring thoughts ("obsessions") and/or behaviors ("compulsions") that he or she feels the urge to repeat over and over.

OIDC - OpenID is an open standard and decentralized authentication protocol promoted by the non-profit OpenID Foundation. It allows users to be authenticated by co-operating sites (known as relying parties, or RP) using a third-party identity provider (IDP) service, eliminating the need for webmasters to provide their own ad hoc login systems, and allowing users to log in to multiple unrelated websites without having to have a separate identity and password for each.

OOP - Object-oriented programming is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields, and the code is in the form of procedures. A common feature of objects is that procedures are attached to them and can access and modify the object's data fields.

OPSWAT - OPSWAT is a cybersecurity company founded in 2002. Based in San Francisco, CA, they focus on cyberthreat prevention including a focus on malware, sensitive information, and application vulnerabilities.

ORM - Object-relational mapping in computer science is a programming technique for converting data between type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

OS - An operating system is system software that manages computer hardware, and software resources, and provides common services for computer programs.

ParamStore - Parameter Store, a capability of AWS Systems Manager, provides secure, hierarchical storage for configuration data management and secrets management.

Patriot Missile - The MIM-104 Patriot is a surface-to-air missile system, the primary of its kind used by the United States Army and several allied states. It is manufactured by the U.S. defense contractor Raytheon and derives its name from the radar component of the weapon system.

Pen Test - A penetration test, colloquially known as a pen test or ethical hacking, is an authorized simulated cyberattack on a computer system, performed to evaluate the security of the system; this is not to be confused with a vulnerability assessment.

Perl - Perl is a highly expressive programming language: source code for a given algorithm can be short and highly compressible.

PGQL - PGQL is a language designed and implemented by Oracle Inc., but made available as an open source specification, along with JVM parsing software. PGQL

combines familiar SQL SELECT syntax including SQL expressions and result ordering and aggregation with a pattern-matching language very similar to that of Cypher.

PHP - PHP is a general-purpose scripting language geared toward web development. It was originally created by Danish-Canadian programmer Rasmus Lerdorf in 1993 and released in 1995. The PHP reference implementation is now produced by The PHP Group.

PHPUnit - PHPUnit is a unit-testing framework for the PHP programming language. It is an instance of the xUnit architecture for unit testing frameworks that originated with SUnit and became popular with JUnit. PHPUnit was created by Sebastian Bergmann and its development is hosted on GitHub.

PII - Personal data, also known as personal information or personally identifiable information (PII), is any information related to an identifiable person.

Pivotal - Pivotal Tracker is a product-planning and management tool designed exclusively for modern software development teams. It was created by Pivotal Labs which was acquired by VMWare in 2019.

Plaintext - In cryptography, plaintext usually means unencrypted information pending input into cryptographic algorithms, usually encryption algorithms. This usually refers to data that is transmitted or stored unencrypted.

PostgreSQL - PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance. It was originally named POSTGRES, referring to its origins as a successor to the Ingres database developed at the University of California, Berkeley.

Postman - Postman is an API platform for developers to design, build, test, and iterate their APIs. As of April 2022, Postman reports having more than 20 million registered users and 75,000 open APIs, which it says constitutes the world's largest public API hub.

PR - A pull request – also referred to as a merge request – is an event that takes place in software development when a contributor/developer is ready to begin the process of merging new code changes with the main project repository.

Procedural Programming - Procedural programming is a programming paradigm, derived from imperative programming, based on the concept of the procedure call. Procedures simply contain a series of computational steps to be carried out.

Promise - A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

PropTypes - ReactJS offers type checking on props for a component when it is assigned the special propTypes property. PropTypes exports a range of validators that can be used to make sure the data you receive is valid.

PSN - PlayStation Network (PSN) is a digital media entertainment service provided by Sony Interactive Entertainment. Launched in November 2006, PSN was originally

conceived for the PlayStation video game consoles but soon extended to encompass smartphones, tablets, Blu-ray players, and high-definition televisions. This service is the account for PlayStation consoles, accounts can store games and other content.

PTO - Paid time off, planned time off, or personal time off, is a policy in some employee handbooks that provides a bank of hours in which the employer pools sick days, vacation days, and personal days that allow employees to use as the need or desire arises.

PubMed - PubMed is a free search engine accessing primarily the MEDLINE database of references and abstracts on life sciences and biomedical topics. The United States National Library of Medicine at the National Institutes of Health maintain the database as part of the Entrez system of information retrieval.

PubSub - In software architecture, publish-subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

Punched Cards - A punched card is a piece of stiff paper that holds digital data represented by the presence or absence of holes in predefined positions. Punched cards were once common in data processing applications or to directly control automated machinery.

Python - Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured, object-oriented, and functional programming.

QA - Quality assurance is the term used in both manufacturing and service industries to describe the systematic efforts taken to ensure that the product delivered to the customer meet the contractual and other agreed-upon performance, design, reliability, and maintainability expectations of that customer.

R&D - Research and development, known in Europe as research and technological development, is the set of innovative activities undertaken by corporations or governments in developing new services or products and improving existing ones.

R1 - The Yamaha YZF-R1, or simply R1, is a 1,000 cc-class sports motorcycle made by Yamaha.

RAID - RAID is a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both

RAM - Random-access memory is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code.

Ramda - A practical functional library for JavaScript programmers.

Raspberry Pi - Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom. The Raspberry Pi project originally leaned towards the promotion of teaching basic computer science in schools and in developing countries.

RBAC - In computer systems security, role-based access control or role-based security is an approach to restricting system access to authorized users. It is an approach to implement mandatory access control or discretionary access control.

RDBMS - A relational database is a (most commonly digital) database based on the relational model of data, as proposed by E. F. Codd in 1970. A system used to maintain relational databases is a relational database management system (RDBMS). Many relational database systems are equipped with the option of using SQL (Structured Query Language) for querying and maintaining the database.

RDS - Amazon Relational Database Service is a distributed relational database service by Amazon Web Services. It is a web service running "in the cloud" designed to simplify the setup, operation, and scaling of a relational database for use in applications.

React - React is a free and open-source front-end JavaScript library for building user interfaces based on UI components. It is maintained by Meta and a community of individual developers and companies.

React Native - React Native is an open-source UI software framework created by Meta Platforms, Inc. It is used to develop applications for Android, Android TV, iOS, macOS, tvOS, Web, Windows, and UWP by enabling developers to use the React framework along with native platform capabilities.

README - In software development, a README file contains information about the other files in a directory or archive of computer software. A form of documentation, it is usually a simple plaintext file called README, Read Me, READ.ME, README.TXT, README.md (to indicate the use of Markdown), or README.1ST

Recharts - Recharts is a Redefined chart library built with React and D3.

Redis - Redis is an in-memory data structure store, used as a distributed, in-memory key-value database, cache and message broker, with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indices.

Redshift - Amazon Redshift is a data warehouse product that forms part of the larger cloud-computing platform Amazon Web Services. It is built on top of technology from the massive parallel processing data warehouse company ParAccel, to handle large-scale data sets and database migrations.

REPL - A read–eval–print loop, also termed an interactive top-level or language shell, is a simple interactive computer programming environment that takes single user inputs, executes them, and returns the result to the user; a program written in a REPL environment is executed piecewise.

REST - Representational state transfer (REST) is a software architectural style that describes a uniform interface between physically separate components, often across the Internet in a client-server architecture. REST defines four interface constraints: Identification of resources. Manipulation of resources.

RFID - Radio-frequency identification (RFID) uses electromagnetic fields to automatically identify and track tags attached to objects. An RFID system consists of a tiny radio transponder, a radio receiver, and a transmitter. When triggered by an electromagnetic interrogation pulse from a nearby RFID reader device, the tag transmits digital data, usually an identifying inventory number, back to the reader.

RGB - The RGB color model is an additive color model in which the red, green, and blue primary colors of light are added together in various ways to reproduce a broad array of colors. The name of the model comes from the initials of the three additive primary colors, red, green, and blue.

ROI - Return on investment or return on costs is a ratio between net income and investment. A high ROI means the investment's gains compare favorably to its cost. As a performance measure, ROI is used to evaluate the efficiency of an investment or to compare the efficiencies of several different investments.

Ruby - Ruby is an interpreted, high-level, general-purpose programming language which supports multiple programming paradigms. It was designed with an emphasis on programming productivity and simplicity. In Ruby, everything is an object, including primitive data types.

Ruby on Rails - Ruby on Rails is a server-side web application framework written in Ruby under the MIT License. Rails is a model–view–controller framework, that provides default structures for a database, a web service, and web pages.

S3 - Amazon S3 or Amazon Simple Storage Service is a service offered by Amazon Web Services that provides object storage through a web service interface. Amazon S3 uses the same scalable storage infrastructure that Amazon.com uses to run its e-commerce network.

SAAS - Software as a service is a software licensing and delivery model in which software is licensed on a subscription basis and is centrally hosted. SaaS is also known as "on-demand software" and Web-based / Web-hosted software.

Salesforce - Salesforce, Inc. is an American cloud-based software company headquartered

in San Francisco, California. It provides customer relationship management software and applications focused on sales, customer service, marketing automation, analytics, and application development.

SAML - Security Assertion Markup Language is an open standard for exchanging authentication and authorization data between parties, in particular, between an identity provider and a service provider. SAML is an XML-based markup language for security assertions.

Sanctuary - Sanctuary is a JavaScript functional programming library inspired by Haskell and PureScript. It's stricter than Ramda and provides a similar suite of functions.

Scud Missile - A Scud missile is one of a series of tactical ballistic missiles developed by the Soviet Union during the Cold War. It was exported widely to both Second and Third World countries. The term comes from the NATO reporting name attached to the missile by Western intelligence agencies.

SDK - A software development kit is a collection of software development tools in one installable package. They facilitate the creation of applications by having a compiler, debugger, and sometimes a software framework. They are normally specific to a hardware platform and operating system combination.

SecOps - Security Operations is a collaboration between IT security and operations teams that integrates tools, processes, and technology to keep an enterprise secure while reducing risk.

Selenium - Selenium is an open-source umbrella project for a range of tools and libraries aimed at supporting browser automation. It provides a playback tool for authoring functional tests across most modern web browsers, without the need to learn a test scripting language.

SemVer - Semantic versioning is a conventional naming system used to specify the version of the software, a plugin, a library, or any other technology with a dot-separated 3 numbers pattern starting from 0 until infinity. Example: 0.1.32

Sendgrid - SendGrid is a Denver, Colorado-based customer communication platform for transactional and marketing email. The company was founded by Isaac Saldana, Jose Lopez, and Tim Jenkins in 2009, and incubated through the Techstars accelerator program.

Sequelize - Sequelize is a modern TypeScript and Node.js ORM for Oracle, Postgres, MySQL, MariaDB, SQLite and SQL Server, and more.

SharePoint - SharePoint is a web-based collaborative platform that integrates natively with Microsoft Office. Launched in 2001, SharePoint is primarily sold as a document management and storage system, but the product is highly configurable and its usage varies substantially among organizations.

Slack - Slack is an instant messaging program designed by Slack Technologies and owned by Salesforce. Although Slack was developed for professional and organizational communications, it has been adopted as a community platform.

SMS - Short Message/Messaging Service, commonly abbreviated as SMS, is a text messaging service component of most telephone, Internet, and mobile device systems. It uses standardized communication protocols that let mobile devices exchange short text messages.

SMTP - The Simple Mail Transfer Protocol is an Internet standard communication protocol for electronic mail transmission. Mail servers and other message transfer agents use SMTP to send and receive mail messages.

Snowflake - Snowflake Inc. is a cloud computing-based data cloud company based in Bozeman, Montana. It was founded in July 2012 and was publicly launched in October 2014 after two years in stealth mode. The firm offers a cloud-based data storage and analytics service, generally termed "data-as-a-service".

SOA - In software engineering, service-oriented architecture is an architectural style that focuses on discrete services instead of a monolithic design.

SOAP - SOAP (formerly a backronym for Simple Object Access Protocol) is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP), although some legacy systems communicate over Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

Social Login - Social login is a form of single sign-on using existing information from a social networking service such as Facebook, Twitter, or Google, to sign into a third-party website instead of creating a new login account specifically for that website. It is designed to simplify logins for end users as well as provide more and more reliable demographic information to web developers.

SonarCloud - SonarCloud is a cloud-based code analysis service designed to detect code quality issues in 25 different programming languages, continuously ensuring the maintainability, reliability, and security of your code.

SonarQube - SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs and code smells in 29 programming languages.

SourceForge - SourceForge is a web service that offers software consumers a centralized online location to control and manage open-source software projects and research business software.

SourceTree - Sourcetree is a free graphical user interface (GUI) desktop client that simplifies how you interact with Git repositories so that you can fully concentrate on coding.

SoW - A statement of work is a document routinely employed in the field of project management. It is the narrative description of a project's work requirement. It defines project-specific activities, deliverables, and timelines for a vendor providing services to

the client.

Splunk - Splunk Inc. is an American software company based in San Francisco, California, that produces software for searching, monitoring, and analyzing machine-generated data via a Web-style interface.

SSD - A solid-state drive (SSD) is a solid-state storage device that uses integrated circuit assemblies to store data persistently, typically using flash memory, and functioning as secondary storage in the hierarchy of computer storage.

Strategy (pattern) - In computer programming, the strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime.

Subversion - Apache Subversion is a software versioning and revision control system distributed as open source under the Apache License. Software developers use Subversion to maintain current and historical versions of files such as source code, web pages, and documentation.

SQL - Structured Query Language, abbreviated as SQL, is a domain-specific language used in programming and designed for managing data held in a relational database management system, or for stream processing in a relational data stream management system.

SQLite - SQLite is a database engine written in the C programming language. It is not a standalone app; rather, it is a library that software developers embed in their apps. As such, it belongs to the family of embedded databases.

SSH - The Secure Shell Protocol is a cryptographic network protocol for operating network services securely over an unsecured network. Its most notable applications are remote login and command-line execution. SSH applications are based on a client-server architecture, connecting an SSH client instance with an SSH server.

SVN - Apache Subversion is a software versioning and revision control system distributed as open source under the Apache License. Software developers use Subversion to maintain current and historical versions of files such as source code, web pages, and documentation.

Swift - Swift is a general-purpose, multi-paradigm, compiled programming language developed by Apple Inc. and the open-source community.

Synapse - Azure Synapse Analytics is a limitless analytics service that brings together data integration, enterprise data warehousing, and big data analytics.

T&M - Time and materials (T&M) is a standard phrase in a contract for construction, product development, or any other piece of work in which the employer agrees to pay the contractor based upon the time spent by the contractor's employees and subcontractor's employees to perform the work, and for materials used in the construction (plus the contractor's mark up on the materials used), no matter how much work is required to complete construction. Time and materials is generally used in projects in which it is not possible to accurately estimate the size of the project, or when it is expected that the

project requirements would most likely change.

TigerGraph - TigerGraph is a native parallel graph database purpose-built for loading massive amounts of data (terabytes) in hours and analyzing as many as 10 or more hops deep into relationships in real time.

TortoiseSVN - TortoiseSVN is a Subversion client, implemented as a Microsoft Windows shell extension, that helps programmers manage different versions of the source code for their programs. It is free software released under the GNU General Public License.

Tower - Tower provides a comprehensive, simple interface to perform basic and advanced Git commands.

Trello - Trello is a web-based, kanban-style, list-making application and is developed by Trello Enterprise, a subsidiary of Atlassian. Created in 2011 by Glitch, it was spun out to form the basis of a separate company in New York City in 2014 and sold to Atlassian in January 2017.

Try/Catch - The try...catch statement is comprised of a try block and either a catch block, a finally block, or both. The code in the try block is executed first, and if it throws an exception, the code in the catch block will be executed. The code in the finally block will always be executed before control flow exits the entire construct.

TTL - Time to live or hop limit is a mechanism that limits the lifespan or lifetime of data in a computer or network. TTL may be implemented as a counter or timestamp attached to or embedded in the data. Once the prescribed event count or timespan has elapsed, data is discarded or revalidated.

Twilio - Twilio is an American company based in San Francisco, California, which provides programmable communication tools for making and receiving phone calls, sending and receiving text messages, and performing other communication functions using its web service APIs.

UI - In the industrial design field of human-computer interaction, a user interface is the space where interactions between humans and machines occur.

Unity - Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Worldwide Developers Conference as a Mac OS X game engine. The engine has since been gradually extended to support a variety of desktop, mobile, console, and virtual reality platforms.

US - United States

UUID - A universally unique identifier is a 128-bit label used for information in computer systems. The term globally unique identifier is also used. When generated according to the standard methods, UUIDs are, for practical purposes, unique.

UX - User experience design is the process of defining the experience a user would go through when interacting with a digital product or website. Design decisions in UX design are often driven by research, data analysis, and test results rather than aesthetic

preferences and opinions.

Underscore.js - Underscore.js is a JavaScript library that provides utility functions for common programming tasks. It is comparable to features provided by Prototype.js and the Ruby language but opts for a functional programming design instead of extending object prototypes.

Vagrant - Vagrant is an open-source software product for building and maintaining portable virtual software development environments; e.g., for VirtualBox, KVM, Hyper-V, Docker containers, VMware, and AWS. It tries to simplify the software configuration management of virtualization in order to increase development productivity.

VCS - Version control systems are software tools that help software teams manage changes to source code over time. As development environments have accelerated, version control systems help software teams work faster and smarter.

Visual Basic - The original Visual Basic is a third-generation event-driven programming language from Microsoft known for its Component Object Model programming model first released in 1991 and declared legacy in 2008. Microsoft intended Visual Basic to be relatively easy to learn and use.

Visual Studio - Visual Studio is an integrated development environment from Microsoft. It is used to develop computer programs including websites, web apps, web services, and mobile apps.

VPN - A virtual private network (VPN) extends a private network across a public network and enables users to send and receive data across shared or public networks as if their computing devices were directly connected to the private network. The benefits of a VPN include increases in functionality, security, and management of the private network. It provides access to resources that are inaccessible on the public network and is typically used for remote workers. Encryption is common, although not an inherent part of a VPN connection.

VPs - Vice Presidents

VR - Virtual reality (VR) is a simulated experience that employs pose tracking and 3D near-eye displays to give the user an immersive feel of a virtual world.

VSCode - Visual Studio Code, also commonly referred to as VS Code, is a source-code editor made by Microsoft with the Electron Framework, for Windows, Linux, and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.

Vue - Vue.js is an open-source model-view-viewmodel front end JavaScript framework for building user interfaces and single-page applications. It was created by Evan You, and is maintained by him and the rest of the active core team members.

WAI-ARIA - Web Accessibility Initiative – Accessible Rich Internet Applications is a technical specification published by the World Wide Web Consortium that specifies how to increase the accessibility of web.

WCAG - The Web Content Accessibility Guidelines are part of a series of web accessibility guidelines published by the Web Accessibility Initiative of the World Wide Web Consortium, the main international standards organization for the Internet.

WebbIE - The WebbIE software programs are programs that make it easier for blind and visually-impaired people, especially those using screen readers, to browse the web, get the latest news, listen to podcasts and radio stations, and other common tasks. They work with any screen reader, including JAWS, WindowEyes, Thunder, NVDA, and Narrator. They have been provided completely free since 2001 by Dr. Alasdair King.

Webpack - Webpack is a free and open-source module bundler for JavaScript. It is made primarily for JavaScript, but it can transform front-end assets such as HTML, CSS, and images if the corresponding loaders are included. Webpack takes modules with dependencies and generates static assets representing those modules.

WordPress - WordPress is a free and open-source content management system written in hypertext preprocessor language and paired with a MySQL or MariaDB database with supported HTTPS. Features include a plugin architecture and a template system, referred to within WordPress as "Themes".

Xcode - Xcode is Apple's integrated development environment for macOS, used to develop software for macOS, iOS, iPadOS, watchOS, and tvOS. It was initially released in late 2003.

XML - Extensible Markup Language is a markup language and file format for storing, transmitting, and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

YUI - The Yahoo! User Interface Library is a discontinued open-source JavaScript library for building richly interactive web applications using techniques such as Ajax, DHTML, and DOM scripting. YUI includes several core CSS resources. It is available under a BSD License.

Zend - Zend, formerly Zend Technologies, is a Minneapolis, Minnesota-based software company. The company's products, which include Zend Studio, assist software developers with developing, deploying, and managing PHP-based web applications.

Legal

COPYRIGHT REMINDER

- Copyright © 2023 Sean Cannon. All rights reserved. No portion of this book may be reproduced in any form without written permission from the publisher or author, except as permitted by U.S. copyright law. Cover designed by Sean Cannon.

DISCLAIMERS

For the disclaimers below, the pronouns I, me, he, his, and mine refer to Sean Cannon, the author.

- Several times in this book, I used fictitious scenarios. For example, I scripted some conversations with product managers or sales prospects or when doing code reviews on a pull request. These are a subset of the fictional text contained in this book. The stories I told involving my career are all true. Stories I made up to teach a lesson are fiction. So, unless otherwise indicated, all the names, characters, businesses, places, events, and incidents in this book are either the product of my imagination or used in a fictitious manner. Any resemblance to actual persons, living or dead, or actual events is purely coincidental.
- This glossary in this book uses definitions generously provided by Wikipedia, which are under the Creative Commons Attribution-Share-Alike License 3.0. It also uses definitions from product or company home pages. Neither Sean Cannon (me) nor Alien Creations, Inc (my company) make any claims of ownership or that these definitions are or will remain accurate. These definitions are provided for educational reference purposes only. Please do your own research online for the latest and most accurate definitions of these terms.
- The annual income and career accomplishments stated in this book are my personal accomplishments. Please understand these results are not typical. I am not implying you'll duplicate them (or do anything for that matter). The average person who buys "how to" information gets little to no results. I am using these references to convey what is possible, as I have achieved these results with years of hard work and some circumstances beyond my control. Your results will vary and depend on many factors, including but not limited to your background, experience, geographical location, hardware, and network limitations, personality, attitude, and work ethic. A software professional's career entails risk and massive and consistent effort and action. If you're unwilling to accept that, please do not purchase this book.
- I have made every effort to ensure the accuracy of the information in this book. However, the information contained in this book is for general information purposes only and is not intended as bespoke professional advice or professional career advice. I cannot guarantee the accuracy, completeness, or suitability of the information in this book and will not be liable for any errors or omissions. I do not accept any responsibility for any consequences that may

result from using the information in this book. Please contact an appropriate professional directly if you require specific career advice.

- While the guidance in this book has worked for me and some colleagues in my network, neither I nor Alien Creations, Inc (my company, the publisher of this book) are responsible for any specific career needs or circumstances that readers may have. The information contained in this book is not intended to be a substitute for professional career advice, assessment, or guidance. Please consult a qualified career professional if you have any concerns about your career path or job search.
- The views expressed in this book are my own and do not necessarily reflect the views of any person or company mentioned in the book. I am not endorsed by or sponsored by any person or company mentioned in the book. I am affiliated with specific companies and colleagues only where I mention so in the book.