From first steps to levelling-up in Software Development

If you're reading this there's a high chance that you are either looking to pursue a new career in tech, or are on a quest for further knowledge to level-up your development game. In this post I round things up to help on your journey - and hopefully provoke a few thoughts along the way.

So, let us begin

In the modern world information is more accessible than ever and gone are the long days of sifting through book after book looking for an answer, now we just have too many resources to wade through and can often find ourselves overwhelmed - crazy, right? Maybe, maybe not. The internet of today has given each of us a voice and with it an opportunity to freely express, share, and collaborate as a collective brain in many cases (with some piano playing cats and 10-hour crab dances thrown in too). It's more a thing of beauty than it is crazy.
When we are spoilt for choice the next question then becomes "whose information can I trust the most?" so, without further ado, here's a round-up of tips stemming directly from industry experience as a software developer.

It starts with the right mentality

Ask questions, shed the fear

From the very beginning always be honest with yourself when it comes to your current technical level of understanding, within the software industry in particular there are very few that know everything in part due to the ever evolving nature of technology.
Regardless of how big or small in scope we should always be comfortable asking questions. We as a society have a subconscious need to please those around us whilst giving off a hardened impression of extreme confidence but in reality it takes more confidence to admit to a weakness, and we in the industry see those people as driven, motivated, eager, passionate, and open-minded.

Shed the fear, reflect, ask, and keep on pushing forward.

Forget complacency, be inquisitive

Each situation is a new opportunity to learn, though it's possible to reach a point in which you know enough to almost autonomously speed through project after project by repeating your tried

and tested process, what do you actually gain? There's always something we can do differently, or better. That intrigue may often lead you down a rabbit hole but that's a good thing; nothing ventured, nothing gained.

Never settle, keep exploring, and don't let yourself become complacent.

Keep yourself updated

The idea of keeping yourself up-to-date in the world of tech can be a touch daunting with so much happening but it's easier than you think.
Social media in particular can be the bane of existence for some but it can also be leveraged for your benefit given that the vast majority of companies, and collectives, report news, general updates, and stimulate discussions through platforms such as Twitter.
Keeping yourself updated doesn't always mean spending hours upon hours daily digesting every single shred of information, most of the time half of the battle is awareness. How can you implement, or use, something new if you don't first know it exists? The bridge of further investigation can be crossed when you come to it.

Here are a few potential follow candidates to get you started:-

- **@FullstackDevJS**
- **@howToCodeWell**
- **@_100daysofCode**
- **@reactjs**
- **@shanselman**
- **@cdibona**
- **@SaraJChipps**
- **@linclark**

Keep yourself aware, explore as and when needed.

Let's talk productivity

Take breaks, seriously

Now hear me out on this one as it may seem somewhat counter intuitive, but what if I told you that taking regular breaks will absolutely make you a better developer? There's actually extensive research into this if you're giving me the side eye but, speaking from experience, the

longer you remain fixed on a given task the more risk you run of hitting a temporary wall due to mental fatigue. You'd be surprised how often developers step away from a problem only to have an unexpected epiphany over a coffee, whilst taking a walk, or using the... facilities (seriously). Some suggest taking a break every hour, but listen to yourself and do what works best for you - if you feel a building frustration, take a step back even if only for a moment.
Take a look at the Pomodoro Technique to really give yourself the chance to reset.

Stretch your legs to stretch your mind, you aren't being lazy you're giving yourself the mental space for free flowing creativity and a clearer view of the bigger picture.

Take command with command line

We have a lot of reliance on fancy graphical user interfaces (GUIs) and I mean, I guess that's fine, but do you really know what's happening behind the scenes? Not only that but clicking through menus may only take a few extra seconds in some circumstances, but that quickly mounts to hours over time - as 1%er as that may sound, who doesn't want to free up a little time? Keep things simple though, you can quickly navigate, create, and edit files and even handle big data (there are even plenty of pretty cool quality of life plugins you could take a peek at).
Aside from the marginal, let's think about workflow: It's more than likely that your IDE has a built-in console meaning you can increase productivity by staying within the same window, plus given how deeply ingrained git and GitHub are in development it never hurts to be extra comfortable with the command line.

Get comfortable with the command line to streamline your workflow.

Customise your Integrated Development Environment (IDE)

This is overlooked far too often.
Regardless of which IDE you've personally chosen to use for your development purposes, I guarantee there's a lot we can do in order to make it work better for you. This could be a change in theme, or font, to make things a little easier on the eyes, reorganising panels for easier access, clearer and more distinctive code handling or additional support for more meaningful tooltips. Though this is incredibly subjective the only real *don't I'd* like to throw out there is this: When you're starting out, or in the early stages, syntax is even more important. Though auto-corrections and auto-completes can save a little time, that time is better spent getting to grips with code structure and your understanding of syntax. It's somewhat similar to being able to read but having terrible handwriting when you hardly use a pen or pencil, practise makes perfect.

Delve into those settings menus, grab some plugins, and make your IDE work for you.

Invest some time into git

Be it for personal development reasons, or working as part of a development team, at some point git and GitHub will feature heavily in your day-to-day life as a developer.
Not to be confused with GitHub, git is a piece of software for tracking changes which also enables us to work collaboratively on the same project via branches without worrying about data integrity. Teamed with GitHub it's great to also push your projects to personal repositories to share, open up to the world, or even just to give yourself more flexibility to work from anywhere by cloning down your project. I won't drone on about this, give it a look and I assure you it will speak for itself - the community is also pretty fantastic, and there's nothing wrong with pushing for a more open-source world.

Don't let the name fool you, git will quickly become your best friend in dev-land.

Coding principles to live by

DRY (Don't Repeat Yourself)

The principle itself comes thanks to Andy Hunt and Dave Thomas in their book "The Pragmatic Programmer" (A strongly recommended read for those wishing to expand their book collection). In their words: "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system".

In simpler terms we should write code that truly works for us; it should be thoughtful, distinctive, and reusable.

Break your code down into small reusable chunks, use smarter logic for repetitive tasks, and ensure that your code has a clear purpose.

I recently wrote a post on this topic, a little more information can be found here.

YAGNI (You Aren't Going To Need It)

This principle is all about being mindful, you should never write code because there's a small chance you may need it in the future. You wouldn't fix something that isn't broken, and you also shouldn't solve a problem that doesn't exist.
Occasionally some may focus too heavily on DRY principles of programming, and as a result end up with a mountain of functions, for example, that are only ever used once or could be pooled together to reduce the amount of code. Abstraction, or more specifically too much abstraction, is a no-no as you'll quickly find yourself with horrendously unreadable code.

If you find code written over, and over, and over then implement a layer of abstraction. Otherwise, best leave it alone.

KISS (Keep It Simple, Stupid)

Arguably on the harsher side of naming conventions, but the message is pretty clear - feel like you're over complicating things? Keep it simple, stupid.
We can often be the thorn in our own sides in an attempt to be too 'clever' to the extent we find ourselves caught up in the moment. If we can introduce a simple one line solution it's without question the way forward, code doesn't have to look fancy to be fancy.
This also applies to being clear, and distinctive, when it comes to naming conventions - just keep your code relevant and to the point.

```
function addNumbers(num1, num2){
   return num1 + num2;
}

function subtractNumbers(num1, num2){
   return num1 - num2;
}
```

Simple solution, to a simple problem, but that's kind of the point.

Open and Closed

This may be more useful further down the road, and as such I won't dive too deep.
But this principle of programming is to encourage direct extensions of the work you do opposed to modification - I.E **open** to extension and **closed** to modification.
Let's say you decide to release a library, if said library is completely open to modification and integration then thing's become a little messy following a major update - as a result your own code, or the code of your users, will more than likely break, and you'll have a fair few fires to put out.
Instead ensure that your code release prevents direct modification and instead encourages extension. By keeping code behaviours separate from modified behaviours your code will be much more stable and also easier to maintain.

Composition over Inheritance

If you live by the 4 pillars of object-orientated programming then this will more then likely save you a lot of headache.
Composition over inheritance is yet another fundamental principle which states that objects with complex behaviours should only contain instances of objects with individual behaviours. They should not add new behaviours or inherit a class.

If we lean too heavily on inheritance we usually experience two primary issues:

- The hierarchy of inheritance can rapidly become a complete mess
- You find you have less flexibility for special-case behaviours

The solution is composition programming which looks cleaner, is more easily maintained, and grants greater flexibility. Each behaviour has its own class and you can also create complex behaviours by combining individual behaviours.

Documentation

Now this isn't to say that every single piece of code you write should be exhaustively documented in some separate format - that would be unrealistic and entirely counter-productive. Code comments go a long way, a long long way in fact. Not only for yourself, but for those who eventually collaborate with you. Aside from the obvious here, let's consider this example: think about your IDE, have you noticed that you can expand and collapse blocks of code to increase readability? Now imagine each block has a descriptive comment above it so that you can quickly navigate and understand the logic without going line by line. Nice, right?

Document to make clear, immediate, connections with the code you've previously written, or are writing... and thank me later.

In Summary

Don't let the volume of resources available deter you from pressing forward, and ultimately don't dwell too much on any one given principle. Sure these are technically some of the first things that come to mind when I think 'best practise' but to be honest it all comes with time and practise, just remain open-minded and generally mindful - always strive to make improvements and be honest with yourself when looking back to review your own code. We can always do something differently, we can always make improvements, we can always refactor to be more optimal, the difficulty is having the openness to see it.

So here's to you fellow developer: ***you got this.***

Here at Code Nation, we cover all of the principles discussed in this article through our Master Coding bootcamp where you can gain the knowledge, skills, and behaviours needed to become a Junior developer in as little as 12 weeks. Speak with us today to find out more.