Mastering the Coding Mindset

Tips and Tricks for Becoming a Better Developer and Having a Successful Career

by **Mads Brodt**

Contents

Introduction

This book contains the ideas and tips that I consider most valuable in order to have a successful development career. It will help you level up your technical skills, frame your mindset, and land your dream development job. The book is designed to be timeless and cross-language, so it doesn't matter if you're into front-end, back-end, mobile, Cloud, DevOps, IoT, data science, AI or Machine Learning - this book is for you.

A career in development is one of the most fulfilling careers you can imagine: You can make good money; You can build stuff to fuel your creative drive; And you can even write programs and applications that help people all around the world; But there's also a lot of pitfalls. Self-doubt, burnout, and job rejections, just to name a few. This book provides my best advice for how to deal with these situations.

The book is split into 4 parts:

1. Technical fundamentals
2. Productivity
3. Mindset
4. Landing the job

Each section has several related chapters, packed with applicable tips & tricks, combined with my own personal experiences. Each chapter can

be read in isolation, so you can skip the chapters that aren't relevant to you.

I've been working as a front-end developer for a few years now, and I've already learned a lot of valuable lessons that I wish I knew when starting out. Knowing the contents of this book would have saved me a lot of emotional distress, while also helping me improve my technical skills more quickly. That's why I'm writing it - to help you on your coding journey. Most of the examples are from my own experience as a front-end developer, but as long as you deal with code in any fashion, the concepts taught will help you on your journey. I also see my brief experience as a strength. Because I'm still new to development, and were in your shoes not that long ago, I feel like I understand the pain points and pitfalls that new developers encounter.

I also feel that teaching is one of the best ways to enforce my own learnings. When I was gathering ideas and writing the book, I had to actively deploy all of the strategies and tactics in my own life, since I don't want to write about something that I haven't personally experienced. This has made me more productive, helped me cope with imposter syndrome, and even land a new job.

Coding can be overwhelming, especially when you're just starting out. There's so much to learn, and it's hard to know what to focus on.

This book is my answer.

PART 1

Technical Fundamentals

Break problems into smaller tasks

“How do you eat an elephant? One bite at a time.“

Coding is exactly the same. It’s easy to get stuck if your current task is to build a website, or design an algorithm. It’s too broad. The problem is simply overwhelming, and you don’t know where to begin.

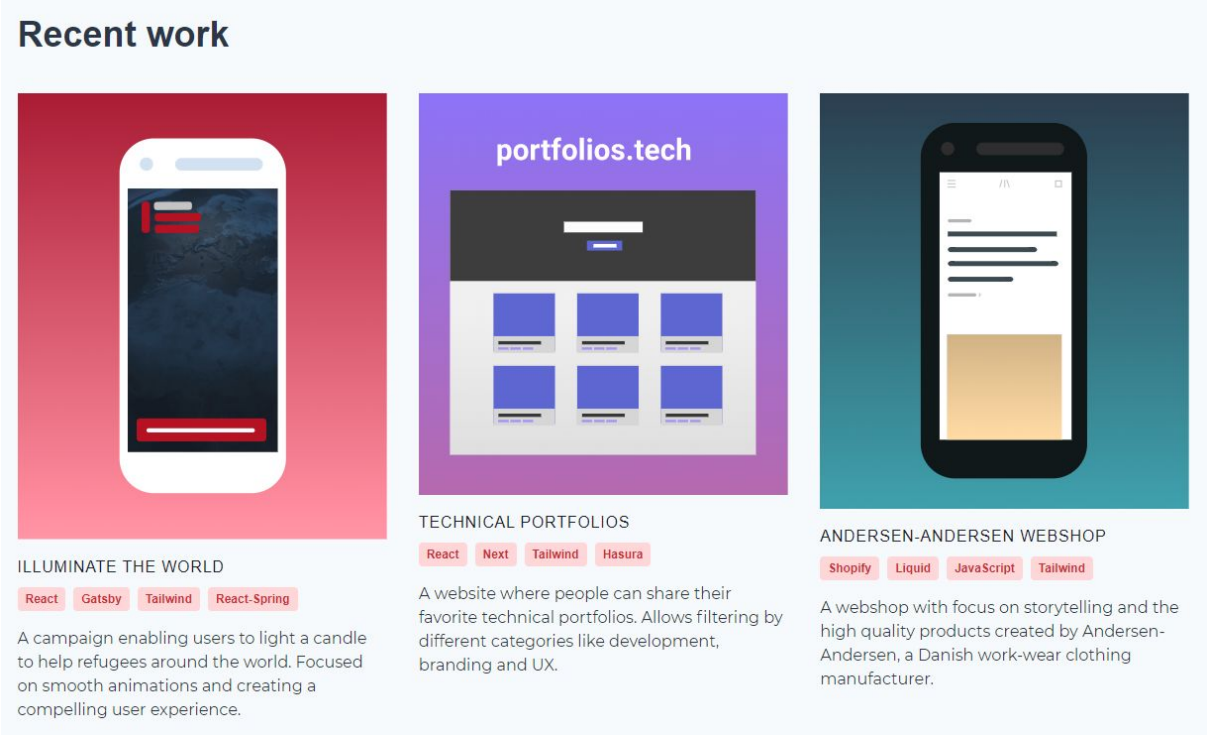
This can scare a lot of people off coding. It makes projects look insurmountable, which hurts your confidence. It also makes you unable to tackle projects that you could actually complete with the right mindset. You might look at someone else's portfolio website or software product and think “I could never build that! I wouldn’t even know where to start!”

I felt this way too, until I learned a very important lesson: **break problems down.**

Let’s take a portfolio website as an example, and try to break it down into smaller, individual parts:

There’s probably a navigation bar at the top with a logo. Below is a header and some supporting text. Then you have a section of projects, each project being its own “card”. There’s definitely a “Contact Me” section somewhere, with some info and maybe a form.

If you were to recreate this page, or one similar to it, you should tackle each of these sections **in isolation**. Let's say we we want to build a "Projects" section (example from my own website):



Where do we start? Naturally, by breaking the problem down again: Let's create **one individual card.** Any guesses how we might do that?

... By breaking it down again! The card here has an image, a title, a description, and some badges with technologies used. When you think about it, that's just a few boxes and some text. That's way more doable for us!

Now we've gone from the behemoth problem of "Create a portfolio website" down to 4 simple subtasks:

1. Add an image
2. Add a title
3. Add a description
4. Add technology badges

Each of these tasks can be completed quickly, and in isolation. When they're done, you'll have a project card. When you replicate that, you have a completed "Projects" section. Way to go! Now we just repeat the process for the next section. And before we know it, we have a complete and functional website.

I can't stress enough how many times this mentality has helped me complete projects I would've otherwise considered impossible. It takes some practice, but when you get comfortable with this mindset, you'll be unstoppable.

And it's not just specific to front-end development. The exact same concepts apply to any coding related field. "Divide and conquer" is another name for the same strategy, and you can use it when designing algorithms, implementing APIs, or analyzing big data - basically anytime you have a problem to solve.

Practice consistently

The best way to get good at any skill or craft, is to practice. It's the only way to gain meaningful experience that will improve your problem solving, speed, and confidence.

Practice will allow your muscle memory to take over mundane tasks, like navigating around your codebase and typing code on your keyboard. It will allow your brain to focus all of its creative energy on solving the problem at hand. The more you practice, the more problems you will solve - further increasing your problem solving abilities.

Finding the time to practice consistently is not always easy, though. Maybe you're working a different job, maybe you got a wife or kids, or maybe you just have a lot of stuff going on. Whatever the reason, it's important that you build the habit of practicing a tiny bit each day. To accomplish this, I recommend making a promise to yourself:

I will code for 5 minutes every single day.

Building this habit can be tricky in the beginning, but you'll find that 5 minutes is the perfect amount:

- You can always find time for it, every single day
- It'll get you sitting in front of your screen with an open code editor
- Your brain will enter "code-mode" every day

The beautiful part of this, is that the 5 minutes is just a trick. Almost every time you do this, you won't want to stop after just 5 minutes. We've used the small timeframe to trick your brain into **starting**, which is the hardest part of doing anything. Once you've started, you're free to code for as long as you want.

If you're having a busy day, stop after 5 minutes. If you're getting in the zone and feeling productive, keep it going! Practice for as long as you feel like. The more you practice, the faster you'll learn. If you get stuck on a problem and start to get frustrated, stop for the day - you'll be back tomorrow with a clear mind for your next 5 minutes.

This is the single most efficient trick I use to improve my coding skills. Building this habit will train your brain in all the right ways, and you'll quickly realize the power of consistent practice. Dan Abramov sums it up beautifully:

Build projects

We talked about the importance of consistent practice, but now you might be wondering what to practice **on.** What do you actually **do** during your daily practice time? If you're just starting out on your coding journey, it's tempting to follow a lot of tutorials. They show you what to code, they teach you the concepts, and you get to code along with them. Sounds great, right?

The problem with tutorials, is that most of the knowledge **doesn't really stick with you.** You feel productive because you're writing code. But if the concepts you learn don't get time to manifest in your brain, it's not worth much.

The best way I find to gain **real, meaningful practice** and really improve your skill set, is to build projects on your own. There's a lot of great resources with different ideas for projects you can build, my favourite being these 2 lists (both by Florin Pop):

15 App Ideas to Build and Level Up your Coding Skills
App Ideas Collection

If you're feeling adventurous, there's also a list of public APIs that you can use and build on top of - the sky's the limit! Finally, check out this article for alternative things you can do, like building websites for non-profits or contributing to open source.

One of the biggest benefits of building real projects, is that you have something to show when applying for jobs. The ability to take a problem description and turn it into a fully-fledged solution, that can be shared online, is a great skill to have. Companies are always looking for people that can take a project from it's early stages all the way to completion. It's also a great conversation topic during interviews, where you get to talk about how you implemented certain features and why.

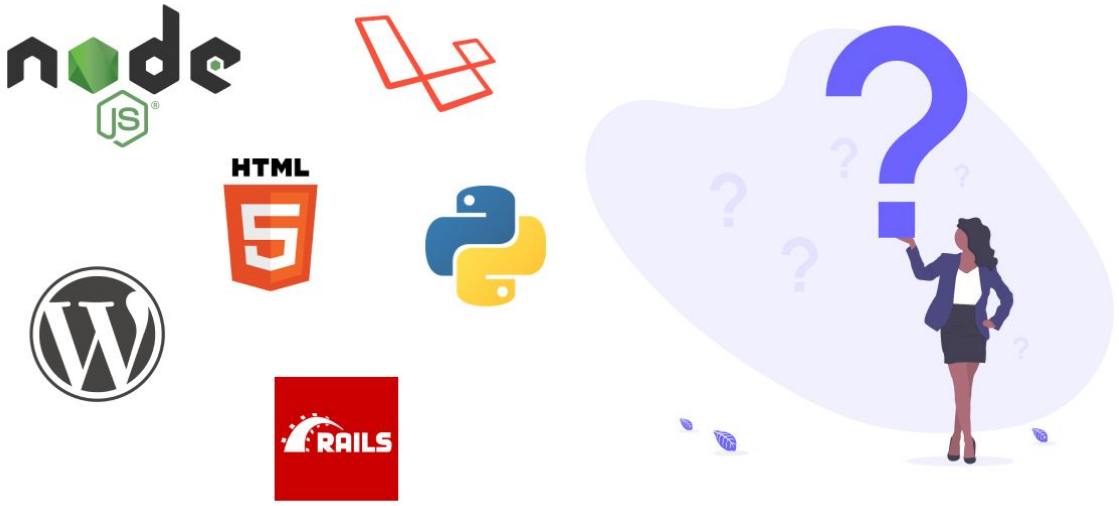
All of this is not to say that you should stop doing tutorials at all. Tutorials are great to learn the basic syntax of a language, or the overall idea behind a library. Tutorials are also nice if they show you how to implement a certain isolated feature, like an animation or an algorithm. The problem with tutorials is that nothing ever goes wrong in them. And that's not what happens in the real world.

To get around this, try to take the idea behind a tutorial, and **adapt it into something else.** For example, if a tutorial is teaching you to get some data about books from an API and display those in a list, switch it up! Use another API from the list above, and create a list of movies instead. This is especially useful as a beginner, when creating an entire project from scratch can be challenging. Modifying tutorials tends to be easier, and you'll still learn valuable lessons by changing the topic, adding features, and running into problems.

By running into problems, either by modifying tutorials or creating your own projects from scratch, you'll learn how to actually solve those problems. **And that is the real key to learning development.**

Narrow your focus

Coding is one of the broadest industries in the world. If you're a developer, you can be working on any range of problems: From back-end, to front-end, to data science, to mobile, to Cloud, and to all the different languages like JavaScript, C#, Ruby or Python. To all the frameworks and tools like Wordpress, Express, Ruby on Rails, Laravel or Django.



Then there's all the stuff **surrounding** code: UI/UX design, copywriting, database management, deployments, DevOps etc.

Don't feel bad if you get overwhelmed by this list - so do I! And that is not even **close** to covering all the topics, languages and tools surrounding development.

The point is: you will never be able to learn all of these things. You **have** to decide which path you want to go down and specialize in. Whatever this may be, will depend on what you find the most exciting and fun to work with. I'm a front-end developer, so my focus is HTML, CSS and JavaScript. On top of this, React is my go-to framework. These are the languages and tools that I specialize in.

It's great to have a basic understanding of other areas in our profession, what they are, and how it all fits together - but expecting to be an expert in multiple will only set you up for failure.

It'll also prevent you from diving deep on any one particular area. It's too easy to learn the basics of JavaScript, then get distracted by the hard parts, and think that Python will be much better for you. Then, when you've learned the syntax of Python and that starts getting hard, you switch again.

Take some time to explore your options before you decide what to focus on, especially as a beginner. You obviously still **can** switch later if you really want to. Just remember that the grass is not always greener on the other side.

Your goal should be to get really good at your niche, and use that knowledge to build projects. This will be more fun, better for your career, and you'll have a much easier time getting hired or finding a new job. Play around with other tools, sure - but remember your focus.

Learn language fundamentals before frameworks

Whatever language you choose, you'll need to get familiar with the libraries and frameworks of that language at some point. Frameworks are great, because they help you get productive quickly. They make sure you don't have to reinvent the wheel, but can use what smart people have built for you.

The issue with frameworks is that it can be hard to understand **what is really going on.** Frameworks tend to abstract a lot of stuff away from you, which is a good thing when you're experienced, but can be a big downside when you're learning.

Another problem with frameworks is that they can quickly shift in popularity. Some years ago, jQuery was the most important tool you needed for front-end development. Now, frameworks like React and Vue are taking its place. It's impossible to predict how this will change again in the future. As a beginner, you need to **focus on the fundamentals of your language.**

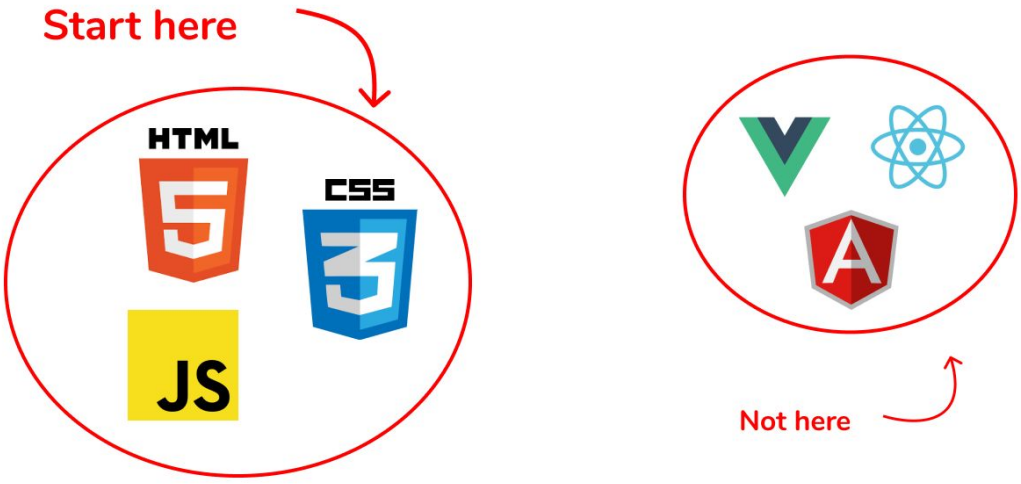
For web development, these are HTML, CSS, and JavaScript. Whatever happens to the ecosystem in the years to come, these skills aren't going anywhere. They are the fundamentals of the web platform, and everything else is built around them. Having a strong knowledge of

these 3 languages will serve you well in the long run, and make learning any new framework much easier.

All of this is not to say that you need to know every single HTML tag before moving to CSS. And you certainly don't need to learn every single CSS property before starting with JavaScript. What you need, is to understand the basics of all 3 of them, and how they fit together.

This example is not unique to web development, either. If you're a data scientist, having a strong Python knowledge will help you adapt to any new number-crunching framework or library that comes out. You'll also have the side benefit of being able to transition to multiple other fields if you so choose. And as a back-end dev, knowing a language like PHP or Ruby in depth, will allow you to quickly learn frameworks like Laravel or Ruby on Rails without sacrificing your flexibility to learn something else down the line.

Fundamentals are what will stand the test of time. Get good with those, and you can always adapt and learn frameworks on top of them.



Problem solving

When we work as developers, our job is actually **not** to write code. As counterintuitive as that may sound, our real job is to solve problems. On a broad scale, we'll be solving problems for users and businesses. And on a smaller scale, we'll be solving concrete coding problems.

That's why **learning how to problem solve** is one of the most important skill sets in our developer toolbox. To solve problems efficiently, we first need to break them down as much as possible, as covered earlier. Then we can solve each problem in isolation.

A great way to practice problem solving is by using online platforms designed for this particular task. I personally really enjoy edabit.com and leetcode.com, but you can pick whichever platform you prefer. When you tackle a problem, I recommend going about in a very pragmatic way:

1. Read the problem description **thoroughly,** to make sure you understand exactly what you are trying to do.
2. Solve the problem in the most **naive, brute forcing** way. Don't worry about performance or code readability just yet.
3. When you have it working, try to think of a better approach. Can you make the code easier to understand? Can you make it perform better? Of course, while making sure your solution still works.

4. When you've implemented the best solution you can think of, move on to the next step. This is where you can usually see **other people's** solution to the same problem.

5. If yours is identical, great. You found the best solution. Most of the time, this is not the case though. Chances are that somebody has made a better/cleaner/easier solution. Read through that solution **carefully**, to make sure you understand what is going on, and why it is better than yours. When that solution makes complete sense in your head, you're probably tempted to move on to the next problem. DON'T DO THAT. Instead, do the final step, which is incredibly key to improving your problem solving skills:

6. **Go back and redo the problem.** Pretend you are trying to solve the problem for the first time (remove your old solution if it's still there). Now, try to implement the solution you just looked at **from memory.** Don't copy/paste the code, but actually try to solve the problem using the technique you've just seen as the best solution. If you can't get it to work, take another look at it - but **keep trying** until you've implemented the recommended solution yourself, from scratch.

I found this method exceptional for improving my own problem solving abilities. It works well because you get to solve the problem in your own way first, and **then** see the "right" way. When you force yourself to also **implement the right way**, instead of just moving on, you start to notice what actually makes a good solution, well, good! And you'll learn a ton from doing it this way.

Solve the problem before you write the code

Of course, this tactic is mostly useful for dedicated practice problems, where you can see the correct solutions. This is not the case in actual project work that you're doing every day. Then you'll have to rely on your own problem solving skills (for things you can't Google, at least). And when you encounter one of those problems in your own work, it's tempting to jump straight into your favorite editor and start hacking away at it. We're programmers after all - that's what we do, right?

Not quite. By jumping straight into code, you quickly end up trying stuff to solve the problem at random. "Oh I'll just write this code and it will work. It doesn't? Let me try this other thing". This can keep going for a while, and you might even find the right solution - but it won't be the **fastest** or **best** way to get to that point. And you might not truly understand it.

Instead, try to solve the problem in your head or on paper, **BEFORE** you write a single line of code. Start by just thinking logically, the same way you did on the practice problems:

How can I achieve what I am trying to do, in the most naive way possible?

Example: Let's say you need to find the largest number in an array. You might start thinking "I probably need to iterate through the array, keeping track of which number is the largest so far. I'll have a starting

variable set to 0. If I get to a bigger number, I set that as my current maximum and move to the next value”

This is a naive solution - it’ll work for positive numbers. If you went ahead and implemented this right away, you might never think twice about it.

But if you think through it logically, what happens if you pass an array with negative numbers to this function? It’ll report 0 as the largest - **which is wrong.**

Thinking through these edge cases **before writing any code** helps to isolate them, and spot potential bugs **before** they occur. Now we get to think of a better solution, like setting the first number in the array as our starting value - which will make our function work on arrays of both positive and negative numbers.

By solving the problem on paper, you also get a much better understanding of what is happening. You avoid trying things at random (which is much easier to do in code) and instead solve the problem logically. When you’ve got the solution ironed out, writing the code for it becomes a breeze.

As with any rule, there are exceptions. Some problems are trivial to solve, in which case, you don’t need to stop yourself and write it down on paper beforehand. There are also times when just trying stuff out is actually useful, especially if you’re working with creative or visual

elements, where you need to **visually see** the result to judge if the problem has been solved. In that case, hack away. But I find it helpful to take any problem that is even remotely complicated, and work through it on paper before writing the code.

Reading code

When you work as a developer, you're going to be spending a lot of time **reading code.** You will probably spend more time reading code than actually writing it. So learning how to properly read code is a key skill to master. It's not always the most fun, but it's **necessary** and can be learned quite easily with practice.

Reading code is about your understanding of what a particular piece of code is doing. There's several different scenarios where this proves useful:

- Understanding code from a tutorial so you can modify it to your needs.
- Trying to familiarize yourself with a new codebase at work or in open source.
- Debugging 3rd party code from a dependency in your project
- Understanding what on Earth is going on in the code you wrote last night, or last week, or 6 months ago.

The last one will probably be familiar to you. Looking back at older projects, it's impossible to remember how the code was working. Knowing how to effectively read code will help you understand what the code is meant to do, and it'll be easier for you to modify it or fix bugs.

3 tips for getting better at reading code:

1. Try to **understand what the author was going for.** What is this piece of code trying to accomplish? Why? There's probably a reason it was written this way. Try to uncover that.
2. Read it **line by line**. If you get to a part that exists in another file, follow the trail. Make sure you understand what each part of the code is supposed to be doing.
3. **Do it often.** Make it a habit to properly read through 3rd party code that you use in your project. You don't need to read every single library, but if you copy a solution from StackOverflow, read it thoroughly to understand how and why it works. Add a few comments if you're unclear about something. That'll make it so much easier if you need to revisit the code later.

Writing code

If reading code is something every developer does daily, it stands to reason that we should **write readable code.** This will help ourselves now, ourselves down the road, and anyone else who might come in contact with our code. Writing readable code is also extremely important for **maintainability.**

Think about any Software as a Service company, or really any company that has a digital aspect to it. A large part of their business revolves around the coded solution, and being able to adapt to changing requirements and new business opportunities. This is nearly impossible if the code is not written in a maintainable way.

Writing readable code is a skill that develops over time, but there are some general tips you can apply that will make your code much easier to understand:

- **Use descriptive variable names.** Don't use names like "a", "x" or "test". They don't convey any meaning, and make it harder to understand what the code is doing at a glance. Instead, use names that describe what the variable represents. For booleans (true/false values) you can almost always add "is" in front, to show that it can be in only 1 of 2 states, like "isLoggedIn". For strings and numbers, just think what would be the easiest for someone else to understand. Something like "imageUrl" immediately tells

the reader what the variable contains, without them even having to look at the value.

- **Split code into several functions.** Each function should do 1 thing only. If you find yourself with a large block of code that is doing multiple things, split it up into smaller functions. Give those functions proper names too, to describe what each of them are doing. A function name like “bookTable” (in a restaurant booking system) conveys a lot more meaning than something like “submit”.
- **Don’t create fancy one-liners if the code is easier to read as several lines.** See the example below. Which one is more readable to somebody who **has no idea** what this code is supposed to do, and is maybe just seeing it for the first time?

```
// Which one is more readable?

if (user.isLoggedIn) {
    // Show the user dashboard
    goToProfilePage()
} else {
    // User is not logged in, go to login page
    goToLoginPage()
}

/* ------------ */

if (u.logged) go("profile")
go("login")
```

- **Write comments**. In some situations, the code is not enough to explain itself. Proper naming and short functions help, but sometimes you end up with a block of code that is just impossible to read for whatever reason. Write a descriptive comment, clearly explaining what the code is doing in a few sentences at most.

Sometimes, a single line in English just does a better job at explaining what the hell is going on, than the code itself can possibly do.

When you employ these strategies, the time saved when you (or someone else) needs to read your code will far outweigh the extra time spent on writing comments and using descriptive variable names. There's obviously a lot more to writing "clean code", but you'll learn most of that as you go along. Especially when you experience poor code yourself.

Understand concepts & patterns

The ability to understand higher-level concepts and notice patterns is a great skill for developers to have. If you think about it, a lot of stuff that we do is similar between projects. Recognizing these areas, and creating the proper abstractions for them, will help you quickly get started with new projects.

CRUD

For example, a lot of front-end applications evolve around reading data from an API. It might be the weather, movies or Pokemon - the idea is the same. What do we usually do with this data? Display it nicely to the user, in some form of list or grid. This is called **reading data.**

If we want to add another layer, maybe we allow the user to **create** objects - like todos, blog posts, or tweets. And if the user can create objects, they should be able to **update** those as well. It should also be possible to **delete** these objects.

What we've described here, is a **CRUD application.** CRUD stands for **C**reate, **R**ead, **U**pdate and **D**elete. It's a fundamental idea, present in nearly all web-apps and software you'll encounter.

Understanding the basics of a CRUD app means that you can transfer this knowledge between projects.

Forms

Another common pattern we see on the web is the use of **forms**. Forms are a simple way to get user data/input, and do something with it. Except, forms aren't simple.

There's different types of form fields (inputs, checkboxes, radios etc.), there's validation (how do we make sure the user input looks correct?), error messages (how do we tell the user if they messed something up?) and where do we even send the data when the form is filled?

This might seem overwhelming, but again, once you **understand** these fundamental ideas behind forms, you can apply them to any project.

Spend some time familiarizing yourself with high level patterns - it'll help you break problems down and **apply general knowledge** to a **concrete problem.**

Engineering principles

In the development industry, you might hear of certain “software engineering principles”. A lot of these tend to be very technical, like the SOLID principles - Single Responsibility, Open/Closed, Lisskov Substitution, Interface Segregation, and Dependency Inversion.

These principles are great, and have been carefully thought-out over many years. But in my opinion, they’re also hard to relate to. What does interface segregation even mean? And what does it have to do with this React component I’m writing, or this iOS app I’m working on?

Instead of busting your ass trying to remember these principles that you might never end up using, I prefer to use some of the simpler ones that I find more applicable to everyday coding:

YAGNI (You aren’t gonna need it)

The idea behind YAGNI is that “you should not add functionality until deemed necessary”. It stems from the fact that a lot of software becomes bloated, or worse, never gets shipped because there’s always more features to add.

Whenever you’re about to add a piece of functionality to your project, try to think if this feature will **actually** benefit the user - or if it’s just a fancy idea that someone came up with. It always works to think if you

need to implement a certain feature **right now**, or if there's just a chance of you needing it later. If the answer is later, then wait.

That doesn't mean you shouldn't plan ahead. You always want to write your code in way that makes it possible to add stuff and modify it later. Just don't optimize for a specific case that might never be reality. You'll waste time creating features that will never be used.

You aren't gonna need it.

KISS (Keep it simple stupid)

Following YAGNI is another one of my favourite principles. KISS helps you answer which direction you should take a certain feature, or design, or idea. By forcing yourself to always keep it simple, you'll end up with systems that are easier to use and has less bloat. It helps to avoid unnecessary complexity that evolves from convoluted design.

As developers, keeping it simple can be hard. We often want our solutions to be fully-fledged, to have lots of exciting features and generally just be awesome. We want to show our skills, and that we can create great systems.

But making systems simple is actually the best ways to make them awesome. Simple systems are easier to use, and avoids many pitfalls of a complex system.

Keep it simple stupid.

DRY (Don’t repeat yourself)

If you’ve been coding for a while, you might have heard of DRY. “Keep your code DRY”. The idea is to avoid copy/pasting code, because copy/pasting leads to inconsistencies and maintainability nightmares.

If you find yourself copy/pasting code, chances are that you need an abstraction. A common scenario is if you wish to make a global navigation across all the pages on your website. In pure HTML, with 4 pages, you would need to copy/paste the navigation code 4 times, once for each file. If you then wished to change one of the navigation items, you’d have to change it in 4 places - or the result will be inconsistent.

In this case, you would need something like PHP or a JavaScript framework to simplify the solution into 1 navigation component, that would keep all of your navigation logic. Any updates here would propagate to all the pages - and you’ve effectively DRY’ed up your code.

Don’t repeat yourself.

WET (Write Everything Twice)

DRY is a great principle, and one you should aim to follow. But by being too DRY, you also end up with premature abstractions that create more harm than good. This is where the WET principle can apply (while also being a clever opposite to DRY).

WET is the idea of writing everything twice before you try to create an abstraction. In the navigation example, this would mean copy/pasting the navigation once. If you only have 2 pages, this is fine. You've effectively written the same code twice, but chances are that it will save you more time than thinking of a way to abstract it. Basically, **you aren't gonna need it** (the abstraction)**.**

Until you add more pages, in which case the duplication starts to become a problem. The risk of forgetting to change code in one of the places goes up, and now it might be time to create the abstraction and DRY up your code. But only after you've made it WET.

Write everything twice.

Perfect code

There's this idea of writing "perfect code". Code that is self-explanatory, concise, maintainable, looks good, and has 0 bugs. As a beginner, it's easy to feel that the code you're writing is the opposite of this. You might be wondering about best practices, and always having the feeling that your code could be better.

The problem is that perfect code doesn't exist. Yes, some code is more maintainable and easier to read than other code. But as a beginner, this should not be your biggest worry. The code doesn't need to be maintainable if you're creating a TODO app as a learning exercise. Of course, you should still strive to write readable code, as discussed in a previous chapter. But **writing bad code is one of the ways we learn how to write good code.**

It's hard to judge a piece of code objectively as a beginner, to decide if it's "good" or not. This is expected - you're still learning. The only way to realize what code is bad, is to actually face the problems that bad code creates. If you try to write your TODO app in a "perfect" way, you will get hung up on technicalities that are way less important than actually building and completing the project.

Make the mistakes. Write some "bad" code that is hard to read, or breaks in unexpected ways. These problems will make you realize what you need to do to improve your code in the future.

Using third party code

When you let go of the idea of perfect code, it also opens up the possibility of using code that other developers have written. This can be from StackOverflow, GitHub or any third party. Using these libraries and code snippets **is completely fine**. You don't want to reinvent the wheel and spend time solving problems that others have solved in better ways than you. The exception being if you're writing something as a learning process. Creating a TODO app or trying to write a simple HTTP server yourself is a great way to learn. In this case, don't worry that others have done it before.

Copy/pasting solutions from the internet and installing dependencies is a natural part of being a good developer. It frees up your time to solve the more interesting problems that are relevant to **your** users and customers. Just make sure that you **properly understand** code that you are copy/pasting. It's very likely that you'll need to adapt it to your project, either now, or further down the road. You won't be able to do that if you blindly copy it and pray that it works.

If you find a code snippet online that you need, I recommend writing it out line-by-line yourself. You'll be forced to think about **what** this code is doing. It won't take long, but it'll help you understand the code and make it easier to modify. It might also surface problems that you didn't expect. Good developers understand **what code to borrow**, and **what code you're better off writing yourself**.

Staying level-headed and pragmatic

Learning how to debug issues, without losing your temper, is one of the most important parts of being a developer. It's easy to get frustrated by code. In your head, you know what you **want** to do. But the code is just not working. There can be a million reasons why.

In these situations, it's important to take a pragmatic approach and look at the issue logically: The code is doing **exactly what you are telling it to do.** That's the only way computers work. So if the code is not doing what you **want** it to do, maybe your mental model is wrong.

This often happens if you're coding in a language or framework that you're not familiar with, but it's also possible that you've just been blindsided. When we work on the same problem for a long period of time, our brain starts to assume things that might not be true. There's a few good ways to deal with this:

- **Read through your code line by line**. You might be overlooking a simple thing that you're not thinking about. Go into each function and make sure it looks the way you intend.
- **Ask for a second pair of eyes.** Sometimes you miss obvious things when you're too close to the problem. Asking a co-worker or someone online for help can often point out the mistake.

- **Talk through your solution out loud (rubber duck debugging).** This is one of the oldest tricks in the book. It involves forcing yourself to **explain** your code to an inanimate object (like a rubber duck - but you can also use a friend here instead). The act of **actually speaking/articulating** your solution often leads to an “aha” moment, where you realize the fault in your logic.
- **Take a break.** Sometimes the most effective debugging tool is a breath of fresh air, a quick bite to eat, or (my personal favorite) a **good night's sleep.** Giving your brain time to rest, and coming back with a refreshed mind, will allow you to approach the problem logically. You’ll see important details that you missed the day before. And problems that kept you stuck for hours earlier, are suddenly solved in 5 minutes.

The beauty of breaks and sleep, is that your brain will continue to process the problem in the background. You’re not actively thinking about it, but your brain is subconsciously working the problem, while still giving you time to reset. Taking breaks is, by far, the best debugging technique available to us. **Use it often.**

One final tip, that will save you a LOT of frustration: **Read the error messages!** It’s far too common to just see an error, think you know what the problem is, and then making a change that might have nothing to do with the actual error. That’s not pragmatic. If our goal is to find and fix the issue as quickly and efficiently as possible, we should start at the root of the issue. The error message will tell us what that is.

Some errors are easy to understand, while others are more cryptic. Either way, a quick read-through of the error might prevent you from setting down the wrong path, and will often present an easy fix. And if it doesn't, Google away. Chances are that somebody else has encountered this error before you.

PART 2

Productivity

Learning how to learn

Development is one of the fastest changing industries on the planet. There's always new languages, frameworks, libraries or tools coming out. You're not expected to know all of these, but at one point or another, you **will** have to learn something new. **Learning how to learn** is the art of being able to quickly pick up new skills as they come along. This will be useful throughout your entire coding career, and it's the absolute fastest way to increase your productivity.

Firstly, you need to figure out how you learn best. Some people learn best from reading books, which tend to be very in-depth but can also become outdated quickly. Others learn best from video, appreciating the visual elements and being able to follow along. Some learn by reading documentation and source code. And others learn best by just jumping into the code and failing along the way.

There is no right or wrong way. It's all individual. What matters is that you find the style that works for you. It might even be a combination of all of the above. For me, I prefer watching videos to get the overall idea behind a new framework or library. Then I'll jump into the code and experiment, consulting the documentation along the way. This works best for me, but might not work best for you. You need to find your own way.

Focus on learning one thing at a time

Whichever way you prefer to learn, it's key to narrow your focus to 1 thing at a time. That's why I recommend front-end developers to start with the basics of HTML, then move to CSS, then JavaScript. If you start with something like React right away, you're learning too many things at once. You won't know where the JavaScript ends and the HTML begins.

If you want to learn React, make sure you understand the others first. Then, focus **only** on learning React. Not NextJS, not Styled Components, and definitely not webpack. Learn each of those in isolation. Trying to learn too many things at once leads to confusion, and the skills will start to blur together. You'll miss the fundamentals of the thing you're **actually** trying to learn because there's too much stuff going on.

Learning the thing

When you've decided on a particular thing to learn, it's time to put in the work. Whatever your method is, it will take some time. There's no shortcuts, only advice that might speed you up:

Grind it out. When you're learning, you need to get your hands dirty. You need as much practice as you can get, and as many hours as possible to allow your brain to obtain the information. Write as much code as possible.

Make mistakes. By coding a lot, you will make mistakes. This is good! By making mistakes and figuring it out why, you're learning the

ins-and-outs of what you're trying to internalize. And when you're learning, you won't need the code for anything else, so it doesn't matter that it's bad or has issues.

Ask questions. When you get stuck on something, ask around! Either in person at your job, or online. There's helpful people all around you. Use their knowledge to your advantage.

Sleep. This might seem weird, but sleep is one of the best teachers that exist. When you sleep, processes are happening in your brain that moves information you've just learned from your short-term memory, onto the "hard drive" that is your long-term memory. By sleeping, this process also frees up your short-term memory to learn even more on the following day. It's why students who cram everything before an exam tend to forget it all a week later. They don't allow the process of sleep to do its magic.

Take notes. I find it helpful to write stuff down when I'm reading or watching a tutorial. It forces me to think actively about a concept in order to put it into words. You can always refer back to them later if you run into the same problem.

Mentor others. Especially if you're not an expert. When you've just learned something, you're in the unique position of being able to **understand** the problems that others are facing. You know the problems they might encounter and the issues they will run into, because you were in their shoes not long ago. This makes you super qualified to

teach what you've just learned to the next person. And as a side benefit, you'll learn it better by forcing yourself to teach it too!

Learning how to learn is one of the best investments you can make in yourself, and it will help you throughout your entire development career. I guarantee it.

Entering “flow” state

“Flow is the mental state in which a person performing an activity is fully immersed in a feeling of energized focus, full involvement, and enjoyment in the process of the activity.”

You’ve most likely experienced this before while coding. You’re completely focused, time stops moving, and all of a sudden you’re done with that feature you’ve been working on.

It’s an incredible feeling. Unfortunately, it’s not always easy to reach. “Flow” is very fleeting, and even the smallest distraction can take you out of it. That’s why I recommend 3 ways to help you focus:

1. **Set time aside for coding where you know there will be no distractions**. It can be early morning or late at night. Just make sure you won’t be interrupted.
2. **Leave your phone in another room.** It’s too easy to pick it up and check if something happened when it’s right there on the table.
3. **Decide on 1 particular task you want to accomplish.** We’re trying to get you in the zone, and we can’t do that if there’s too many problems to solve at the same time.

Learning how to manage and enter “flow” state on command is one of the best ways to increase productivity. You’ll also get more confident because of the speed you can achieve while churning out projects.

Pomodoro

The Pomodoro technique is a useful way to increase your focus and productivity. The idea is simple:

- You set a timer for **25 minutes,** where you do **uninterrupted work.**
- When the timer goes off, you take a break for **5 minutes.**
- After the break, you work another 25 minutes, etc.
- After a few cycles, you take a longer break of 30 minutes - enough to go for lunch or a walk.

Pomodoro helps you to really focus on the work during the 25 minutes, because you know a break is coming up soon. So you’ll wait to check your phone, or run to the bathroom, until it’s break time.

The beauty of the 25 minutes is also that you’ll often stop in the middle of something - making you eager to get back to it after the break.

There are plenty of online tools to help you keep track of the time. I use this one: https://tomato-timer.com/. Feel free to play around with the intervals too: If you find that 35 minutes of work with 10 minutes break works better for you, use that instead. Just don’t keep the working

period too long, or you'll get distracted midway and the technique loses its potency.

Habits are the key

I'm a huge believer in habits as a way to learn any skill or improve any craft. Programming is no different. But how do we define a habit?

A habit is an action that you perform per reflex. Something that doesn't require a lot of thinking or planning. Some examples of things you might be doing habitually are brushing your teeth in the evening, checking your phone first thing in the morning, or smoking. Naturally, some habits are healthy and great - like brushing your teeth or working out. While other habits are damaging - like smoking or eating unhealthy foods.

As developers, we want to develop habits that help us improve our programming skills and productivity. We also want to develop habits that make us more hireable, and that makes it easier to land a development job. To do this, we need to understand how habits work.

The stages of habit

There are 4 stages to any habit:

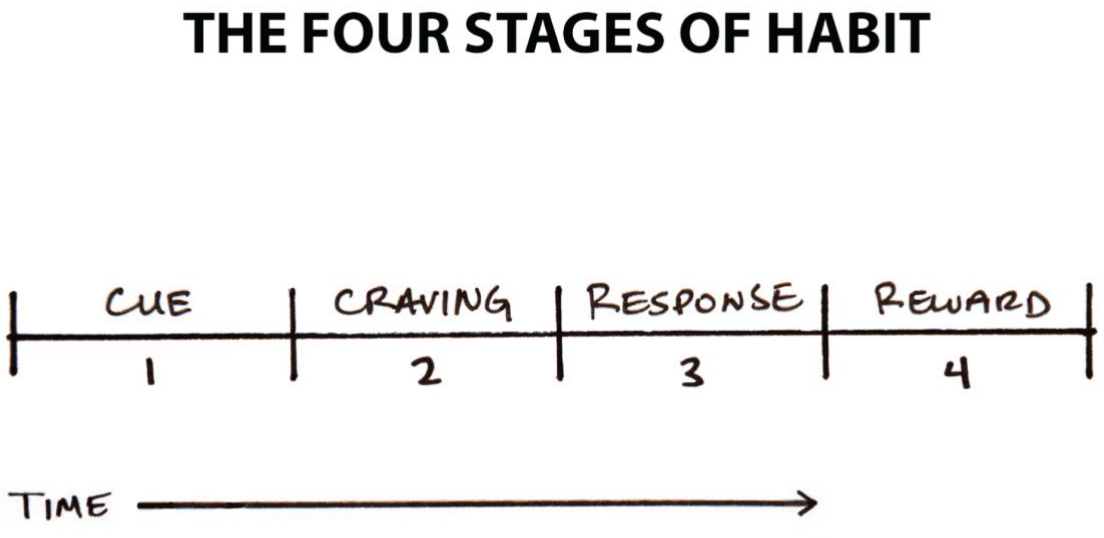


Image courtesy of https://jamesclear.com/three-steps-habit-change

The **cue** is what triggers the start of your action. This could be a cookie left on the table, turning off your computer to get ready for bed, or the Twitter icon on your phone's Home screen.

The **craving** is what the cue makes us want. This is a motivating force that prompts us to perform an action. We might crave the sweet taste of the cookie; the freshness of a clean mouth; or the dopamine released by seeing our number of Twitter notifications.

The **response** is the literal action that we perform to satisfy our craving. In these examples, that would be eating the cookie, brushing our teeth, or launching Twitter. This is done automatically, because our brain has

noticed the craving, and naturally wants to perform the action to satisfy it.

The final step is the **reward.** This is where we get rewarded for the response that was triggered, through a tasty cookie, a healthy feeling in our mouth, or our desire to be socially accepted.

When we understand how habits work, we can start using them to our advantage. In fact, we already did this in the chapter on **Practicing consistently.** If you recall, we used the "5 minute rule" to get our brains into code mode. We're trying to build the habit of coding every day:

Cue: An open code editor
Craving: A desire to learn and improve our skill set
Response: Coding
Reward: We're better than yesterday

Other habits worth creating is around areas like **reading code**, how to enter **flow state** and **building projects.**

Habits like this allow us to improve ourselves by 1% every single day. In the long run, that leads to a **37x improvement** over the course of a year.

$$1.01^{365} = 37.18$$

The math checks out.

Create quick feedback loops

Feedback is key to improving yourself, your code, and your software as a whole. Without feedback, you'll spend a lot of time guessing and trying stuff at random. You won't learn what really works. The only way to do that, is to create **feedback loops**.

A feedback loop is how you receive feedback, and how often. As a developer, there's two feedback loops of particular interest:

1. A personal, user-focused feedback loop. This is where real people/users can play around with your solution and provide feedback.
2. A technical, code-focused feedback loop

We'll focus on the latter for now.

A technical, code-focused feedback loop is how quickly you can get feedback when you make a change in your code. Depending on your setup, this can look very differently:

- **Hot Module Reloading** instantly reloads your development page every time you save your code. The feedback is near-instant. This is the experience you get with modern frameworks like React or Vue.

- **Refreshing the page manually** after every code change you make. A bit slower, since you need to tab out of your editor, into your browser, refresh, and wait for the page to load.
- **Saving, but having to wait for your build tool.** You might have experienced this if you use something like Webpack or compile your SASS. There's a delay, anywhere from milliseconds to several minutes (rarely), depending on your project size. This can really slow you down on large projects.
- **No local environment at all.** It's not common, but I've worked on projects where code needed to be **deployed** to see the effect. This means dragging and dropping files through FTP. It's painful.

By lowering the feedback time after each change, you can **severely** increase your productivity. Coding is an iterative craft, so you need to be able to change your code and view the changes quickly.

The fastest feedback loop you can create is with the browser's developer tools. This allows you to change values and see them update in real-time. It's a good idea to play around with a lot of different CSS properties here, to see what looks best to you.

Done is better than perfect

One of the greatest things about software is that it's never finished. There's always more features to add, bugs to fix, or code to refactor. Creating software is an iterative process, where you can continuously improve the software and your code over several years.

Unfortunately, this has some downsides too. Because software can always be improved, we sometimes strive to reach **"perfect"** instead of being happy with **"done".** We keep thinking that our software is not good enough, and that if we had juuust a bit more time, everything would fall into place.

This is a dangerous line of thinking. **Our code is not helping anybody before it goes live in the real world**. If nobody is using our code because we keep trying to improve it, we're not **creating value** and **solving the problems** that we designed our software to solve.

Embrace the process of creating, and always strive to do better. But don't let the idea of perfection stop you from shipping your code. Get it out in the world as soon as possible, take feedback, and **then** improve your solution. It's much better than going back and forth between technical implementation details or other factors that users don't care about.

Create your systems

I love systems. Systems enable us to do incredible things that we never thought were possible. And systems can help us to learn or improve any skill, especially development.

You might be wondering what I even mean with “systems”. Earlier, we talked about dedicating 5 minutes to sit down and practice code every day. That’s a system. It’s a clearly defined procedure, an organized method for getting something done.

Another system you might consider, especially as a junior developer, is how much time you spend on a problem before asking for help. There’s a fine line here. You don’t want to ask for help every 3 minutes, before exhausting your own ideas and different possibilities. But you also don’t want to wait several days stuck on a problem that someone else could help you solve in 5 minutes. So an example system could be to try to solve a problem for 30 minutes, and then asking for help if you haven’t made any progress.

You’re also going to need a system for learning new things. If you’re working and feel comfortable with a framework like React, how much time do you spend learning things **other** than React? You probably want to spend most of your time getting better at React itself, but it’s also useful to have a look around and experiment with alternatives.

Try to think in systems rather than goals. Having a goal like “I want to land a full-time development job” is a nice idea, but it’s not actionable. The goal itself doesn’t help you achieve it.

What does help you achieve it, though, are your **systems** - like practicing 5 minutes per day, asking for help after 30 minutes of being stuck, and spending 2 hours per week learning about a technology you’ve never tried before.

These are just examples. Systems are an individual thing, so **create some that work for you.**

Get comfortable with your tools

As developers, there are some tools (outside of programming languages) that we use every day. If we're gonna be spending 7+ hours a day on coding, we should create an environment that we like and that helps us be productive.

IDE

Your IDE (Integrated Development Environment) is your code editor of choice. It's where you'll be writing every single line of code. Spend some time getting used to it and customizing it - you'll be much happier and more productive. Things to work on:

- **Choose your editor.** There are several different ones to choose from, for most, this will be VsCode https://code.visualstudio.com/ - it's the most popular and flexible, and gets continuous updates with great extensions.
- **Find a theme that you like.** There's no reason your theme can't be pretty, with all the time you'll spend looking at it.
- **Learn the basic hotkeys**. Learn to use the keyboard navigation for things you do often (examples from VsCode on Windows): Changing files (CTRL+P), moving lines up/down (Alt+Up/Down) or toggling a comment (CTRL + /).
- **Add your own hotkeys**. Don't be afraid to set up your own hotkeys, or change the default ones. If there's an action you find

yourself doing every day, **make a shortcut for it**. I have one, combined with an extension, that I can use with my cursor in a variable to add a “console.log(‘value:’, value)” with the press of a button. I use it every day.

Shell/Terminal

A “shell” (or “terminal”, or “CLI”) is a way to interact with your computer using text. We’re used to the opposite, navigating and interacting through a User Interface (UI).

As developers, one thing we often do is to install packages of 3rd party code. This is done through your terminal with a command like “npm install *package*”.

Knowing how to use a shell to install programs and packages is a great skill to have. You should also learn some of the default shell commands to move around your computer: commands like “cd”, “ls”, and creating files directly through the shell. You’ll experience a nice productivity boost by spending some time learning basic shell commands and concepts.

Keyboard

We inevitably spend a lot of time on our keyboards: writing code, searching for solutions, asking questions online etc. Learning how to type faster and make fewer errors will not make you a better developer by itself - but it **will** speed up your workflow and make you more

productive overall. Use an online tool like https://www.keybr.com/ to improve your typing speed.

Googling

As developers, we run into problems. It doesn't matter how much experience you have. You'll still be googling for solutions every single day. Knowing how to properly use Google to quickly find the solution to your problem is an incredibly time-saving skill. If you run into an error that needs googling, **read the error carefully.** Remove any reference to your local files, and then paste the error in Google.

Often, appending a keyword like "React" or "webpack" will help you get better searches. So instead of "how to submit a form", go for "how to submit a form React" etc. That'll help you get results more specific to your situation.

Git

Git is software for version control. It will keep track of previous versions of code, when new code is added to a project, and who has written which parts of the code. Git is completely universal in the software industry. I have yet to experience any company or product that does not use Git to manage their code.

Git has a **ton** of features, which can make it quite complicated. But there's really only a handful of features you need to know by heart, because you'll be using them every day. Learn the fundamentals of

branches, adding files, **committing** and **pushing** code. The rest can be learned as you need it later.

Finally: **Git** is the software we use to manage code. This is what we use when we run commands like “git add” and “git commit”. **GitHub** is a website to store and host code. This is where we’ll typically “push” our code to share it with other developers. Know the difference!

Take breaks to avoid burnout

To become a better developer, you need to build your habits and practice consistently. But you also need to be weary of a very real threat in our industry: **burnout.** It doesn't matter how quickly you can learn new things, or how fast you can work, if it leads to burnout.

Burnout is when you overwork yourself, to the point that you strongly dislike coding. The very idea of sitting down and writing code makes you anxious. And if you manage to power through and do some work, you don't feel good about it. Maybe it's too slow, or you just hate every minute of it. All of this can happen to you. I'm not saying this to scare you off. I'm saying it so you can begin to notice the early signs of burnout, and use precautionary methods to avoid it altogether.

If you start to notice any of the signs, take a moment to ask yourself **why** you're feeling this way. Is it just because you're stuck on a hard problem, or can't find the motivation to finish the last 10% of a project? That's fine - it happens to everyone. This is the kind of stuff you just power through.

But there might be a deeper issue. If you feel this way for weeks or months, and the stress begins to build in your body, it's time to take a break. And I don't mean a 5 minute break to clear your head. I'm talking about a serious, several day-week-month long break, to fully reset your mind and figure out what's causing these feelings.

Sometimes all you need is a weekend without coding. Sometimes, you might need a week's vacation. Other times, you might simply be **coding too much** every day (if you work a 9-5, and spend 5 hours coding when you get home, it's probably too much)

Finding the balance between pursuing your dream of coding and improving your skills, while avoiding burnout, is a tricky one. It's very individual, but there's a few important points to always remember:

Prioritize your mental health. You should never sacrifice your mental (or physical) health to improve a skill or for a job.

Taking breaks is completely fine. Take the time you need - your skills won't disappear or become obsolete in the process. If you need a week or a month without coding, go for it. You can also take smaller breaks, by putting a limit on the time you work. Pick a time, like 9-5 for your job, or 8-10 in the evening if you're still learning. And don't do any work outside of that.

You burn out from repeatedly doing things you dislike. If you get burned out often and easily, ask yourself if coding is **really** for you. If your only reason for being a developer is to make a ton of money, burnout will surface quickly. If your motivations for coding are more inline with building stuff, helping people and making a decent living of it, you probably just need a short break to rediscover your passion for coding.

PART 3

Mindset

Dealing with imposter syndrome

“*Imposter syndrome is a psychological pattern in which one doubts one's accomplishments and has a persistent internalized fear of being exposed as a 'fraud'.*"

Imposter syndrome is a huge problem among developers. It is this nagging feeling of not being good enough. That you will never be good enough. And that sooner or later, **someone** will figure out that you don’t know how to code at all - and that you’ve been a fraud the entire time!

If you've ever felt this way as a developer, believe me, you are not alone. It's very common in the industry to perceive yourself as inadequate. Coding can be difficult, and will make you feel stupid when you take a while to solve a problem or fix a bug. This is the root cause of imposter syndrome in developers: "If I'm so bad at coding, why would anybody pay me to do it? It's just a question of time before they find out that I suck". Almost every developer has felt this at one point or another.

My advice for dealing with imposter syndrome is to:

1. Realize that everybody goes through it. I've been coding for years, and I still feel stupid and like a fraud sometimes. It's completely normal.

2. Look at your progress. Chances are that you've already improved a lot since you started coding. Remind yourself how far you've come.
3. Remember that you're never done learning to code.

You need to stop seeing programming as something you either know, or don't know, and instead view it as a continuous journey. You will never be "finished" learning, and you will never be a "perfect" developer. Realizing this is one of the key steps to dealing with imposter syndrome. Every failure along the way, every bug you spend hours fixing or "easy" features that end up taking weeks, is an opportunity to grow.

The other day I spent 30 minutes wondering why my code change wasn't showing up. I was pulling my hair out, trying to be pragmatic, but it just made no sense. Nothing made sense. In those 30 minutes, I questioned my entire life choices. I was positive my boss would show up and fire me on the spot (okay, not really, but you get the idea).

It turns out I was refreshing the live website instead of my local development version.

I felt so stupid. But looking back, it also taught me something: I was oblivious because I hadn't experienced this problem before. And the next time I'm in a similar situation, I'll know what to check first. The experience of failure has helped me grow.

Experience over intelligence

As a developer, you will run into countless problems throughout your career. Some of these will be easy, but most will be hard. Some, you will feel like you're not smart enough to solve. Maybe you've been trying for hours, days or weeks, and you just can't quite crack it. This is where imposter syndrome can **really** rear its ugly head. But when you get to this point, there's one piece of advice that will help you more than anything else:

You need to shift your mindset from "I can't do this, I'm not smart enough" to "I haven't done this before, I'm inexperienced".

When you do that, every problem becomes a matter of whether you've solved it before or not, rather than if you're intellectually capable of solving it (because you are). It's not about intelligence. It's about experience and competence. This shift in your mindset from "I'm stupid" to "I'm inexperienced" leads to the natural solution that you just need **more experience**. Which you get through practice and just **BEING** a developer.

So instead of thinking "I **can't** solve this, I **shouldn't** be a developer", you can reframe your brain into thinking "I **can** solve this, by **being** a developer".

Everybody has gaps in their knowledge, and this is where imposter syndrome strikes. So a good way to deal with it is to get better at **learning new things quickly.** By mastering the art of "learning how to

learn”, you can more efficiently gather new experience that you need to solve more problems.

Another root cause of imposter syndrome, is when you see other amazing developers, in your job or on social media, who seem so much better than you. Then you begin to feel inadequate, that if they’re so much better than you, why would any company choose you over them? More on this in the next chapter.

Don’t compare yourself to others

It’s very easy to fall into the trap of comparing yourself to other developers. Especially on social media platforms like Twitter, with tons of amazing developers that all seem smarter than you. They make more money, they have more followers, and they're probably better coders. This might hurt your confidence, or make you feel inadequate and incapable of even **attempting** to become a developer.

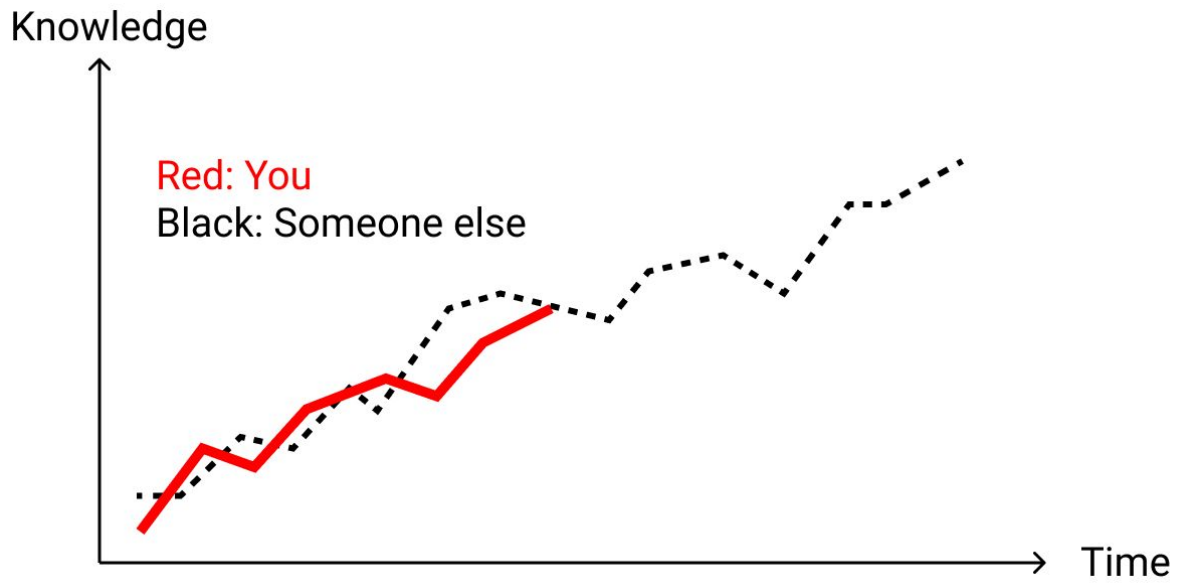
The problem with social media is that you only ever see what the sender wants you to see. You only see the cool CSS art, or the hugely popular blog posts, or the pictures from someone working on the beach.

People tend to not share the hard and frustrating parts of coding that they **100% also experience**. You rarely see tweets talking about the 4 hours someone spent debugging a CSS issue, or the long nights of struggling to implement a small new feature in an iOS app. These experiences simply don’t get shared as often.

It’s also worth noting, that coding isn’t a competition. You’re not trying to “beat” the other developers, online or in your job. Everybody is on their own journey. You don’t know how far someone else is along theirs. So it doesn’t do much good to compare yourself to them. Maybe they’ve been programming for 10 years? Or maybe their brain is wired in a way that they can quickly pick up new languages, but struggle with other fundamental developer skills? There’s no way of knowing.

Focus on yourself

The only person worth comparing to, is your past self. As long as **you** are improving over the course of weeks, months or years, you're going in the right direction. The speed is not important, only the progression. Improvement is not linear, and like I said, everybody is on their own journey. If someone appears to know more than you, maybe they're just further along in their journey than you are. Perhaps they have simply **spent more time** in this industry than you.



Fundamentally, everybody learns at a different pace. And everybody has periods with a lot of growth and new knowledge influx, and periods that feel more stagnant. Some concepts might stick easier to one group of people, but be almost impossible to grasp for another group - and vice-versa. Does this make one of the groups "better"? Absolutely not. And it doesn't matter which group you are in, as long as you focus on improving yourself in whatever way works for you, at your own pace.

Measure your progress

It can be tough to measure your own progress, but it’s important to create confidence and use your improvement as motivation to grow even further. One trick you can use, that has helped me a lot, is to keep access to old projects you’ve built. This can either just be the code hosted on GitHub, or the actual solution available somewhere.

When you're feeling inadequate and like you're not progressing, look back at your older projects and compare them to your newest ones. You'll instantly see how much you've learned, both in terms of the end product and the code quality.

You can also take notes whenever you feel like you accomplished something. It can be something short like “learned how for loops work in JavaScript” or large like “finished my first project”. It’s very easy to forget how much you’ve learned along the way, and looking back at these notes will help you realize it. As a bonus, it also helps to cope with imposter syndrome by literally showing you all the things you’ve accomplished.

Coding is a marathon, not a sprint. You don’t need to become a rockstar coder and land your dream job in 2 weeks. Take it slow, improve a bit day by day, and measure **your** progress. Don’t compare it to anyone else’s.

With that said, it is absolutely crucial that you **learn** from other developers. Looking at other people’s success, and trying to learn how

they got there, is a great way to improve yourself. Just don't be worried that you aren't quite there yet. Learn from them what you can, and use that to progress on your own journey.

You are not your code

As developers, we spend a large amount of our time writing code. It's one of our primary functions (not the only one, mind you. More on that in the chapter on Understanding the Business).

Since we spend so much time writing our code, optimizing it, fixing bugs and refactoring, it's easy to attach the code to our identity. We begin to see the code as part of ourselves, and this can be dangerous. It becomes harder for us to take feedback. If someone criticizes our code, it feels like they're criticizing us. If our code doesn't pass code review the first time, we feel like we **failed.**

This is absolutely, 100% not the case. Some code you write will be good, and some will be bad. That has nothing to do with who you are as a person, and does not reflect on your character at all. You need to **keep your identity separate from your work.** This can be tricky, but it's the best way to see feedback as an opportunity to grow, rather than as a personal attack.

You're not a bad person because there was a bug in your code. If you find it yourself, or if someone points it out, consider that improvement. You've learned something new that you didn't know before. And that other person is most likely just trying to help.

Communication

As a developer, you will be communicating with people every single day, regardless of your field. It doesn't matter if you work in a small team, a large team or by yourself. Communication is an absolute **key** skill to master to have a successful development career.

There are 2 different types of communication, and which one you use most often will depend on your particular situation, but both are super important to know and be good at. There’s written communication, and spoken communication.

For us developers, a majority of our communication tends to be written:

- A job application
- An email to a potential client
- Code documentation
- Internal communication between team members
- Asking for help online

That's not to say you should ignore spoken communication, because it is equally important. Written communication is just where I feel there is the most opportunity to improve for a lot of people.

Below is a list of some of my absolute favorite "hacks" to improve your written communication ten-fold:

1. **Proofread before you send.** Make sure whatever you are communicating is readable by the receiver. If you're writing in a language that's not your native one, consider using a spell-checker like Grammarly to check your mistakes. A well-written piece is infinitely more likely to receive a good response.
2. **Always be polite.** Whether you're sending a job application, asking for online help or emailing a client, you should never be rude. You are asking them to give you their time, and maybe also their expertise or money. Use "Please" and "Thank you", to show that you value them, and understand that **they** are helping **you**.
3. **Get to the point.** Don't write paragraphs upon paragraphs of needless information. State your objective clearly and immediately. Cut out the fluff. Don't make them read a novel before you get to your question or ask.

There's tons more to say about communicating effectively (infact, it's an entire industry!), but these tips should be a good starting point.

Working in a team

Regardless of your field of interest, chances are that most of us will be working in teams - with designers, other developers, project managers etc. Whether you like it or not, these people are your teammates, and you need to work together to reach the common objective.

One of the best tips for being a good teammate is to **listen**. When someone else is communicating, make it a habit to really listen to what they have to say. Even if you disagree, and have your own ideas about something, you will learn so much by **keeping an open mind** and listening properly. This will also help you gain **empathy**, that is, understanding where the other person is coming from and why they say the things they say.

When working with other people, you absolutely need to **leave your ego at the door**. There's no room for it in any team environment. If your ego is more important to you than collaborating to create the best possible solution, you're going to be hurting the team, and ultimately the business. You all need to work together, and that's very difficult when egos come into play.

Taking feedback

When you work in a team, you need to be able to take feedback. Actually, even if you work by yourself, learning **how to take feedback** is a great skill to improve yourself in the field of software development. Being criticized is never fun, and can severely hurt your confidence. But being able to **filter the criticism, locate the important points**, and **use those as opportunities to improve** is an incredible skill.

There's no harm in admitting that you don't know everything and standing up to your mistakes. In fact it shows that you possess an important ability: to realize that you're not perfect, but that you have a **willingness to improve.**

A quick example from a job I applied for: After a written application, 2 video calls, and a code test that I did, I got an email from the company: They chose not to hire me. At first I got a bit upset, but then **I wrote back and asked what I could have done better.**

Not every company you apply for will respond, but this one did. They gave me some very concrete points that had affected their decision: My profile was great, they really liked my person and I had made a nice solution to the coding test. However, they were looking for somebody with more creative drive, to think outside the box and innovate.

This gave me some great areas of opportunity to work on. I would've never realized that had I just accepted the rejection and moved on. A simple comment helped me to begin reflecting on myself and my skills. And that comment helped me land a different job a few months later.

Creating value

Users don't care if your text is 17 or 18 pixels. They don't care if you used React or Vue, and they don't care if your backend is Python or PHP. What matters to them, is that your website/program/solution **helps them to achieve their goal.**

As developers, we sometimes get caught up in technicalities and forget the real reason that we code: To create value for people. At the end of the day, we are **problem solvers**. Code just happens to be a great tool to help with that, and provides a way for us to share our solutions with the world.

Of course, who the users are will vary a lot depending on your field of interest. If you're a data scientist in a big corporation, your users are likely other employees, like marketing or sales people. They need to understand the numbers and graphs you create in order to drive sales. Their main objective is to increase revenue. And your program is simply an aid in that task.

On the other hand, if you're a web developer, your users will likely be "normal", everyday people. They can have a myriad of different goals, with different reasons for being on your website. It's your job to understand what these goals and reasons are, and create the site to best accommodate them.

What matters is that we always focus on **creating value.** We want to make it easier for people to do their jobs or to improve their lives. And our code can only do that if we ship it. We need to get our code out into the real world for it to have meaning. So remember this if you're struggling to complete a side project, or something at your job is taking ages to finish. It's not doing anybody any favors sitting in your code editor.

Example: At my job we got a new client, a high-end Danish clothing manufacturer. They had a webshop, but it was not performing well. Sales weren't great, the shop was confusing to use, and the website didn't align with the brand and the high quality of their products. So they hired us to build a new shop. After a few months, the new shop was basically done. It had a sleek, modern style, with large pictures showing off all of the company's products. It had a blog, and several pages detailing the production workflow of the company. And of course, it had a cart and checkout flow.

When we were getting ready to launch, tiny design details and small changes kept coming up. We'd adjust the site to fit their wishes, and then something else would come up. This went on for **months.** Details like lineheights, font sizes, and which color of gray to use in the footer. I tried as hard as I could to push for a release, to get the new store out into the world. But they insisted that we complete all these tasks beforehand.

When the new webshop finally went live, sales skyrocketed. In the first three days, the sales more than doubled. The conversion rate went from around 0.5%, to over 2%. The new store was **actually solving problems** for people, by making it easy for them to find the right product and complete their purchase. It also showed off the company much better, creating a feeling of trust and stability for the users.

I estimate that several thousands of dollars were lost for every day we held off on releasing the new store. So not only did people not get the clothes they wanted, but the business was also losing large amounts of money. The lesson? **Always keep the end-goal in mind**. Don't get caught up in details. Focus on creating value.

Understand the business

Continuing in the same vein, of course coding can be fun, creative, and inspiring. Sometimes, you want to enjoy nerding out over a new technology or a cool trick you've just learned - and that's great!

But when you code as part of your job, you also need to understand the business side of things. You are not coding "just for the sake of coding". Coding is just the primary tool in your toolbox that you can use to solve real problems. And in a business, there are **lots** of problems to solve. Below I've outlined some of the key areas to focus on when you're working with code as part of your job:

Keep deadlines. When you're coding as a job, "I was trying to decide which JavaScript framework to use" is not a valid excuse. It's an important discussion to have, but not one that should ever interfere with your ability to meet business deadlines. If a client expects a project done at a certain time, there needs to be a good reason for any delays.

Write documentation. Chances are that the business will be around longer than you will. By documenting your code throughout your time, you make it easy for the person after you to understand what's going on. You never want to burn any bridges when leaving a job, so being known as the person who wrote clumsy and undocumented code is not a great reputation to leave behind.

Suck it up. Sometimes you'll have to work on a project you don't like. And sometimes, you might have to work a few hours overtime to get a project out the door or meet an important deadline. You don't need to love it, but being acceptive of this goes a long way towards showing that you understand the bigger picture. This should never be the norm, of course. If you're constantly forced to work overtime or on projects you don't care about, it's time to have a talk with your boss. But every once in a while, you'll simply have to deal with it.

And finally, the most important one:

Businesses exist to make money. And that's not just the job of the sales people or bosses in the company. It's the responsibility of every single employee, you included. You need to keep this goal in mind when making decisions, technical or otherwise. That doesn't mean you should sacrifice your morals, and you definitely shouldn't compromise the user experience. At the end of the day, it's the users that will be paying the business (and your salary!). But you need to remember it if you start to get too caught up in technical details that might not matter all too much.

Staying motivated

Coding is a journey, and you will experience many ups and downs along the way. Staying motivated through the hard times can be difficult. Below are some of my favourite ways to stay motivated:

Personal projects. Regardless if you currently have a coding job or not, personal projects are projects that you create entirely by yourself. This gives you full autonomy to pick the technologies and direction, to make it exactly like **you** want to. You can even create them around something you're passionate about. Someone I know is very religious, so he created a webapp for church members to communicate and manage memberships online. Make sure it is personal to you, and not just a course/academic project.

Solve mundane tasks using code. Maybe you spend a few hours per week copy pasting stuff from one spreadsheet to another, or scouring 5 different websites to see recent bike offers. With code, you have the power to automate this stuff. You could write a script that will copy all of the spreadsheet content, or make a webapp that gets the bike offers from all the different websites and concatenates them into one. This can help you rediscover the power of programming.

Think about how awesome it is that you can make a computer do stuff. This one is a bit more abstract, but just consider it for a moment. You have a piece of electronic equipment sitting in front of you. And you

have the power to make it do whatever you want. You can make it play music. You can make it calculate things. You can make it solve problems for other people. And anything you create can be shared on the internet, with potentially **millions** of people seeing or using it. That's pretty wild when you think about it.

Action creates motivation

Motivation is difficult to plan around because it's always fleeting. It's inherently unpredictable. You can't expect to have the same level of motivation all the time, randomly, for whatever you do. It just doesn't work that way. But there is one way that you can create motivation on demand.

Normally, we tend to wait for motivation before jumping into action. I suggest you flip it around, and use **action as a way to create motivation.** How do you do that? By building habits and being consistent in your work.

Let's take an example: You want to get better at React and hopefully land a React job in the near future. In the first week or two, you're very excited and motivated to learn. Everything is new, and you can constantly feel yourself progressing. But then, once you get past the basics, you start getting stuck. You run into harder problems, with fewer online tutorials to help you. **Your motivation begins to falter.** Now you have a choice:

1. Give up entirely, and start learning something else.
2. Create a system for how you're gonna continue learning React.

This system should consist of tiny, specific actions that will push you in the right direction. It can be as simple as telling yourself you will code at least 5 minutes of React per day. Or it can be a bit more abstract, by saying you'll learn at least one new React concept every day. The important thing is that the actions are **small**. You want to pick an action that you can do right here, right now. By getting started with the particular action, you'll begin to learn the harder parts. The parts that motivation alone could not force you to learn. And by learning them, you'll get hungry for more - fueling your motivation.

Instead of waiting for motivation to strike (which is hard to plan for), you've designed a way to **create motivation on command.** And once you've acquired this skill, you can apply it to anything you want in life. Of course, if you notice yourself **still** being unmotivated after days or weeks doing this, then perhaps the topic is not right for you. But you'll only know for sure if you go the extra mile and give it an honest shot.

Don’t give up

Development is difficult. And when you work with code all day, it’s easy to get frustrated. Sometimes the code is just not working the way you expect. Other times you might question your own abilities. There might even be times where you consider quitting development all together, especially early in your career.

This is when you’re not sure if you can even “make it” as a developer. You might think that you’re simply not cut out for this field. And on top of that, there’s an overwhelming amount of stuff for you to learn.

Everybody (and I mean, **everybody**) has these thoughts from time to time. It’s completely normal. But there are a lot of ways you can deal with them that will hopefully help you to realize that a development career is one of the most fulfilling careers you can pursue.

Firstly, when you’re feeling doubt, you need to figure out what the root cause is. Are you simply frustrated with a small problem you couldn’t solve today? A good night’s sleep will probably help. On the other hand, if you feel anxious about coding over a longer period of time, perhaps you are burned out and need a serious break.

Whatever the issue might be, try to remember why you got into development in the first place. Was it money? The urge to create? The ability to help people all around the world? This reason likely still holds

true, despite the frustration and doubt you might encounter. And it can serve as motivation to power through difficult times.

Coding is a lifelong journey

There could be many reasons why you got into development, but you need to remember that **coding is a lifelong journey**. It's not something you just "learn", and then live off of for the rest of your career. Code is always evolving, and you can never know it all. You need to keep your skills sharp and be ready to learn new technologies as they come along.

Because of the ever changing nature of code, it also takes a large time investment to stay up to date. I see this as a good thing. If coding was easy, everybody would do it. But because it's a skill that needs constant nurturing, many people give up early. This allows you to stand out of the crowd if you manage to practice consistently and stick with it. You just need some trust in yourself and your abilities to learn and pick up new tech as it comes along, and that you **will** get through the frustrating times.

I've learned that there are no shortcuts to learn coding. It takes a large amount of grit and tenacity, especially when going through difficult times. The good thing is that consistency will **always** pay off in the long run. So if you manage to build the right systems, then all you need to do is just **stick with it**. Don't let small stints of doubt impact you too much. With the right habits, time is on your side. Everything will work out if you just keep at it and work on improving yourself a tiny bit each day. **Just don't give up.**

PART 4

Landing the job

Applying for jobs

It's very hard to know when you are ready to apply for jobs, especially if you're new to development. You don't really know how much companies expect you to know, and you don't know what you don't know. You know?

I recommend that you have **at least** 3 projects and a portfolio website about yourself. It doesn't need to be large projects. In fact, it's better if you create smaller projects, but actually **complete** them. Have them do 1 thing, and 1 thing only. 3 small, polished projects are worth way more than several large, unfinished, or half-assed projects.

Since you don't have previous work experience as a developer, having a portfolio to showcase your projects is the way to go. It's the only way to show off your actual skills. This point is obvious for front-end developers, but it applies to back-end, data scientists, and app developers too. If you're not into visuals and design, get a WordPress theme, a premade template, or throw together a simple site. **The point is to show off your projects and provide more detail about yourself.**

Think about it from a company perspective: If a person claims to know HTML, CSS, JavaScript, React, Python and MySQL, but has no previous work experience and no way to show anything they've built - why should the company be interested in them? Anyone can write lots of

keywords on a resume, but by **actually showing** that you know those skills, you'll be miles ahead of the competition.

You'll also want to narrow down the range of jobs you apply to. If you want to work as an app developer, only apply for app jobs. It might take a bit longer to find, but it's better than applying for jobs in a field you're not really interested in. And then you can focus your resume and portfolio on showing off your app development skills.

Be open about your experience

The last thing you want is get a job that you're unable to do. That will reflect badly on you and hurt your confidence. So when you're new at development, you have to be open about your experience.

Let the company know that you've been coding in your own time, that you're new to development, but that you have a great **willingness to learn and improve.** It is absolutely key that you sell yourself on the points that are true, without promising anything you can't deliver. Some companies will reject you based on your lack of experience, but that's okay. You likely wouldn't have fit in those companies anyway.

But if you do manage to find a company willing to take a chance on a junior (and they do exist!), they'll appreciate your honesty and your passion for coding. They'll see that you have the right mindset, and will teach you the coding parts. You just gotta be honest.

Where to apply?

For your first development job, I highly recommend that you apply for larger companies. The type of company isn't too important, but bigger companies tend to be better suited to junior developers for a few reasons:

- Larger companies = more money = less pressure to perform.
- Large companies are more likely to take a chance on you. They've been around for a while, and will continue to be around. This makes them more likely to invest in training junior developers, because they see the long-term benefit.
- Having large company names on your resume will provide you better leverage when applying for other jobs later in your career.
- This might be a bit of personal bias, but in my experience, large companies tend to have a more relaxed culture. People don't need to bust their ass off to make payment every month.

Startups and small businesses are different. They tend to be busy and change quickly. This can be a great environment to learn, because you'll experience the importance of deadlines first-hand. You'll likely get more responsibility, which further increases the speed at which you learn and grow.

However, startups can also be punishing. They might realize that there's no time to properly train a junior, or the whole company might go under.

And having a dead company on your CV doesn't help you much when applying for future jobs.

Getting a student job

If possible in your country, I highly recommend applying for student positions. If you're still studying (engineering or something unrelated), landing a student job is an incredible way to get your foot in the door without the pressure of a full-time job. Companies that hire student developers generally don't expect much from them, and consider the student more of a bonus. This is a great opportunity for you to over deliver and make a name for yourself within the company.

In fact, that's exactly what I did. About halfway through my studies, I applied for a front-end development student position. I showcased my personal projects and portfolio, and made it clear that I was inexperienced but incredibly passionate and hungry to learn. They took a chance on me, and it worked out beautifully for both parties. Around a year later, while still studying, I applied for another student position at a different company. I got the job, and this allowed me to transition into a full-time engineering role at the company when I graduated, avoiding all the competition.

When to apply?

This is the big question. It's easy to see the long list of requirements, even for junior positions, and feel that you're not ready. But if you've built a few projects, and are open about your experience, you should apply anyway. Even if you don't feel ready. **You only need 1 chance**.

The key is to get your foot in the door, and you'll learn so much faster on the job. Think of it this way: what could go wrong by applying? Worst case, you get some experience and a better idea about what companies are looking for. Best case, you get a job that you can grow into along the way. As long as you are open about your experience, you won't end up in a job that you can't perform at. So you have everything to gain from applying as early as possible.

Create a compelling resume

Your resume will be the first point of contact with the company you're applying for. It's absolutely **crucial** that your resume creates a good first impression. The receiver of your resume can either be a recruiter, or someone directly from the company. Either way, the entire goal of your resume is to get the recruiter or company to advance you to the next step in the hiring process.

Adapt to the position

There's a lot of stuff that goes into creating a compelling resume, but most importantly: **you have to adapt your resume to the job you're applying for.** One of the most common mistakes is to send the same resume to 500 companies and hope to hear back. This doesn't work. Instead, you need to personalize your resume to match the job spec. If you're applying for a React job, put your React projects on top of your project list. If you're applying for a data science job, highlight your strong Python capabilities. Make sure you include the most important **keywords** from the job spec.

This does not just apply to technical skills either. When you're researching the company website and reading through the job specification, take note of **which values are important to the company**. If they talk a lot about the importance of learning and self-improvement among employees, mention how you learned to code by doing tutorials

at night. Or if their mission evolves around creating transparent, customer-centric solutions, talk about how you did that while working at a previous job.

There's several reasons for doing this. Firstly, you show that you've done your due diligence. You've read the job specification carefully and researched the company. This proves that you've spent some time on your application, and definitely more than the people who throw the same resume to every company. It also proves that you understand and care about the company mission, which makes you more relatable to hiring managers and indicates that you will fit in well with the other employees and the company culture as a whole.

Highlight skills learned from other fields

A great resume is especially important early in your career, when you don't have a lot of relevant experience. This is where you will need to rely entirely on your resume to move on to the next round. Later on, your previous work experience will do a lot of the heavy lifting, but that's not the case if you're applying for your first development job.

However, even if you don't have a lot of related developer experience, you can still show your previous work places. Use these to highlight skills you learned **that are also relevant for the development job.** If you've worked as a cashier, write how that has helped you interact with customers and learn the "customer is always right" mindset. Or how being a football coach has taught you the importance of teamwork and

leadership. Whatever your previous experience has been, highlight the key learnings from each.

You can also use this to showcase some of your favourite moments from previous jobs, developer related or not. Things like “got promoted to manager after 2 years” or “launched a non-profit website on a super tight deadline” or “learned a new language for a specific project on the job”. If you’ve done awesome things that you’re proud of, don’t be afraid to show it. It feels more human than just listing the responsibilities you had.

Practical quick wins

Keeping all of this in mind, there are also some simple, quick-win tips you can apply to make your resume more compelling:

- Keep it at **1-2 page maximum**. The receiver will simply not have time to read long walls of text. There might be potentially better candidates applying for this job, but by keeping your resume concise, you’ll get an advantage over them by actually having your resume read.
- **Put your experience and skills at the top.** This is the most important part of the resume, so make sure the receiver doesn’t accidentally miss it, or discard your resume before getting to these sections.

- **Show some personality.** You don't need to go crazy with colors and images, but adding small visual design elements will help your resume to stand out.
- **Use a template.** There are tons of good CV template builders on the internet. Using a template will make your resume look more professional at a glance.
- **Proofread.** And have a friend proofread as well. You want your resume to be as free of spelling mistakes and poor wording as possible.

To round off: Your resume might be your **only** chance at landing a particular job. The receiver of your resume will have to make a very quick judgement call of whether or not there's a chance of you providing value to the company. If there is, you'll move to the next round. If not, your application will get discarded.

This may seem very cutthroat - and that's because **it is.** But it's how the world works. When reviewing your resume, try to put yourself in the shoes of the company or recruiter receiving it. Would you pass yourself on to the next round? Even if you had as little as 7.4 seconds to decide? If not, your resume could still use some work.

Catching attention

Most development jobs get tons of applications. As discussed, your resume needs to be great to launch you forward into the next round. But there are other things you can do to stand out from the crowd.

Portfolio

If you're applying for a junior job and you have very little experience, a portfolio is one of the best ways to show off your capabilities. Since you don't have a lot of work experience to show, your portfolio serves as proof that you know what you're doing. Anybody can write a bunch of skills on their resume. But with a portfolio, by showing off projects you've done, you can actually demonstrate your abilities.

As mentioned earlier, you need a few projects on your portfolio. These can be modified tutorials, projects you've built from scratch, or even your actual portfolio site (it counts!). If you have a lot of projects, highlight your favourite ones. There should rarely be more than 5-6, and make sure to put your best ones on top.

You can also use your portfolio to show off some personality. Maybe add a picture, show off your favourite colors, or make it interactive somehow. There's a lot of opportunity here, even if you're not a front-end developer. Be creative with it - this is your website.

You'll probably also want the usual stuff like an "about" section, a way to contact you, and maybe a blog. I also like to showcase some of the technologies I'm familiar with, like programming languages, frameworks, or something like Git. Since this is your own site, and not a resume, you can add a lot more information about yourself. If your resume doesn't sell you on its own, the goal should be to get the reviewer to visit your website. And then your website **has to** do the selling.

Some "Do's and Don'ts" to help you with this:

Do:

- **Show some personality.** Remember, you're trying to stand out from the crowd.
- **Write clear and informative copy.** A line or two to grab the attention, and short paragraphs here and there to elaborate.
- **Showcase your experience and skills.** Highlight your projects, and anything else you've worked on.
- **Be responsive.** Your website **has to** be mobile-friendly - no exceptions.

All of these will help you create a good first impression that the recruiter or company is more likely to remember. In the meantime, avoid the things below:

Don’t:

- **Hide your experience**. You don’t know how long the reviewer will be on your site, so don’t force them to go digging for the most important thing they want to see.
- **Write large amounts of text.** Nobody is going to read several pages about your life story.
- **Use skillbars.** These are popular amongst developer portfolios, but they’re completely arbitrary. What does “80% HTML” mean? Does it mean you know 80% of the tags? Or that you can write 80% of the HTML needed for a website? The point is that skill bars are viewed subjectively, and that makes it better to avoid them altogether.

Online presence

Your online presence is everything that shows up when someone types your name into Google. It can be your website, your Twitter account, LinkedIn, and any other social accounts you use.

It’s important that these media provide a clear picture of **who you are,** and that they show your most professional side. Specifically, you should write your profiles as if you are already in the position you want to be in. If you’re applying for a position as front-end developer, write that in your LinkedIn title. Have a good looking profile photo on every site. Spend some time writing out the sections and adding your experience.

Another great way to stand out, is to have a blog. A blog will do many great things for your career (more on this in a later chapter), but it's especially important to further build your online presence. It shows that you care enough about development to actually **write about it in your free time.** That's a great message to send to companies and potential employers.

Make them remember you

Building an online presence takes time. It's a great investment, but there's also a few tips you can quickly apply that will make recruiters and companies more likely to remember you:

Make yourself look good. Use professional photos (or at least very good looking ones) only. Proofread every email and application. Make sure it's concise, to the point, and clear of any grammatical errors. This is your chance to show that you can communicate in writing, an important skill for any developer.

Reach out before applying. If there's an email in the job spec, or you follow someone from the company on LinkedIn or similar, try to reach out. Just say hello and ask a simple question. This will show that you're proactive, and will improve your odds of them remembering you when they read through your application.

Example: I added the CTO of a medium-sized company (50 employees) on LinkedIn, after seeing their open position for a front-end developer, with him listed as the person to contact. I wrote a small message, just saying hi, and asking if they were still actively looking for applicants with all the uncertainties of COVID-19 going around. He said they were, and that I should definitely give it a shot. I applied right away, and the next day he offered me an interview.

This also leads nicely into the next tip:

Be quick to move. When you notice a new job opening, apply as early as possible. If the company gets back to you, reply quickly. I can't stress this enough.

Example: I got my first job after having met the company at a career fair at my university. I wasn't sure if I should apply, since I was pretty new to development at the time, and didn't fit all of the requirements. After having a chat, they told me to just give it a shot and apply anyway. I went home and did it on the same day. This company took a bit longer to respond, with more emails and interviews back-and-forth. Every time I would respond right away. After I got the job, I learned that they were very impressed with the speed at which I applied. And that it was this speed that made me stand out over other candidates. They had even talked about it in the office, so when I started, people literally came up to me and said "oh yeah, you're the guy that moves super quickly, right?" That's a great identity to have.

Passing the interviews

Hopefully your compelling CV, professional online presence, and awesome portfolio has helped you stand out from the crowd and land an interview. Even if you haven't quite gotten an interview yet, you **will** have to pass them at some point to get any development job. So knowing how to interview and sell yourself will prove immensely valuable.

Each company has their own methods for interviewing, including the amount of interviews, who's part of them, and what the subjects are. In my experience, interviews tend to come in 3 forms:

The informal, personal conversation

This is usually the first interview. It will be you, and likely 1 or 2 people from the company. This interview is mostly to see if there's a personal and cultural fit between you and them. You'll tell your story, they will tell theirs, and you'll chat about your interests and past experience. It tends to be very informal. It's designed to be a simple conversation, so no need to freak out before this phase. This is also where both parties get to see if their long-term goals align.

Advice for passing:

- **Just relax!** It's a normal conversation, and can even be pretty chill if the company you're applying for has the right culture.
- **Think of some bullet points in your head,** so you're ready when they ask you to "tell us about yourself".
- **Be honest about your own abilities.** Definitely sell yourself, but don't promise things you can't deliver.
- **Prepare some questions for the end of the interview**, and even feel free to ask stuff as you go. The conversation will be a lot better that way, and you'll show that you really care about the job and want to fit in.

The technical / whiteboard interview

This one is very common, and it usually comes after the first conversation. You'll likely get asked to solve some kind of programming problem on the spot. The exact topics will vary and depend on your specific development field. But the idea is the same: they want to see how you approach a problem. They want to understand your thought process and see if you are capable of solving technical challenges. They don't expect you to perform flawlessly, or even be able to solve every problem perfectly. They're just interested in how you take a problem, break it down, and use logic to arrive at a meaningful conclusion.

Advice for passing:

- **Do your homework.** If you know which technology they'll ask around, spend some time preparing questions related to that technology.
- **Approach the problem logically.** Think through the problem, break it down, and make a plan before you start. It's also fine to ask questions if something is unclear to you.
- **They're interested in your thought process.** Not perfect code. Try to think aloud. Start with a poor solution, and if you notice a better one, explain why it's better and switch to it.

Dealing with technical interviews is a whole topic by itself. If you're interested in learning more about it, check out Decoding the Technical Interview Process

The take-home coding test

After your first conversation, the company might ask you to complete a coding test. Again the format varies. They might give you a set time-frame, like 48 hours, to complete the task. Often, the scope of the test will be larger than what you can reasonably expect to complete in the time-frame. This is intended for you to prioritize individual aspects of the test. The purpose of the take-home assignment is to, again, verify your technical skills. And the assignment will get followed by a code reviewing interview. Here you'll have the chance to explain the decisions

you've made, the technologies you've chosen, and the areas of the assignment you decided to focus on. You'll also get the chance to explain how you could improve the solution if you had more time.

Advice for passing:

- **Read the assignment carefully.** The last thing you want is to fail the interview because you missed an important detail in the assignment description.
- **Make a plan.** Which features are must-have, which would be nice-to-have, and which do you have to omit. The key is to have a finished project within the time-frame. By planning ahead, you already have the basis behind your decisions that you can discuss in the interview.
- **Spend time writing clear code.** How your solution looks and performs is important, but so is the code quality. Look for "messy" code and try to refactor it before you submit the assignment.
- **Throw in some fun details.** I've done a few take-home assignments, and the companies have really appreciated a touch of personality and some light-hearted fun. This is especially important for front-end developers - throw in some confetti, fireworks and animations. It really makes a difference.

Interviews are harder than the job

Regardless of the exact form of the technical interview, you need to remember that **interviews tend to be harder than the actual job.** They put you on the spot, and force you to think on your feet, which will rarely be the case if you actually land the job. In reality, you'll always have proper time to think, Google, or ask your co-workers. So even if you feel like you did poorly in an interview, it doesn't reflect at all on your ability to perform on the job.

You should definitely prepare well for the technical interviews, and always try to do your best. Remember that they're mostly testing your ability to approach a problem logically, and your ability to think your way to a solution. It doesn't need to be perfect, and it's okay to make some mistakes. Try to stay cool, and don't stress about the exact details of your solution, or even the correct syntax if you're writing code on a whiteboard. Be confident, ask questions, don't pretend to know everything, and show your thought process.

And finally, remember **that you are also interviewing the company.** They need to be a good fit for you as much as you need to be a good fit for them. Ask questions and be open about your expectations. If they're not a match for you, politely thank them for their time and move on.

How to negotiate

If you've managed to get past the dreaded resume phase, the technical interviews, and perhaps a coding test, all the way to a job offer - Congratulations! Give yourself a pat on the back. You've earned it.

Before you accept the offer, though, there is one key step left that some might overlook: **Negotiating**. Negotiating is the part where you and the employer decide on your salary. This can be an uncomfortable situation for many, because talking about money is hard. And putting a dollar value on your own worth is even harder. Nonetheless, negotiating is absolutely **crucial.** You have so much to gain, with very little to lose.

Let's look at the facts here:

- The company has spent a significant amount of time on you through the entire application process
- They're a business, so they will always try to get the best deal
- They've negotiated hundreds or maybe even thousands of times before
- They want to hire you

It stands to reason that they want to hire you for as low an amount as possible without you feeling cheated. That's Business 101.

But with all of the time they spent and their interest in you, they're not going to get angry or dismiss you for asking too highly. Not at this point. They might refuse, or counter with a lower offer, but they won't suddenly back out.

This puts **you** in the position of trying to get the best deal possible. Negotiating is an artform in itself, but there's some key points that will help you out:

1. **Always try to make them say a number first.** You can try to ask what the range is for this position. Most companies won't tell you. But you'll have the advantage if they do.
2. **Never mention your previous salary.** They're legally not allowed to ask. And your previous salary has nothing to do with your capabilities in this job or how much your skills are worth.
3. **Provide a range.** If you have to go first, always give a range with the lowest being higher than what you would be satisfied with. Let's say you'll be happy with a job paying 50k per year (adjust to whatever is appropriate in your country/situation). When you find this number, increase it a little bit. That's the lowest point of your range. Your highest point should be a tad higher, but not too much. If the numbers are too far apart, the range makes no sense. In this example, you could say a range of 56-62k.

4. **Use odd numbers.** A number like 62k works better than 60k, because it seems like you've put a lot of thought into it. This is kind of a trick, but it works in my experience.

Hopefully with all of this, you'll go back and forth a bit until you land on a number. And since you anchored the conversation higher than what you would be satisfied with, you're likely to earn more than if you had just suggested 50k outright. You'll be surprised how often this technique works.

If you're unable to meet each other, consider accepting a lower salary in exchange for other perks that have value to you. This can be stuff like an extra vacation week, the ability to work from home 1 day per week, or leaving an hour or two early on fridays. All of this adds up, and might even be worth more to you (personally) than a slightly higher salary.

Start a blog

For the final chapter, I will leave you with a piece of advice that will help you throughout your entire coding journey:

Start a blog!

The beauty of blogging is that it's so easy to get started, and once you've written a post, it'll be available on the internet for the entire world to see. What to blog about is largely up to you, but I have some recommendations later in this chapter.

You might be wondering how a blog can help you grow as a developer, and there's many key reasons for that:

It'll reinforce your learning. By blogging, you force yourself to learn topics deeply. You can't explain something to anybody if you don't know it well enough, so you'll have to learn concepts properly to blog about them. Blogging also helps you **retain** the knowledge that you gather while learning. And should you forget something, you can always refer back to your own blogpost!

You'll improve your written communication. Regardless of your development field, written communication is a key skill to master. By blogging you'll improve your vocabulary, your grammar, your eye for

detail, and how you communicate topics that are important to you. All of this will help you in your development job.

It'll make you stand out when applying for jobs. Having a blog shows that you care more about development than most other candidates. You go the extra mile to document your learnings, and you spend time on coding related activities outside of work. Both of these are important to employers.

You'll be helping others. Anyone who has encountered the same problem as you can run into your blog post. It might help them solve an issue they otherwise wouldn't have been able to. I'm a big believer in "paying it forwards" in development, so just like how we use blog posts to learn and improve, we should aim to create them too, so others can do the same.

Choosing a topic

Since this is **your** blog, and you're (primarily) doing it for your own sake, it can be about anything you want. The easiest way to get started is to write about your experiences. Document your journey, how you got started in development, how you landed your first job etc. Or if you're very early in your career, talk about some of the struggles you are facing applying for jobs, or technologies you have trouble learning. Use your blog as a way to reflect. It'll also work as a nice motivation boost later, when you can look back at your early days and see how much you've grown.

You can also write about more technical subjects. If you're a front-end developer, write how you created a certain project in JavaScript. One of my first blog posts was a tutorial on how to build a Gif searcher with Vue, using the Giphy API. This forced me to really **understand** the project and what every line of code was doing. And it helped me later, when I was unsure of how to fetch data from an API in Vue - I had my own post to refer back to for a working example. This will, again, also make you stand out when applying for jobs. Who do you think a company would rather hire: A candidate who claims to be good at Vue, but with nothing to show for it, or someone who literally writes tutorials **teaching others** how to use Vue? I know which one I would pick.

I wrote this blog post years ago, and I still get people messaging me from time to time telling me how it has helped them in their own journey. It's a pretty incredible feeling.

You can even write about something completely unrelated to development. Non-technical blog posts will still help you improve your communication skills, and you can still use the blog to reflect back on later.

Getting started

The easiest way to get started is with a blogging platform like Medium or Substack. These already have great editors, and solve a lot of the pain-points you might run into when getting started. The downside is that the posts will be hosted on their servers, meaning you won't be able to store them on your own website/domain.
Another alternative is WordPress. You set it up, pick a theme, and you've got a working blog with comments, tags, images, and everything else you need. WordPress is a great way to get started, and allows you to host the blog yourself. You can use this to make the blog an integral part of your personal website. Getting it setup can be a bit tricky, but you only need to do it once, and you'll completely own your content forever.

If you're up for more of a technical challenge, you can write your own theme using something like Gatsby. This will give you full flexibility of the look and feel of your blog, using React, and the ability to pick whichever Content Management System (CMS) you like. Some suggestions could be Contentful, Netlify CMS, or just Markdown files.

For my blog, I use Sapper (which is a framework on top of Svelte), since I host the blog on my own personal website. I write each article in Markdown, and Sapper then turns it into HTML and generates a unique page for each post. The articles also contain meta information like the date, some tags, and the expected reading time.

I like this approach because I have 100% control. Markdown gives me all the options I need for code snippets, lists, links etc., and then I can easily style the posts to fit nicely with the rest of my website and brand. However, I don't necessarily recommend this option if you're just starting out. You want to make it as easy as possible for yourself to write, so spending a lot of time setting up your own blog is generally not the best idea. You can always revisit this option later when you have more experience and content to show off.

Reducing friction

Whichever method you choose, you want to **reduce friction as much as possible** regarding new posts. Writing and publishing a blog post should be as quick and painless as possible, so keep that in mind when choosing your platform. In the beginning, the biggest obstacle you'll run into when having a blog is how to write posts consistently. So by reducing friction, you're making it much easier to build the writing habit.

I recommend starting small, maybe writing 1 post per month. It's tempting to only post when you **feel like it,** or when you have an interesting topic you want to cover, especially in the beginning. In my opinion, this is the wrong approach. Inspiration and motivation are both very fleeting, and you never know when they'll kick in. But by building the habit of publishing 1 post per month, you'll actually be able to **create motivation on command**. And finding topics will help to exercise the creative part of your brain, and realize which areas you care deeply about.

Finally, remember that this is your platform. You can write about whatever you want, at whatever pace you want. As long as you are actually writing, the blog will be a good investment. And that investment will pay off throughout your entire coding career.

— Mads Brodt