More Than Code

UNTITLED

1 Introduction

Welcome to the book. I hope you find it useful. Over the next few pages, I'll give you a high-level overview of what the book is, where it comes from and why I think I'm qualified to be writing it. If you don't need convincing, then feel free to skip ahead to the next chapter.

1.1 What is this?

If you work as a computer programmer, then you write code. That will, of course, take up a lot (probably most) of your working day. Not all of that time is spent actually typing code into your programmers editor. A lot of it will be in meetings where you're tying down the specification for the software that you're about to write. Or you're sitting, thinking through how you're going to implement a certain part of the new system. Or you're trying various scenarios trying to understand why your code isn't working the way you expect it too.

All of that counts as programming. But there are other activities that aren't strictly speaking programming but that programmers do on a daily basis. Or things that, if a programmer was to spend time thinking about them, then he or she would become a better programmer (or a more successful programmer or a more fulfilled programmer - however you want to measure those things).

Those other, non-programming, activities is what this book is about. We'll look at the working practices of a modern programmer and describe some of the most useful extra things that a programmer will be doing (or should be doing).

The book is based on a half-day workshop that I ran at a Perl conference in Glasgow in 2018. The workshop was rather loosely structured. I had list of topics and we discussed them as a group. I put forward my thoughts and the dozen or so attendees chipped in with their opinions. At the end of the session everyone seemed to

think it had been a useful use of their time and a few attendees approached me over the rest of the conference and said they thought the workshop would make an interesting book. I've been pondering that for eighteen months and this is the result. It's a much updated and expanded version of the workshop. I hope you find it as useful as the original attendees did.

1.2 Fluffiness vs time

I've structured this book so it gets fluffier over time. We'll start with some very hard technical skills that I think all programmers will find useful. And as time passes we'll start to look at slightly softer skills.

This was a deliberate choice for the original workshop. When you're talking to people in a small room on a Tuesday afternoon, any trainer will tell you that the audience's attention will wander more the later it gets. I therefore wanted to get the harder skills covered first when the audience was at its most alert. As it happens, the softer topics covered later in the day seemed to encourage just as much discussion as the earlier ones - so perhaps I needn't have worried.

I have, however, kept that structure in the book. I've realised that we don't just move from hard topics to softer ones, but we also move from nitty-gritty everyday discussions like "how do I get the most out of my programming tools?" to wider issues like "how does my employer make money?" and "how do I promote my person brand?" that you probably don't need to be thinking about every day.

The broad strokes that we will be covering are as follows:

- **Know your tools** - getting the most out of the tools that you use every day.
- **Adjacent technologies** - what other technologies does your programming work touch on?
- **Software engineering and architecture** - how do you efficiently design your code; and how does it interact with the rest of the system.

- **The business** - how much do you know about the company you work for and the market it exists in?
- **Personal development** - how do you structure your career to make it as fulfilling as possible for you?

1.3 Who are you?

You might be wondering who I am and why I'm qualified to write this book. So I thought it was worth taking a few paragraphs to tell you a little about my history in the industry.

My name is Dave Cross and I'm very old. I've been working in the IT industry since 1988. For most of that time (since 1995) I've largely worked as a freelancer. I'm old enough that when I was at school, there was no computing included on the syllabus - even though I stayed on in the sixth form and did two maths A levels. There was an option to study for a computer studies A level, but that involved travelling once a week to a different school fifteen miles away and it didn't seem that important at the time.

After leaving school, I did the first year of a physics degree at a university in London. Surprisingly, there was no computing included on that course either - although I'm told that had I stayed on for the second year of the course, I would have learned some FORTRAN. But I dropped out after the first year and spent some time considering my options.

I stayed in London, but on visits to my parents I found that they had bought a Sinclair Spectrum for my younger brother. He wasn't particularly interested, but I was drawn to the concept of programming. I bought a copy of Donald Alcock's *Illustrating BASIC* and slowly taught myself the basics of programming. I soon realised that I found programming interesting and started to wonder if I could build a career doing it. Unlike many of my peers (who were coding from an early age) I was twenty-one before I wrote my first code.

Back in London I looked for a degree that would teach me more about programming. I was offered interviews at two or three polytechnics and the first I attended was at South Bank Polytechnic. At that interview, the course director told me that the course was aimed at people who would spend their careers writing COBOL in the data processing departments of large companies. It was 1984 and neither of us had an inkling of how computing would change in the very near future.

I was accepted onto the course and studied there for four years (including an industrial placement year). The syllabus had us learning Pascal and COBOL and practicing what we learned on DEC VAX computers running VMS (except one that had some version of Unix installed). AI was represented by Prolog and expert systems and databases were dBase at one end of the spectrum or CODASYL monstrosities at the other. I recall writing one extended essay on the Japanese “5th Generation Computing Project”. I wonder what became of that.

There was useful stuff too. We learned C in our final year and I became well-acquainted with Unix. SQL was introduced as a new tool that was intended to allow end users to query databases. And during my industrial training year I worked at IBM and used a new markup languages called SGML. Embarrassingly, my final-year thesis dismissed the newly-emerging window- and mouse-driven computer interfaces in favour of a text-based menu system. To be fair to me, if you’d used Windows versions 1 or 2 you would probably have dismissed them as well.

I graduated with a first class honours degree and got a job with a company who did consultancy on software design. This was 1988, so the design method they advocated was their own adaptation of SSADM (Structured Systems Analysis and Design Method - a UK government designed method which was the very antithesis of today’s agile approaches). The company had a CASE (Computer-Aided Software Engineering) tool and I joined the team that was writing the logical data modelling part of the product.

The new version of the product that I was working on was going to run on Windows. This was a brave decision for the company to make in 1988 as Windows was far from ubiquitous at that point. It was also challenging for the developers as no-one really knew how to write Windows code at that time. But I spent four years there and got pretty good at writing for Windows. My C and SQL both improved dramatically over that time too.

I left there in 1992 and got a job working for a communications company. Back then “communications” meant things like fax and telex. But I got to learn a lot about networking and I moved from C to C++. I remember having my first real email address at that company - it was from Compuserve. I talked to some senior people at the company about including email in their products but they didn’t seem that keen. I could see that they weren’t very forward-looking, so I left after only six months.

My next job was with Walt Disney. Or, more accurately, with their home video subsidiary - Buena Vista Home Video. At the time, they were just starting to release a lot of their films on VHS. All of the video production and distribution was handled by third parties in each European country and they wanted a system that could aggregate all of this sales data and report back to people in Burbank. So that’s what my team was building. I got the job on the basis of my C and SQL knowledge and, in return, I learned a lot more about Unix and my SQL knowledge became more specialised towards Sybase.

After a couple of years with Disney, I started to get calls from recruitment agents looking for contractors to work in the City of London. It seems that by using C and Sybase on Unix, Disney had chosen exactly the set of technologies that made their developers very popular in investment banks. Over a period of about six months, most of my team (including me) became contractors in the City.

Over the next five or six years, I worked for a number of banks in the City. I started off using my C/Unix/Sybase skills, but I soon picked up a few new technologies. Perl was very useful to me for a long time. It was largely used to build dynamic web sites and that led me into the

web. Remember I mentioned using SGML at IBM? Well, SGML was the forerunner of XML which was the forerunner of HTML. So my SGML knowledge (which I had assumed I would never need again) gave me a bit of a head start on the world wide web.

For a while a lot of my work involved using Perl to put web front-ends onto various types of database. To be honest, a lot of City jobs aren't particularly exciting, but my web work opened up a few more possibilities. I did stints at a few internet start-ups along with more established media companies. All the time I was able to use existing skills to get a contract and then pick up other, newer, more useful skills while I was there.

I smile now when I think of the course director interviewing me for a place on his degree course. I really don't think I would have been happy in one of the data processing centres he was talking about. But I've successfully managed to avoid that and have a successful and interesting career.

I estimate that I've only ever used 25% of the topics I learned on my degree. A lot of the rest of it was out of date before I even finished the degree. Keeping up to date with industry will always be a problem for the education system - particularly in an industry that changes as quickly as ours does. I'm sure that it's still possible to have a long and lucrative career if you're just using skills that were current when you started in the industry. But I'm also sure that you'll enjoy yourself more if you take the time to keep up to date and move with the industry.

And that, I hope, is one of your reasons for picking up this book.

2 Know your tools

There are three tools that every programmer will use pretty much every day. Knowing how to get the most out of them will save you time and make you a better programmer. The tools are an editor, an operating system and a source code control system. Let's look at each of these in more detail.

2.1 Editor

If you're creating code then you're using an editor. Therefore, how well you can use your editor will have a massive effect on how productive you are as a programmer. If you don't believe me, take the time to watch a couple of your more experienced colleagues at work. You'll be amazed at how quickly their fingers fly across the keyboard and new code appears. And it's not just the creation of new code, you'll also be opening new files and moving around a number of open files to find the definitions of variables and functions that you're using. An experienced user with a good programmers editor moves around the code at an incredible rate.

Traditionally, on Unix and Linux in particular, the choice of programmers editor has come down to Emacs versus vi (which is, actually, usually vim). But these days, there are many more options to consider. I'm currently using Atom and I see a lot of programmers using Microsoft's Visual Studio Code. People using Windows often seem to use Notepad++ and Sublime Text is popular on MacOS.

2.1.1 Programmer's Editor

A programmer's editor is a type of text editor that is specifically designed for software development. Unlike a general-purpose text editor, a programmer's editor includes a number of features that make it easier to write and edit code. Here are some of the key

advantages of using a programmer's editor for software development:

Syntax highlighting: One of the most useful features of a programmer's editor is syntax highlighting, which is a visual formatting of the text that makes it easier to read and understand. Syntax highlighting automatically assigns different colors to different elements of the code, such as keywords, comments, and strings. This makes it easy to identify the different parts of the code at a glance, and can help you spot errors and inconsistencies more quickly.

Auto-completion: Another useful feature of a programmer's editor is auto-completion, which is a type of code completion that automatically suggests possible completions for the code that you are writing. This can save you time and effort, and can help you avoid mistakes. For example, if you start typing the name of a variable, the editor may automatically suggest the full variable name based on the other variables in your code.

Code navigation: A programmer's editor can also help you navigate through your code more easily. Many editors include features like code folding, which allows you to collapse and expand sections of your code, and code outlining, which provides an overview of the structure of your code. This can make it easier to find the specific part of your code that you are looking for, and can help you understand the overall architecture of your application.

Integrated debugging: Many programmer's editors also include tools for debugging your code. For example, some editors allow you to set breakpoints, which are points in your code where the execution will pause so you can inspect the state of the program. This can help you identify and fix errors in your code more quickly.

Customizability: Another advantage of using a programmer's editor is that they are often highly customizable. Most editors allow you to adjust the color scheme, define your own keyboard shortcuts, and install plugins to add additional functionality. This means that you

can tailor the editor to your specific needs and preferences, which can make your coding experience more efficient and enjoyable.

Whether you are a beginner or an experienced developer, a programmer's editor can be a valuable tool in your toolkit.

It is well worth getting to know the features of your editor well. Some of the benefits you'll get from this are:

Increased productivity: The more familiar you are with the features of your programmer's editor, the more quickly and efficiently you will be able to write and edit code. For example, if you know how to use keyboard shortcuts, you can avoid using the mouse, which can save you time and reduce strain on your hands. Similarly, if you know how to use the auto-completion and code navigation features, you can quickly find and edit the parts of your code that you need to change.

Improved code quality: Knowing the features of your programmer's editor can also help you write better code. For example, if you know how to use the syntax highlighting and code outlining features, you can easily spot errors and inconsistencies in your code. This can help you avoid mistakes that could cause your code to break, and can help you ensure that your code is clean, well-structured, and easy to read.

Greater flexibility: The more you know about your programmer's editor, the more flexible you will be in terms of the types of projects you can work on. For example, if you know how to use the debugging tools, you will be better equipped to handle complex projects that require you to find and fix errors in your code. Similarly, if you know how to customize the editor, you can adjust it to fit your specific needs and preferences, which can make your coding experience more enjoyable and effective.

Enhanced learning and problem-solving skills: Knowing the features of your programmer's editor can also help you develop your broader software development skills. For example, if you know how to use the code navigation and outlining tools, you will be better able

to understand the structure and organization of your code. This can help you learn new programming concepts more quickly, and can make it easier for you to solve complex problems.

Editor Features

- Syntax highlighting
- Auto-complete
- Indentation
- Folding

IDE Features

- Deeper knowledge of languages
- Compile / run
- Debugging
- Source code control
- Multi-file projects

More Features

- Everything is configurable
- Everything is extensible
- Everything is programmable

2.2 Operating system

Your operating system is the interface between you and the processing power of your computer. You're using it in some way or another every second that you're using the computer. I don't care if you're developing on Windows, MacOS or Linux (or even some more obscure OS), but I think that you should put some effort into getting the most out of your operating system.

In the 1990s and early 2000s, people seemed to place great importance in the operating system they used. It was like joining a tribe. When Macs started running on a Unix-like OS, I saw a large

number of developers switch to using Macs, which had previously been seen as a machine for designers. Mostly, they switched from Linux which, back then, could still be a little tricky to get working well on a desktop or laptop. Windows users were often disparaged as people who were unable to use a “real” operating system. These days, that all seems a little silly as a lot of development takes place using virtual machines or containers (of which, mush more later) and your operating system is largely used for running a browser and launching your virtualised development environments.

- Your operating system is where you live
- Get to know its features
- Design your working environment
- Automate things

Operating Systems

- What do you use your operating system for?
- Web browsing
- Reading email
- Social media
- Development?
    - Virtual machines
    - Containers

Configure Your OS

- Make it as comfortable as possible
- Window decoration
- Window behaviour
- Virtual desktops

Automate Common Tasks

- Automate anything the third time you need to do it
- Learn your OS’s scripting mechanism
- Linux / Unix shell scripting
    - Aliases

    - Functions Powershell

SSH Connections

- You will need to connect to other systems
- Secure shell (ssh)
    - ssh-agent / ssh-add
    - ~/.ssh/config
    - Keepalive
    - Username
    - Agent forwarding
- Putty does all of this too

2.3 Source code control

When I first started in the industry, source code control was a relatively new idea. At one large bank I worked at in London, I was part of an infrastructure team and one of our projects was to convince the development teams to make more use of source code control. One team leader didn't agree with me. He said "I understand exactly what problems source code control is there so solve - but it's a problem that my team doesn't have." Later on, I spent some time working in that team and realised what he meant. They used tarballs with dates in their names to keep track of the different versions of their codebase.

Times have changed now though. These days, no sane developer is going to start a project without using a source code control system. And currently (and for the last ten years or so) the most popular source code control system has been Git. There would need to be a pretty good reason for a software project to choose anything else.

- Why aren't you using git?
- Why aren't you using Github?
    - Microsoft
- Understand how your source code control works

Switch to Git

- Git has been the de-facto industry standard for ten years
- More powerful
- More flexible
- Harder to use
- Different paradigm
- “Optimistic concurrency”
- Good merge tools

Git Advantages

- Distributed model
  - Off-line access
- Branches are easy and quick
- Rebasing
  - Interactive rebasing
- “Fetch” vs “Pull”
- Safety net

Git in the Cloud

- Don’t run your own Git infrastructure
- Other people are better than you at doing that
- Social coding
- Github
- Gitlab
- Bitbucket

3 Adjacent technologies

There are many technologies that aren't actually part of a programmer's core skill set, but that are very closely associated with programming and which a serious programmer will take a close interest in. In this chapter, we'll look as some of the most useful of these technologies.

3.1 User interface design

3.2 Search engine optimisation

If the system you work on powers a web site, then it's likely that you would like to get as many visitors as possible to the site. And that means you will want the site to rank as highly as possible in the Google results for various search terms. And that will lead you to the world of search engine optimisation.

On larger projects, you will probably have a separate SEO team to work on this. They will monitor your site's performance and suggest changes to be made by the developers. But on small projects the developers might be expected to carry out this work. And, even when you do have experts on-hand to suggest the required changes, it's still worth knowing a little about what can affect your site's ranking in Google.

There are basically two strands to SEO. There's the "content marketing" side, which is all about tweaking the text on your site so that it contains enough of the keywords you are trying to rank for. This is a delicate balance between ensuring that Google recognises the keywords in your text and ending up with content that is still readable English. I'm sure we've all read Google keyword-heavy pages where the Google keywords appear so frequently that the Google keywords overwhelm the text and it becomes hard to understand because of the repetition of the Google keywords.

Other than cautioning you to do your best to keep your text readable, I'm not going to talk about that side of SEO; I'm going to concentrate on techinical, or "on-page" SEO. This is all about small tweaks you can make

to the structure of your page so that Google stands more chance of understanding what your content is about.

3.2.1 Semantic HTML

HTML mark-up is supposed to be semantic. That is, you use HTML to mark-up the *meaning* of the various sections of your page. It is the browser (with help from your CSS files) that decides what your page looks like. For example, the most important header on your page should be marked as H1, the next most important headers should be H2, and so on. Don't make the mistake that a web designer that I worked with twenty years ago made. Because he wanted the main header to be the size that his browser showed an H3, he made the main header an H3. To be fair to him, CSS was a relatively new technology and I don't think he really got it.

Another example from the days before CSS is the "font" tag. In the days before CSS (and, to be honest, for a depressingly long time after the introduction of CSS) HTML pages were full of font definitions. Every block element needed its own font tag. And when you wanted to change the font that your page used then you needed to edit every single one of them.

HTML uses tags that can be used to describe the parts of your text - headers, footers, paragraphs, sections. HTML5 added a lot more that many people don't seem to be aware of. When you mark up your page correctly with these tags, Google can use them to work out the structure of the page. If you fail to do that, then Google is just guessing at the relationship between the various pieces of text on your page - and the algorithm might guess wrong.

3.2.2 URLs

A Universal Resource Locator (URL) is the web address of your page. Often, URLs are disguised from users as, for example, they are hidden behind the text in an HTML link. You might think, therefore, that URLs aren't very important. But they can be important to Google, so they should be important to you. You should be thinking carefully about your URL structure.

In short, a good URL should be meaningful, hackable and permanent. Which is the most memorable of these two URLs?

- https://morethanco.de/upcoming-public-training
- https://morethanco.de/?p=68450021

Clearly, the former is more meaningful. It's easier to understand and, therefore, easier to remember. It's not only more meaningful for humans; it's also more meaningful for Google. Google will try to extract from a URL useful information about the page that the URL refers to. And the second example gives no clue at all about what will be found at that address.

In general, simple URLs are to be prefered. Use words instead of meaningless strings of numbers. It's also worth trying to avoid having anything in your URLs that gives away information about the technologies that are driving your your site (like `some-page.php` or `another-page.asp`). The only people who might find that information useful are unscrupulous people trying to look for backdoors into your server.

What does it mean for a URL to be hackable? Well, take an (imaginary) URL like:

- https://morethanco.de/news/2020/03/some-seo-tips

For a start, you can extract useful information from the URL. We can see what the article is about, the approximate date of publication and, also, that it was a news story. But we can also get clues to other URLs that might be interesting. It seems likely that visiting https://morethanco.de/news will show us a list of the most recently published news stories. And I'd expect that https://morethanco.de/news/2020 will show all of 2020's news and https://morethanco.de/news/2020/03 will give me all of the stories published in March 2020. A power user will certainly try visiting those pages. And Google will expect them to exist too. So when your URL structure implies the existence of a page, you should ensure that the page actually exists.

Finally, a good URL should be permanent. I'm addicted to Facebook's Memories application. Every day, it shows me a list of Facebook items that I posted on that day in previous years. And it's depressing how many of the interesting links that I posted for the amusement of my friends no longer exist. These dead links fall into two categories.

First there are the sites that just no longer exist. Perhaps a company closed down or a developer got bored of a side-project and just let the domain lapse. One particular friend has used a number of different domains for his blog over the last fifteen years. And each time he moves to a new domain, he doesn't move the old content over. I'd be gutted to lose all of the content, but some people have a different relationship with their writing. There's nothing that can be done in situations like these. If you no longer own the domain then you can't put anything in place to respond to request to that domain.

Then there are the pages where the domain still exists, but some individual pages no longer exist. A good example is an arts venue that chooses not to maintain pages for previous events. I often come across an old Facebook post that contains a link to a page about a gig I was going to or a exhibition I was planning to see. But because the event was ten years ago, the venue doesn't see the value in maintaining that page (or, more likely, just doesn't have the resources to maintain old pages). Probably, they've reorganised the site a couple of times in the intervening years - and old event pages just didn't make the cut to be migrated to the new structure. It's a shame, but you can't argue with the economics.

But permanence does matter. Longevity is one of the criteria that Google uses to measure the importance of a page. And you don't want to be serving 404 "page not found" errors to users if you can avoid it. Far better to redirect users to your home page or (even better) a search page where they can look for the content they're trying to find. If you rearrange your site, then map the old URLs to the new ones and set up mechanisms to redirect the users to the new version of the page. If a page has been removed from your site for good reasons and you don't want Google to care about it any more, then return a 410 "gone away" response.

3.2.3 Structured data

Earlier, we talked about using semantic HTML in order to make it easier for Google to understand what your content is about. There's another step you can take and that's to use structured data on your web pages. There are two types of structure data that Google currently recognises - microdata which is built into the existing HTML and JSON-LD which is another view of your data which you embed into your "head" tag.

Microdata is a way to extend HTML so that it contains more information describing the data on the page. For example, you might have some HTML that displays information about a film:

```
<div class="film">
  <h1>Avatar</h1>
  <div class="director">
    Director: <span>James Cameron</span>
    (born <time>August 16, 1954</time>)
  </div>
  <span class="genre">Science fiction</span>
  <a href="/movies/avatar-theatrical-trailer.html">Trailer</a>
</div>
```

Your CSS would use a combination of the HTML tags and the classes to tell a browser how to display this data. Note, for example, that we've used a "div" tag with the class "film" to denote the section of the page that describes a film. But that's just a convention that we've invented; it's not standard. Some other company might use other conventions - for example a class of "movie". Google can't be expected to unpick all of these local conventions for defining data.

Microdata is a way to standardise this markup. A microdata version of the same information might look like this:

```
<div itemscope itemtype="http://schema.org/Movie">
  <h1 itemprop="name">Avatar</h1>
                  <div         itemprop="director"         itemscope
itemtype="http://schema.org/Person">
    Director: <span itemprop="name">James Cameron</span>
      (born <time itemprop="birthDate" datetime="1954-08-16">August
16, 1954</time>)
  </div>
  <span itemprop="genre">Science fiction</span>
              <a       href="/movies/avatar-theatrical-trailer.html"
itemprop="trailer">Trailer</a>
</div>
```

We've added various attributes to our HTML tags. There are three new attributes.

- itemscope defines a new data item.
- itemtype defines the type of the new data item. This will be a URI, pointing to a web address that defines that data item type.

- itemprop defines a property of a data item. The definition of the data item will tell you which properties a particular data item type should or may have.

So, in our example we have a top-level object which is a movie as defined at http://schema.org/Movie. The movie has four properties - a name, a director, a genre and a trailer. The director is defined as being an instance of a new data item type called a person. Our person object has two properties - a name and a birthDate.

Google (or, indeed, anyone who is interested) can look at the source code of our web page and can immediately get a far richer (and standardised) view of our data. As you might expect, there are dozens (probably hundreds) of data item types defined and you can model very complicated relationships between the various pieces of information on your web page. It's worth spending a few hours browsing the https://schema.org/ web site where these types are defined. There is an incredibly rich selection of types available.

Microdata is relatively easy to add to your web page. Well, there's the complexity of mapping your information onto the types and properties available, which can take a while. But once you have that mapping work done, it's not hard to add the relevant mark-up to your existing HTML. The structured data is woven tightly around the HTML. In contrast, JSON-LD (LD stands for "Linked Data") is a rather different approach where very similar structured data is embedded in a completely different part of the HTML.

In order to build a JSON-LD representation of our page, we still have to go through the same mapping process, using the same set of data item types. But when we have finished the mapping, we build a JSON data structure instead of embedding the information within our HTML. The JSON-LD version of our film data might look like this:

```
<script type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@type": "Movie",
  "name": "Avatar",
  "director":
  {
    "@type": "Person",
    "name": "James Cameron",
```

```
    "birthDate": "1954-08-16"
  },
  "genre": "Science fiction",
  "trailer": "/movies/avatar-theatrical-trailer.html"
}
</script>
```

You will recognise the two data items (a movie and a person) and all of the properties that we used in the microdata version. This is just another representation of the same information. This chunk of JSON should be inserted somewhere in your HTML page. Google recommends putting in the “head” section, but the “body” sections works too if that is easier for you.

Google provides a structured data testing tool at https://search.google.com/structured-data/testing-tool. You can give it the URL of your page or paste in a piece of structured data and the tool will show you all of the data items and properties it can extract from the sample. It will also tell you about any problems it finds with your structured data. This usually consists of information about missing properties that you should consider adding.

3.2.4 Open Graph

Open Graph is a standard that was defined by Facebook in order to make it easier to share external content on its site. It consists of a number of extra header tags that give useful information about the page. This means that, for example, Facebook can find an image to display alongside a link without having to parse the entire page. A simple set of Open Graph tags looks like this:

```
<meta property="og:title" content="Some SEO Tips">
<meta property="og:site_name" content="More Than Code">
<meta property="og:type" content="article">
<meta                                                property="og:url"
content="https://morethanco.de/news/2020/03/some-seo-tips">
<meta                                              property="og:image"
content="https://morethanco.de/images/seo.png">
<meta property="og:description" content="SEO is important. Get some
useful tips here.">
```

Most other social media sites will also make use of this information. Twitter has a slightly different concept called a “Twitter Card”. They will

use the Open Graph data, but it’s worth adding the following Twitter-specific tags as well.

```
<meta name="twitter:card" content="summary" />
<meta                                          name="twitter:image"
content="https://morethanco.de/images/seo.png"/>
```

As far as I know, Open Graph tags don’t, in themselves, increase your site’s Google ranking. But by tuning how links to your appear when they are shared, you can encourage more people to share your content and popular content is prefered by Google.

3.2.5 Sitemaps

A sitemap is a file that sits in the top level directory of your web and contains a list of all of the pages on your site that you want Google to crawl. It is an XML file. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://morethanco.de/news</loc>
    <lastmod>2020-03-10</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>https://morethanco.de/services</loc>
    <lastmod>2020-03-01</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.5</priority>
  </url>
</urlset>
```

The top level element is “urlset” and, within that, you have a number of “url” elements. Inside the “url”, you only need the “loc” element (to tell Google where the page is) but you can also give Google hints on how to crawl your site by including the last modification date, the (approximate) change frequency and a number between zero and one indicating how important you think the page is.

A sitemap file has a limit of 50,000 URLs. For many sites that is plenty, but if you have a particularly large site, you can have multiple sitemaps and

also a sitemap index file which shows Google where to find them. The contents would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <sitemap>
    <loc>https://morethanco.de/sitemap1.xml.gz</loc>
    <lastmod>2020-03-10</lastmod>
  </sitemap>
  <sitemap>
    <loc>https://morethanco.de/sitemap2.xml.gz</loc>
    <lastmod>2020-03-09</lastmod>
  </sitemap>
</sitemapindex>
```

The fields in this file should be self-explanatory, but notice that the sitemap files can be zipped in order to save space.

3.3 Canonical pages

Google doesn't like duplicate content. If different URLs on your site point to pages that look substantially the same, then Google might think you are trying to game the system and could penalise your site. There may be good reasons to have different URLs for the same page, so Google allows you to mitigate this by telling them which is the canonical version of the page.

Imagine you're running a site that lists athletics world records. I can go to the URL https://your-records-site/ and it will show me a list of the current world records in a number of events. Or I can go to https://your-records-site/2000-01-01 and it will show me the same records as they stood on 1st January 2000.

World records don't change very often. It's likely that the page I get for https://your-records-site/2000-01-01 is identical to the one for https://your-records-site/1999-12-31. In fact most of your pages would be duplicates of the pages just before and after them. And Google wouldn't be happy about that.

To get round this, we need to generate a list of canonical pages. That is, a list of the dates when a record changed. The pages for those dates will be the only unique pages on the site. On pages that aren't canonical, we can

include a header tag, telling Google that we know about the duplication but that we only want them to index the unique pages on the site. The tag looks like this:

```
<link rel="canonical" href="https://your-record-site/1999-06-30">
```

Here, we've assumed that a record changed on 30th June 1999. So any page for date between then and the next record change date would include this tag. This is us saying to Google "yes, I know that this page looks a lot like the one for 30th June 1999, so please don't index this one but just index that other page instead". And that will prevent Google from being suspicious of the duplication.

There are many reasons why you might have duplication. Some others include:

- You're in the process of moving from one domain to a new one (the pages on the old domain should have a canonical tag pointing to the version on the new domain).
- You're switching your site from HTTP to HTTPS (the HTTP versions should have a canonical tag pointing to the HTTPS version).
- You have a page of lists and various filtering and sorting options that are included in the URL can result in similar pages (all version should include a canonical tag pointing to the unfiltered and unsorted version).

It's perfectly possible for a page to include a canonical tag that points to itself. This often makes the developer's life easier as he can include a canonical tag on every page - whether or not that page is, itself, canonical.

3.4 Other features

There are a number of other SEO tweaks that you can make to your site.

- robots.txt: The main use of a robots.txt file is to give web crawlers advice about which parts of your site they should crawl. But there are other things that you can do in the file. For example, you can use the "Sitemap:" statement to tell crawlers where you site maps are.

- Response headers and meta tags: There are a number of response header that you can use to control how your site is crawled. Each response header has an equivalent meta tag that you can use if you're not able to control your site's response headers. The "X-Robots-Tag:" header can be set to "index" or "noindex" along with "follow" or "nofollow". The former controls whether a crawler will index the current page and the later controls whether links in the current page should be followed.

- Security: For some years now, all browsers have been trying to gently nudge web site owners towards serving their content over HTTPS instead of HTTP. Recently, that has been stepped up as some browsers have started to display a warning on HTTP pages. And Google gives a ranking boost to pages served over HTTPS. You should be using HTTPS and when someone tries to access your site over HTTP, you should redirect them to the HTTPS version.

- Mobile first: A couple of years ago, Google announced that they would start crawling mobile versions of web sites and using those results in preference to the desktop version. They've been taking this slowly and emailing site owners when they are taking a mobile-first approach to their sites. But later this year, they plan to make this switch for every site. Your site should give equivalent experiences to people using mobile devices and desktop devices. The best approach is probably to have a responsive site that automatically adjusts to the size of the display it is being used on.

3.4.1 How well are you doing?

Google supplies tools that you can use to see how well your site is performing. Most people have probably heard of Google Analytics. You just drop a small piece of Javascript on every page of your site and Google will give you as much information as you need about the people who visit your site - where they came from, what they were searching for, how long they stayed, which pages they visited. It will also give you lots of demographic information about them.

But there's is also the Search Console. This is where Google tells site owners how it has crawled their sites. You register your sites with this service and, within days, you will start to get reports on any problems that

Google's crawler has found on your site. It will also tell you how many pages Google has crawled and how many of those are currently in Google's index. This is an essential tool for webmasters who want to get as much information as possible about how Google sees their site. There was an older version of this site called "Google Webmasters Tools", but this new site has been replacing that over the last two or three years.

3.5 Data storage

All systems use some kind of data. And that data is probably going to need to be stored somewhere or exchanged with some other system. So you're going to need some kind of database or data files. Do you just use what everyone else uses, or do you have enough knowledge to choose the right tool for the job? Do you, for example, know when you might choose a NoSQL database like Redis or MongoDB over a more traditional relational database system?

Are you constrained in your data interchange format choices? Many people would prefer something structured, but human-readable like JSON or YAML, but you might be communicating with a system that requires a particular format, for example XML.

3.5.1 Relational databases

A very large proportion of computer systems (probably most of them) will have some kind of relational database system to store their data. A large, commercial system will probably use a proprietary system like Oracle, but it's very common to see open source databases like MySQL, PostgreSQL and SQLite being used as well.

You might be surprised to hear that the most widely-used relational database system today is SQLite. This is because it's a very lightweight implementation of the relational model and is therefore the database system of choice for smartphones. Pretty much any app that you install on your Android or Apple phone will be using an SQLite database to store its data.

3.5.1.1 Introduction to relational databases

The idea of relational databases started to become popular in 1970 when IBM employee, Ted Codd, published his paper “A Relational Model of Data for Large Shared Data Banks”. This used a relatively obscure branch of mathematics, called relational calculus to derive a mechanism for storing large amounts of data.

In Codd’s paper, data is stored as tuples which are gathered together in relations. A tuple contains all of the data items about one particular object (for example the name, date of birth and sex of a person) and a relation groups together all of the data about objects of the same type (for example data about all of the people your system is interested in). These days, it is more common to talk about a table which holds data about people and a single row in that table which holds the data about an individual person.

A table is defined by the set of attributes (or columns that it contains). Each attribute has a name and a data type. The data type defines the valid values that can be stored in that column. We use a data definition language (DDL) to define a table. The definition for our example table storing data about people might look like this:

```
CREATE TABLE person (
  name CHAR(50),
  date_of_birth DATE,
  sex ENUM('M', 'F')
);
```

The table is called “Person” and it has three columns.

In practice, you will also want some kind of unique identifier for each row in the database. There are various mechanisms for allocating those. The simplest (and probably the most common) is to allocate the next integer in an ascending sequence as a record is inserted into the table. So the table will probably look more like this:

```
CREATE TABLE person (
  id INT,
  name CHAR(50),
  date_of_birth DATE,
  sex ENUM('M', 'F')
);
```

Your DDL will also allow you to define more advanced aspects of your data. For example, there might be inherent uniqueness constraints in your data. There are no obvious uniqueness constraints in our current table (it's perfectly possible for people of the same sex and with the same name to be born on the same date) but if we also included a person's government-issued tax identifier, then that would need to be unique.

So the table definition above says that we will be storing data about people. The data we will store about each person is as follows:

- The person's identifier, which is an integer
- The person's name, which consists of up to fifty characters
- The person's date of birth, which must be a valid date
- The person's sex, which is an enumerated value that can only be 'M' or 'F'

As it stands, our table doesn't insist on any of those values being filled in. In a relational database, a column without a value is said to contain the special value "NULL". We can change our table definition to prevent null values from appearing in certain columns. We do that by adding "NOT NULL" to the table definition.

```
CREATE TABLE person (
  id INT NOT NULL,
  name CHAR(50) NOT NULL,
  date_of_birth DATE NOT NULL,
  sex ENUM('M', 'F') NOT NULL
);
```

Here, we've said that none of our columns can be empty. An important part of the database design process is deciding which data items are optional. And now we can start to see some of the complexities of data design. We know that the id column must be mandatory (because we're going to assign that value to each record that is added) but what about the others? It seems unlikely that we'll be storing data about a person without a name, so it's sensible to make that value mandatory. But what about the date of birth and sex? Do we really need those values for everyone? Do we actually need those values for anyone? Of course the answers to these questions will depend completely on what this database is being used for. If it's being used to calculate tax payments, then you might well need to know the age and sex of the person you're dealing with. If it's

storing genealogical research, then it's quite possible that you want to store data about people whose date of birth you don't know.

The table also contains a good example of where theoretical database design just doesn't work in the real world. We've said that a person's sex can only be 'M' or 'F'. But that's not how it really works. Some people are intersex. Some people are non-binary. Some people will be transitioning. Some people just won't want to tell you. What do you do in those situations? Do you add a whole list of other options to the ENUM? Do you just add 'O' for 'other'? Do you make the value optional? Only you know what your data is being used for, so only you can answer those questions. But please try to be sensitive.

- Which database vendor do you use?
- Why did you choose MySQL?
- Would PostgreSQL be better?
- How does it scale?
- Replication vs sharding
- How well do you know SQL?
- Which RDBMS features do you use?
- How vendor-specific are they?
- Vendor-specific SQL extensions prevent migration
- Do you use stored procedures?
- Do you use triggers?
- How are your databases split?
    - One database? Per application? Per service?
- Who owns the database?
- How do you change the schema?
- How do you know which version of your schema is installed?
- Do you use a database migration system?
- What problems does that lead to?

3.5.2 NoSQL

- Which NoSQL database do you use?
- MongoDB, Redis, CouchDB, Memcached
- Pros and cons for each - understand the differences
- Would you be better off with an RDBMS?

3.5.3 Data files

- What formats are your data files in?
- How are they processed?
- Are you using the best available tools?
- Javascript Object notation
- Popular data interchange format
- AJAX
- Decode to native data structures
- Encode from native data structures
- Popular choice for APIs

JQ

- General purpose JSON utility
- Reformat JSON
- Extract data from JSON
- Powerful query language

XML

- Still popular in so areas (SOAP)
- You will need to deal with it
- Find good tools
- XPath is great
- XML query language
- Extract data from XML documents
- Standard syntax

3.6 Networking

There are very few systems these days that don’t use networking in some way. Some of the more obvious networking protocols that you might come across include:

- SSH (secure shell) and SCP (secure copy) for getting shell access to remote servers and for copying files to and from them.
- SMTP (simple mail transfer protocol) and IMAP (internet message access protocol) for sending and receiving email.
- HTTP (hypertext transfer protocol) and its secure version, HTTPS, for handling web requests.
- DNS (domain name system) for translating human-readable domain names to computer-useable IP addresses.

Other, less well known protocols that you might see include:

- FTP (file transfer protocol) has been superseded by SCP - as it's more secure. If you come across somewhere that still uses FTP, you might question their attitude to security.
- POP (post office protocol) is an older mail retrieval protocol.
- NTP (network time protocol) for ensuring that all of the servers connected to a network have their clocks in sync.
- SNMP (simple network management protocol) is a way to gather information about the devices that are connected to a network.

In most cases, a networking protocol is a well-defined series of requests and responses that take place between a client and a server. The client sends a request to the server and the server responds with some information. This cycle takes place on a certain "port", which is a numbered communication channel where a server will listen for requests. Each protocol has one or two well-known ports that it runs on. An SSH server, for example, will listen on port 22 and an HTTP server will listen on port 80.

The instructions that drive these conversations are usually plain text. It can often be interesting to manually play the part of the client yourself. One common tool for doing this is called `telnet`. This command line program has largely been superseded by `ssh` (because it's more secure) but it can still be installed on most systems.

If you run `telnet some.server` then you connect to "some.server" on port 23 (the default port for the telnet protocol) but you can give it another port number as a second argument and you will then be talking directly to whatever server is listening on that port. For example, to talk HTTP to web server, telnet to port 80:

```
$ telnet morethanco.de 80
Trying 185.199.109.153...
Connected to morethanco.de.
Escape character is '^]'.
```

You can then type any request that you would expect an HTTP server to understand (note that HTTP requires an empty line to end the request, so you need to hit the enter key twice):

```
HEAD / HTTP/1.1

HTTP/1.1 404 Not Found
Server: GitHub.com
Content-Type: text/html; charset=utf-8
ETag: "5cb0f185-239b"
[... more omitted ...]
```

It doesn't happen everyday (or even every week) but when I really want to know what's going on in a network interaction, I often find it useful to debug it using `telnet`.

3.6.1 Seven-layer model

Networking is traditionally described as containing seven layers. This is useful as for a specific problem, you usually only need to think about one or two of these layers at a time. As a developer, that's probably the application layers - most of the time, you can assume that all of the other layers Just Work.

It is, however, useful to know what the other layers are in order to hold a meaningful conversation with any network engineers who might be helping you to solve a problem. The seven layers are:

1. Physical layer. This is the cables, boxes and wifi transmissions that actually make up the physical network.
2. Data link layer. This is the protocol that provides data transfer between two adjacent nodes in your network.
3. Network layer. This is the protocol that allows data to be transmitted between different physical networks.
4. Transport layer. This is the protocol that allows data to be transmitted from a source host to a destination host.
5. Session layer. This is the protocol that defines seemingly-persistent connections between clients and servers.

6. Presentation layer. This is the protocol that formats application data for transmission across a network.
7. Application layer. This is where all of the application-specific network protocols we discussed above sit.

3.6.2 HTTP

HTTP, and its more secure cousin, HTTPS has become one of the most ubiquitous networking protocols in our modern world. Not only is it used whenever someone visits a web page, but it's also the most common mechanism used for API calls and most "Internet of Things" devices will use HTTP to communicate with each other.

At its core, HTTP is a simple request/response cycle. An HTTP client (which is often a browser) makes a request to a server and the server returns some data. Both the request and response consist of a number of headers separated from the body by a blank line. We saw a simple request/response example above. The request looked like this:

```
HEAD / HTTP/1.1
```

And the response looked like this:

```
HTTP/1.1 404 Not Found
Server: GitHub.com
Content-Type: text/html; charset=utf-8
ETag: "5cb0f185-239b"
[... more omitted ...]
```

The `HEAD` command is the simplest of the HTTP request types and it generates one of the simplest responses. It specifically only asks for the header portion of the response. And, because this was a request that I typed in manually in `telnet`, I gave it the bare minimum request - just a command, with no headers.

The full specification of a request is:

- A request line, consisting of a command (`HEAD`), the path of a resource on the server (`/` - which is the top-level resource) and the version of HTTP to use (`HTTP/1.1`).
- A number of optional header lines. These consist of a header name, a colon and a value. For example, you might tell the server that you

only want HTML returned using an “Accept” header - `Accept: text/html`.
- A blank separator line.
- A optional body which can contain data that you wish to transfer to the server (for example, a file upload).

A response is very similar. In place of the request line, it has a status line which includes the HTTP version, a status code and a brief message. The header lines will define details like the content type (`Content-Type: text/html`) and the body will contain the actual data for the resource (with the obvious exception of the response to a HEAD request which will only contain headers).

The response status codes fall into five groups. Each code is a three-digit number and the first digit tells you which group the code belongs to.

- 1xx (Informational): You don’t see many of these.
- 2xx (Success): These tell you that the server accepted and processed your request successfully. The most common of these is “200 OK”.
- 3xx (Redirection): These tell you that you should make an additional request to an alternative URL which will be in the “Location” header. In most cases, a browser will make the redirected request for you.
- 4xx (Client error): These tell you that you have made a mistake in your request which you need to correct before resubmitting the it. The most common of these is “404 Not Found”, but “403 Forbidden” is also seen quite often when you don’t have permission to access a particular page.
- 5xx (Server error): These tell you that something has gone wrong on the server and there is probably nothing you can do about it. The most common is “500 Internal Server Error” which is pretty generic.

There are a number of different HTTP commands available. The most commonly used ones are:

- GET: returns a representation of the resource at the given path.
- POST: sends data to the server in the body of the request. The path in the request line will tell the server what to do with the data.
- HEAD: returns only the headers of the equivalent GET request.

You’re doing well if you could also name the next three:

- PUT: is similar to POST, except POST is expected to contain new data to be stored on the server and PUT is expected to contain a replacement for existing data.
- DELETE: asks the server to delete a resource.
- PATCH: supplies partial modifications to a resource.

And almost no-one (in my experience) knows about these:

- OPTIONS: asks the server to return a list of the commands that the it supports for the given path.
- TRACE: returns the request that the server received. This is useful for debugging as you will be able to see any modifications that might have been made to the request by intermediate servers.
- CONNECT: converts the transient HTTP connection to a TCP/IP tunnel. This one is pretty obscure.

A standard web application will usually only use GET (to display a web page) and POST (to send the contents of a web form). This is slightly unintuitive as you can find yourself using a GET or POST request to delete data from the server. If you're writing an API, however, you should make use of a wider range of HTTP commands. If you consider your API as a query tool to a database, then there are four operations you will need to carry out on the data stored in that database.

- Create - to add a new data object to the database.
- Read - to retrieve an existing data object.
- Update - to change the attributes of an existing data object.
- Delete - to remove a data object from the database.

These are known as the CRUD operations and they map rather nicely onto four of the HTTP commands.

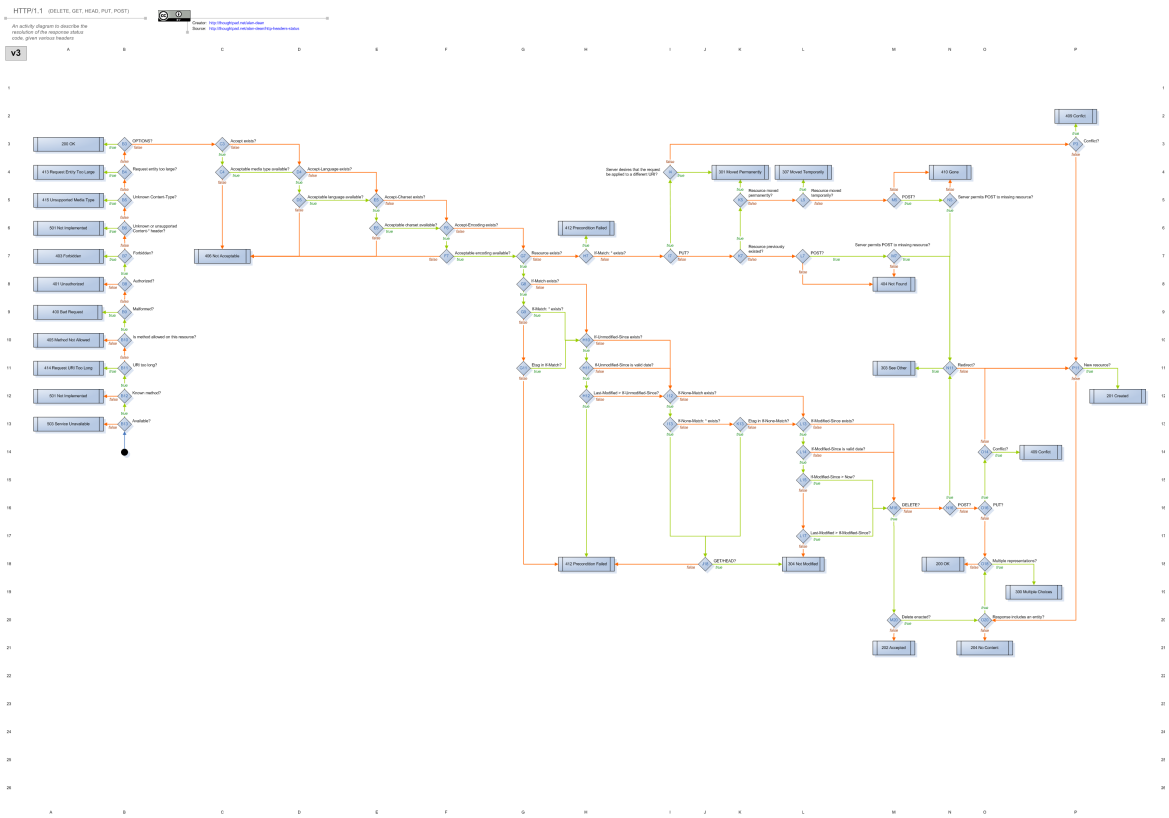
- POST - to create a new data object.
- GET - to read an object.
- PUT - to update an object.
- DELETE - to (rather obviously) delete an object.

One common-used approach to creating an API is called Representational State Transfer (REST for short). One of the tenets of REST is to use the four HTTP commands listed above to handle the CRUD operations on your data. So if, for example, you had an API which

contained details of films you would want to support the following API calls.

- POST /films - to create a new film. The request body would contain a representation of the various attributes of the new film. The response to this request would echo back this representation, adding any new attributes that were added as the record was created - this would almost certainly include the identifier for the new record and the URL where the new record can be accessed.
- GET /films/999 - to read details of the film with the identifier 999. This would return a representation of the film (which would probably be very similar to the representation returned by the original POST request).
- PUT /films/999 - to update the details of the film with the identifier 999. The body of the request would include a representation of the attributes of the record that have changed.
- DELETE /films/999 - to delete the film with the identifier 999.

You can also make more precise use of HTTP status codes when writing a REST API. For example, instead of just returning “200 OK” from a POST request, you should return “201 Created” to indicate that the object was created successfully. A web framework called webmachine provides a comprehensive mechanism for returning the correct status code for a given sitution. They provide a state transition diagram that nicely demonstrates the complexities.



REST State Transition Diagram

3.6.3 HTTPS

HTTPS is a secure version of HTTP. It supports everything that HTTP does and builds a secure socket layer (SSL) on top of that. With standard HTTP, all of the data that passes between the client and the server is unencrypted and anyone who could intercept that traffic could see the information. With HTTPS, the data is encrypted using keys that the client and server share, so that only the client and the server can see the data.

At the very least, whenever a user is sharing confidential information with your web site (passwords or credit card details, for example) that form should be served using HTTPS. A few years ago, browsers started to indicate that a page was secure by displaying a padlock icon in the URL bar but, more recently, this has started to switch to displaying a warning when a page is not served over HTTPS. Presumably, you don’t want your users to see these warnings when they visit your site, so the best approach is to server all of your pages over SSL and to redirect anyone requesting a page over HTTP to the secure version. With organisations

like Let’s Encrypt issuing SSL certificates for free, there’s really no reason not to do that.

3.7 Testing

- Do you write unit tests?
- Does your system have automatic integration tests?
- How are you tests run?
- Automatically?

Unit tests

- Never program without a safety net
- Does your code do what it is supposed to do?
- Will your code continue to do what it’s supposed to do?
- Write unit tests
- Run those test all the time
- Code that exercises an individual “unit” of your code
- Provide known inputs
- Look for expected outputs
- “Bottom-up” testing

Test-driven development

- Have you tried TDD?
- What did you like?
- What didn’t you like?
- Will you try it again?
- Tests are a specification of your system
- If the tests pass, the program is finished
- Know when to stop

Behaviour-driven development

- Let end-users define test in their language
- Simple domain-specific language
- Framework converts user tests to runnable code
- Cucumber

Test automation

- What test automation frameworks do you know?
- What test automation frameworks do you use?
- Selenium (WebDriver?)

Test coverage

- How good are your tests?
- How much of your code is exercised by your test suite?
- Measure to improve

3.8 Logging, monitoring and alerting

- Centralised logging
- Log query tools
- ELK stack

3.9 Deployment

Sooner or later, your code will need to be deployed into an environment where it is available for your users to access. Previously, it was unlikely that, as a developer, you would have much to do with this deployment process, but the move towards DevOps personal being embedded in development teams makes it far more likely that you will need to understand this part of the process.

3.9.1 Deployment environments

It's likely that your code will go through a number of different environments before it gets into production. Not all companies will have the same set, but the four environments listed below seem to be fairly common.

- Development: This is where you will do your development. It will hopefully be an environment that only you are working on. Every developer should have their own development environment. This can be running locally on the developer's own machine or somewhere remote (on a share development server or, perhaps, in the cloud).

- Integration: Once a developer thinks a feature is ready for production, it will need to be tested on an integration environment. This is where

all of the features that are intended to be included in the next release can be tested togther. This will ensure that new features which have been developed in parallel since the previous release don't affect each other in strange ways.

- Staging: When a release candidate has been thoroughly tested in the integration environment, a release is built and deployed to the staging environment. This is built to be as much like the production environment as possible and is a place where you can test the release one final time to ensure that it works as expected.

- Production: This is the live environment from where your users access your server. Once your release hits this environment, it is officially released. There will normally be at least two instances of this environment. This is mostly for resilience (in case one environment goes down for some reason) but can also help with scalability and, sometimes, teams use multiple production environments to help with staged releases.

That's a lot of environments. One development environment for each developer on the project. Probably an integration environment for each team or each major component of the system. At least one staging environment and at least two production environments. Just keeping that number of environments up and running will be a major headache. Keeping them all in step with each other will be worse.

We also need to consider exactly how code is moved from one environment to the other and what quality gates we place at each transition to ensure that only working code gets moved through the process.

There's also the question of the databases for these various environments. It seems unlikely that you will want a full database dump on all of the development or integration environments. It's not just the size of the live database that could cause problems here, there's also the the fact that this database will contain personal data about your customers that the development team should not have access to.

But the live database has a richness of data that you will never find if you create a specific development database with mocked up data. Whoever creates the test data will never be able to reproduce the complexities of

the data that is found in your live database. So it is certainly useful to seed your development database from the production database in some way. In order to work round the two problems of database size and customer privacy, you will want to take a dump of the production database and massage it in two ways before making it available to developers.

1. Remove some percentage of the data. Perhaps only include data about 25% of your customers and your products.
2. Anonymise the data so that anyone getting access to the database cannot see confidential information about your real customers.

I've seen this approach taken in a number of companies. It is never as easy as you initially think it is. Getting this set up to work well is a major development project. But it's one that it is well worth taking the time to get right. Once you have got it right, you should run that process on a regular basis (perhaps weekly) and give your developers a single command that will overwrite their development database with the latest cleaned-up dump.

There is also the question of how frequently a developer updates their development system with an up to date version of this database. A production database system isn't static. As new features are developed, new tables are created in the database, columns are added to existing tables and the definitions of existing columns are changed. Your development process can become fragile if a developer is using an older version of the database schema.

I've often seen developers set up their development database in their first week at a company and then never refresh it. They will make the changes to their database that are required by the work they are doing. But unless they also apply the changes that other teams are also making, their database will slowly fork from the production version. How do you address that?

The best approach is probably to insist that all schema changes are carried out using database migrations. A migration is a small program that makes a set of changes to a database schema. Usually this will be the set of changes required to add an enhancement or fix a bug. A migration will include the code to apply the migration to the schema alongside the code to roll it back. The migrations for a system will all be numbered in an ascending sequence and the source code for the migrations will be stored

in an easily-accessible place (perhaps in a directory that is part of your source code repository).

The database itself contains a table which has data indicating the current schema version number and migrations are applied using a program which takes as an argument, the version of the schema that you want to end up with. The program then compares the required number with the version currently on the database server and applies (or rolls back) the necessary migrations to get to the correct version. A nice extra feature is for your application's configuration file to contain the schema version that the current version of the application requires and for the application to check it's connecting to a database with the correct version of the schema each time it starts up.

The sequence of events then goes like this:

- A new developer joins the company and sets up a development system. This includes a development version of the database which is marked as being version 15 of the schema. The new developer checks out the latest version of the application and it requires schema version 15. The developer can therefore run the application against this version of the database.

- The developer spends a month working on projects that don't require any changes to the database. But, occasionally, other developers will make changes to the schema. When the developer merges a code branch that contains a new migration, the application shows an error and will not start until the database is running the correct version of the schema. This can be done by either running the migration program or by reloading the entire development database from the latest available dump.

- When the developer works on a project that requires a database change, he creates a migration for the changes and includes that in the commit with the code changes (and, also, an update to the configuration telling the application that it needs the new version). Once those changes have been released, that new migration (along with the code and configuration changes that require it) are in the main branch of the version control system and are available for any other developers.

By following a procedure like this, it's possible for developers to have their own development database which is easy to keep up to date with the released version of the schema. This minimises the changes of two developers making contradictory changes to the schema that aren't found until the release hits the integration or staging environments.

Deployment options

- Real hardware
- Cloud servers
- Virtual machines
- Containers

Real hardware

- Company owns or rents computers in a data centre
- Ops staff spend a lot of time in the data centre
- Commissioning servers
- Decommissioning servers
- No scalability

Cloud servers

- Servers commissioned from a huge farm
- Commissioned/decommissioned from a dashboard
- Or with an API
- Never see the actual hardware
- Easily scalable

Virtual machines

- Run one machine inside of another
- Many VMs on one real server
- Easy to commission/decommission
- Easy scaling (within the limits of the hardware)

Containers

- Docker
- Cut-down virtual machines
- Share a lot of the underlying OS

- Very quick to commission/decommission
- Layered architecture
- Modular
- Store configuration in source code control
- Treat servers as cattle, not pets
- Make it easy to create new ones
- Do that a lot

[Add stuff about serverless]

Environment configuration

- Keep your environments in step
- What software is installed
- Which versions of software is installed
- Avoid “works on my machine”
- Puppet, Ansible, Chef

Jenkins

- Continuous integration
- Run tests on every commit
- Continuous deployment
- Make releases available on every commit
- Quality gates
- Other, similar, products are available

Amazon web services

- Most popular cloud services provider
- Many services available
- Generic cloud servers (EC2)
- File storage (S3)
- Relational databases (RDS/Aurora)
- Caching (Elasticache)
- Many, many more

Other cloud providers

- Google Cloud Platform
- Microsoft Azure

- Oracle Cloud
- Rack Space
- Open Stack (self-hosted)
- Etc…

4 Software engineering and architecture

As a software engineer, it should be obvious that it’s a good idea to keep up to date with the latest ideas in designing software system. In this chapter, we’ll look at some modern ideas in this field.

4.1 Software development lifecycle

- Agile vs Waterfall
- Agile manifesto
- Scrum vs kanban vs extreme programming
- Know your tools
    - Jira
    - Code reviews

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Individuals and Interactions

- People are more important than processes
- Development teams know how to organise their work
- Business says what they need
- Development team works out how to achieve that
- Giving people more control and responsibility

Working Software

- Documentation is often out of date before it is finished
- Mismatch between business language and technical language
- Business often don’t know what they want until they see the options
- Build prototypes and discuss alternatives

Customer Collaboration

- Don’t just involve users when the project is specified
- Have user representatives as a part of the development team
- Instant response to developer questions
- More involvement leads to more business ownership of project

Responding to Change

- Requirements will change during the project lifetime
- Be flexible
- Plan for change

4.2 Programming paradigms

What paradigms does your system use?

- Procedural
- Object Oriented
- Functional
- Who made those choices?
- Why?

Procedural Programming

- Dumb variables
- Procedures, functions
- Hard to maintain
- Hard to scale

Object Oriented Programming

- Classes define both data (attributes) and behaviour (methods)
- Intelligent variables
- Encapsulation
- Subclassing
- Easy to maintain
- Easy to scale

Functional Programming

- Pure functions
  - Take inputs, return values
  - No global variables
- Immutable variables
- Prefer recursion to loops
- Easier to maintain
- Good for asynchronous programming

4.3 Design patterns

- “Gang of Four” book
- “Monolithic” architecture
- Service Oriented Architecture
- Microservices

The Power of Names

- Design patterns weren’t new or exciting
- Naming them was
- Catalogue of known techniques
- Easier to discuss
- Easier to share

5 The business

You don’t write code in a vacuum. Unless you’re lucky enough to be self-employed and working on your own projects, you will be working for a company that pays your for your work. This means that the code you’re writing will be used in some way by your employers in order to move forward some aspect of the company’s mission. This means that the more you understand about your company, the better-placed you’ll be to help the company achieve its goals.

In this chapter, we’ll look at some of the questions you could ask in order to gain a deeper understanding of your company and how it works.

5.1 What does your company do?

- What service does your company provide?
- Why do customers choose your company?
- What is the USP?

Making Money

- What is your company’s business plan?
- Where does the money come from?
- Are there multiple sources of income?
- How secure is that income?
- How much profit does the company make?
- How does that compare to your salary?

Competitive Marketplace

- Who are your biggest competitors?
- How are you ranked?
- Why do customers choose your company?
- Why do customers choose your competitors?

- What is your company doing to change the balance?

Mergers & Acquisitions

- Has your company bought any other companies?
- Might your company buy any other companies?
- What companies might it buy?
- Who owns your company?
- Might they sell it?

Stocks & Shares

- Is the company private or public?
- Can you buy shares?
- Is there an employee share scheme?
- What are the shares worth?
- Can you get share options?

5.2 What does success look like for your company?

- What is the company trying to achieve?
- How close are they to achieving that?
- How well-know are your company?

5.3 How do you contribute to the company's success?

- How crucial is your system to the company's success?
- Can you measure the financial contributions your work makes?

6 Personal development

In the previous chapter, we looked at how you can better help your company achieve its goals. In this chapter we'll be a bit more selfish and see how you can benefit from your work. This will take two forms. We'll look at how you can be more successful within the the company, but we'll also talk about how you can build a reputation outside of your company and become a key person of influence in your industry.

6.1 Two (or five or ten) year plan

The best way to ensure that your career is heading towards a place where you want it to be, is to know what you're aiming for. In other words, you're far more likely to get somewhere interesting if you know where you're heading as you set out.

Do you know what you want to be doing in two years? Or five or ten years? I realise that might be a tricky question to answer in an industry that changes as quickly as ours does, but that shouldn't prevent you from having some kind of roadmap - even if it's something that you revisit and replan every couple of years.

Most companies will have some kind of annual review process. You could use that as the basis for your plan. In my experience, the annual review will often only concentrate on the next year, but there's nothing to stop you using a similar process to plan further ahead - either as part of your review with the help of a friendly manager or as something that you do by yourself.

The output from an annual review will usually be a list of goals that you want to achieve over the next years. These goals should be SMART, that is they should have the following attributes. They should be:

- **Specific** - defining exactly what needs to be done to meet the goal.
- **Measureable** - defining how you will measure whether you have achieved this goal.
- **Achieveable** - something that you can be expected to achieve in the given time period.
- **Relevant** - something that is useful to you in your job.
- **Time-dependent** - defining the time period in which you will achieve this goal.

Examples of good goals might be something like:

- Lead a successful project that is estimated to take four weeks.
- Take technical responsibility for the some microservice by the end of February.
- Give a talk about the architecture of your microservice by the end of April.

You should also take the time to identify any training courses will be useful to move you to the next level in your role. While it's often tempting to just look at hard technical training courses, don't forget that softer skills can also be really useful.

You can use a similar goal-setting approach to set targets for your longer-term career. The further ahead you are planning, the vaguer your plans can be, but you should still make a note of them somewhere so that you can revisit them (and, hopefully, tighten them up) when their deadlines get closer.

6.2 Career structure

Different companies will, of course, have different career structures. But one common approach is to have a management stream and a technical stream, with employees able to choose which of the two they are most drawn to.

The management stream is the more traditional approach. When you show that you are good at a job, you get promoted to a position where you manage other people who are doing the same job. As a programmer, you would become a team leader and then, eventually, a development manager. Over time, you would spend less of your day doing actual programming work and more or it helping other programmers with their work. You would get more involved with the internal politics of your company but, in return, you would get to make a bigger impact on the business by driving larger projects. You would become the line manager for people in your team, running their annual reviews, helping them with career advice and even being involved in setting their salary levels.

While the management stream has you managing people, there's another, more technical route you can take where you're managing systems rather people. Instead of taking responsibility for a team of developers, you might look after a specific area of the system. In our current world of microservices you might take technical ownership of one or two microservices. You become the person who knows the technical details of those systems and who decides how new features are going to be implemented. In time you might become a technical architect for large parts of the system. Like a team leader, you will eventually find yourself spending less time writing code and more time in architectural meetings, sketching possible system designs on whiteboards. In exchange, you will drive the technical direction of the parts of the system you are responsible for and this is a great way to have a large impact on the way the company's systems work.

Don't get too hung up on job titles. They don't mean much in the IT industry. No-one has ever put much effort into calibrating them. Each company has its own version. How senior is a senior programmer? How junior is a junior programmer? Does becoming a senior programmer mean that you've contributed something at a senior level? Or does it just mean that you've been a the company for a certain amount of time?

I've often seen job titles used as an incentive. But unless they come with more interesting work (and, hopefully, an increase in salary) they mean nothing. I was once being interviewed for a job and was asked where I saw myself in a few years time. I talked about wanting to be a software architect. When I was offered the job, it was with the job title software architect. But, in reality, it was just a standard senior developer role with responsibilities (and a salary) commensurate with that role.

It's also worth noting that (particularly in start-ups) development groups often have a rather flat structure. I've lost count of the number of times I've worked in a team where my manager is only one level away from the CTO. This means that there's not much space to promote people and in an environment like that people often place a lot more importance in job titles.

6.3 Internal self-marketing

If you want to be successful within your company then it helps if the management in the company knows who you are. This might be described as "internal self-marketing". Some people have a natural ability for becoming known in a group without really thinking about it, but (at the risk of stereotyping slightly) many people who make good developers don't have that skill and have to work at it.

It starts with being sociable. When you're in the kitchen making your coffee, look around and make eye contact with people. Talk to them. Ask them what they do in the company. Maybe your kitchen is only used by the developers, in which case it's worth making occasional sorties to other kitchens in the office (pretend that your kitchen has run out of milk or something like that).

The next step is to go to lunch with people. If your team doesn't go to lunch together regularly, then see if you can instigate it. Or, when a project, launches successfully suggest a lunch for all the people involved (you might even get a budget for it). The advantage of a project lunch is that it, hopefully, won't just involve the technical

team, you'll also get to meet some of the stakeholders and business users of whatever you've been working on.

But the best socialising happens after work. Either in a restaurant or (more likely) in the local pub. That's when you can really get to know people. Even better if it's in a pub following a quarterly all-hands company meeting. Everyone will be there and there's probably a few quid behind the bar. If the pub isn't your usual hangout or you're not much of a drinker, it's well worth making the effort to go along for a while and get to know people.

It's the cross-team conversations that happen in situations like that which really make it worth your time. I've had situation where I've been talking to someone who uses some back-office system that I maintain and they've mentioned a little problem they've had with the system. I've realised that it would only take a tiny tweak to the code to remove that niggle and the following day I've raised a ticket to fix the problem and it gets done in the next development sprint. That's one more person amongst our users who thinks I'm a person who gets things done. Which is never a bad thing. There's a chance that the user would never have thought of mentioning that problem through the usual ticketing channels, but after a couple of pints he mentioned it and it was fixed soon afterwards.

I mentioned quarterly all-hands meetings. Most companies have something similar. Everyone gets together (often in some off-site venue) and the executive team update them on how the company is going, or announce another massive reorganisation. That's another opportunity to become known. These meeting always have Q&A sessions and if you can make a habit of asking intelligent and interesting questions, then you'll soon start to be recognised by other people. Sometimes there will be opportunities to speak at these meetings and you should always be prepared to talk about your projects when given the opportunity.

As a measure of how well my internal marketing is working I like to ask myself how many of the executive team know who I am. It would be unsurprising if the CTO didn't know my name (assuming it's not

too large a company) but if the CEO also knows me, I think I'm doing well.

6.4 External self-marketing

Internal self-marketing is all very well, but in order to really help your career, you should also be thinking about external self-marketing. Internal self-marketing might get you a promotion with a decent salary rise, but it's external self-marketing that will get you head-hunted for a new job at Google, Facebook or whoever you would love to work for.

We will discuss four different types of external self-marketing. In order of increasing difficulty, we have:

- Answering questions about technologies you are expert in on web sites or mailing lists.
- Contributing to an open source project using your skills.
- Blogging about your work or the technologies that you use.
- Speaking about your work of the technologies that you use.

As is often the way, it's the items at the harder end of the list that will have the greatest effect.

6.4.1 Answering questions

There will always be people who know less about your areas of expertise than you do. And those people will always be asking questions about those topics on web sites like https://stackoverflow.com/ or various mailing lists. By answering these questions you can increase your reputation as an expert in those technologies.

For example, I'm like to answer StackOverflow questions about Perl and other technologies that I use. Because I've been doing that for several years, I'm now one of the highest-rated Perl experts on the site. This has had an effect on my career. People have come up to

me at conferences and told me that they found a particular answer useful and I've been told that I've been given job interviews because the people doing the screening recognised my name from the site.

There are, however, plenty of other people who are answering the same questions. So how are you going to differentiate your answers from the rest of them. I have a few ideas.

1. **Be polite** Often the people asking these questions are new to the technologies or even new to programming. And experienced developers can sometimes be a little rude in the way they ask them to explain exactly what they are trying to do. Being a little gentler and trying to understand the original question without immediately calling the questioner an idiot can make your answer stand out from the crowd.

2. **Be correct** I've seen cases where people are so determined to be the first person to answer that they just post the first solution that comes to mind. Sometime, these solutions don't even compile. It can pay to stand back a bit and take the time to post something accurate and easy to understand.

3. **Be instructive** The person asking the question is usually up against some kind of deadline. They will just want to get their answer as quickly as possible. However, I still think it's worth taking the time to explain your answer. Some of these sites have pretty good Googlejuice and it's useful to have detailed answers to questions come up with your name attached when other people are searching for solutions to similar problems in the future.

6.4.2 Open source work

If you have certain programming skills, then a good way to advertise them is to contribute to an open source project that requires those skills. This has a few useful effects. You'll be getting better-known amongst the rest of the project team. You'll be working with other

people who know your technologies of interest and that will, hopefully, improve your skills. And you'll be contributing to something that some people will find useful.

6.4.3 Writing about your work

It's likely that you have useful or interesting things to say about the technologies that you use regularly, so why not write those down and share them with the rest of the world.

The most obvious approach to take is to write a blog. You might not even have to set up your own blog. More and more companies are setting up a technical blog to share information about the technologies they are using. Mostly, I think, they are using it as a recruitment tool - "Hey, look at all the cool stuff we're doing; wanna come and join us?" - but we can use this to our advantage by writing articles with our name attached and publishing them on the company blog. You'll need to find out who is responsible for organising the blog and ask them how you go about proposing and writing an article. There will probably (hopefully!) be some kind of editing process that your article needs to go through before being published. The company will only want high-quality posts on their public-facing blog site and there will also be some level of negotiation about what you can write about. Some of your work will be too commercially sensitive to be made public. If the company doesn't already have a technical blog, then you just found yourself a project that will also help with your internal self-marketing. And a shared blog like this will put less pressure on you to keep producing content than starting your own blog.

If you do want to start your own blog, then there are many options open to you. The *de facto* standard blog platform is WordPress. It's available in two versions. There's a hosted version at wordpress.com or you can install the software from wordpress.org and install it on your own server (assuming you have your own server). The self-hosted version will give you more flexibility in how you run your blog, but you will also be responsible for installing regular security updates

(as the most popular software for building web sites, WordPress is also where many unscrupulous people concentrate their hacking efforts).

But WordPress isn't the only option. Many people like Medium. I've posted a number of programming articles there and I like the site. The only slight downside for a technical blog is that it can be hard to post code in a readable format (but I believe you can work round that by using GitHub gists). Medium has a ready-made audience and one advantage is that you can make your work available for access through the site's paywall - which means that you might get paid for your work.

Dev is another hosted web site that is aimed firmly at the development community. Posts to the site are written as Markdown documents and it therefore happily supports including code samples. If your posts are going to largely technical in nature, then this site would be a good choice.

Another, more manual approach is to use GitHub Pages. This is a system for creating web sites hosted in GitHub repositories. It has built-in support for a site builder called Jekyll which makes it simple to convert Markdown documents into good-looking web pages. It takes more work than something like WordPress but people like the level of control that is gives them.

Of course, blogging is not the only option open to you when it comes to writing about your work. There are magazines (both print and online) that could be interesting in publishing articles about your work. Most of them will even pay you for an article. You might well find that building a following for a blog about your areas of expertise will open doors to writing articles.

And then, of course, there are books. Traditionally you would pitch an idea for a book to publisher (or, less often, a publisher would approach you, as an expert, with an idea) and they would work with you to get your work into bookshops. That option is still open to you, but the emergence of ebooks has opened up another route. You can

now write your book completely on your own. upload it to Amazon (or other ebook marketplaces) and keep all of the income for yourself instead of getting a 10-15% royalty as you might get from a traditional publisher. That might sound attractive, but remember that the publishers do actually do useful work for their share of the money. They will supply people to proofread and edit your book. They will design covers and deal with allocating an ISBN. They will then deal with marketing and distribution. With the ebook route, you'll get a larger percentage of the money but you'll probably need to pay professionals to do some the things that a publisher would do for you. This will, obviously, eat into the amount of money you make.

6.4.4 Speaking about your work

I think the best way to be seen as an expert on your favourite technologies is to give talks about your projects.

I realise that many (perhaps most) people feel uncomfortable speaing in public, but it is a skill that can be learned just like any other. I've been giving public talks for about twenty years and if you had seen my first attempts, you would be astonished by that fact.

I spent a lot of 2000 working on my first book (it was a technical book about Perl programming). In the summer of 2000, the London Perl Users Group was running the first European Perl Conference. I was slightly involved in organising the conference, but I had no plans to speak at it. Public speaking was not something I had any interest in at all. I have a slight stutter and in stressful situations it tends to get worse. I could see no reason for putting myself under the stress of giving a talk.

But my publishers had other ideas. They could see that speaking at the conference would be a great opportunity to market the book. They persuaded me to propose a twenty-minute talk summarising some of the main points from the book and, nervously, I agreed.

The day of the talk was stressful. The book hadn't been published yet, but the publishers had printed a booklet of extracts and posted two hundred copies to me. They were supposed to arrive a few days before the conference, but something went wrong and the couriers had attempted to deliver the box to my house while I was at the first day of the conference. Which meant that I had to spend the morning of the second day traipsing across south London to their depot so I could collect a heavy box which I then took on the underground to the conference venue. By the time I got there, I was hot, sweaty, tired and more than a little grumpy.

My talk was in the afternoon, so I had a while to relax. I sat in the hall for the talk before mine - which was science fiction author Charles Stross being effortless informative and entertaining. And then it was my turn.

It was a disaster. I hadn't practised enough. I hadn't really thought about what I wanted to say. And, to be honest, after a year working on the book, I was slightly bored by the material. And if the speaker is bored by a talk, there's no chance that the audience will be interested. I stuttered through my slides and left the stage, vowing that I would never speak in public again.

But I had forgotten something. At the end of the conference, we had organised as session of "lightning talks". This are five minute talks that are aimed at tempting new people to try speaking. Or sometimes you'll get more experienced speakers letting their hair down a bit and trying something a bit different. As a fairly well-known name on the London Perl scene, I had put my name down to give a lightning talk in the hope that it would persuade other people to give it a try.

So I had to go back on stage the very next day. This time, the experience was completely different. It was a far shorter talk, the material was much more interesting (I talked about a completely ridiculous code library that I had written) and people laughed at my feeble jokes.

It was great. I immediately decided that I'd been far too quick to give up on public speaking and that I'd like to give it another try. The London Perl Group ran occasional technical meetings and soon I became a regular speaker at those events. By the time the next year's European Perl Conference came round (this time in Amsterdam) I was an old hand and was very happy to offer them some talks.

As I said before, most people are scared of public speaking. But there's really no need to be. You might not be great the first time you stand up in front of an audience. But it only takes a little effort to get better. And once you experienced the applause you get at the end of well-presented talk, I guarantee you'll want to do it again.

I strongly recommend that you start with something like a lightning talk at your local users' group. Maybe your local users' group doesn't have lightning talks. If that's the case, why not ask if they'll consider reserving space for two or three of them at the next meeting. Or, perhaps, volunteer to organise an evening of lightning talks. The London Perl Group does this about once a year and it's a great way to draw new speakers in. Once you've organised a meeting like that, grab a prime speaking slot for yourself.

And talk about something you're interested in. You don't need much material to fill five minutes. How about a list of five libraries that you think people don't use enough. Or an amusing exploration of some strange code that you've recently found in your codebase (all codebases have code like that somewhere).

Some Public Speaking Secrets

Allow me to let you into some secrets of public speaking.

Most people who do, really aren't very good at it. And they make almost no effort to improve. If you spend any time at technical meet-ups or conferences you'll soon start to realise how bad most speakers are. Or perhaps you've seen people giving terrible presentations at your company. They give no thought to what

information they need to pass on or the most engaging way to do that.

So when I say it's easy to be a good public speaker, what I actually mean is that it's easy to be better than the average public speaker - and that's because the average public speaker is very bad at it. You only need to put a little effort into honing your skills in order to stand out from the pack.

In order to improve you need to watch other speakers. You can watch bad speakers and identify what it is they are doing wrong. And you can watch good speakers and identify what they are doing right. You should also watch videos of yourself (painful as that might be) so you can see what you need to do in order to improve.

It's easy enough to find bad speakers to watch. As I said above, you just need to watch the vast majority of presentations. It's becoming very common for conference talks to be streamed live or published on YouTube - so that will give you a ripe seam of examples to learn from. It's harder to find good examples, but there is one source that will give you more good examples than you will ever need - TED.

The TED web site is treasure trove of fantastic speakers giving engaging talks on fascinating topics. The people who organise TED work hard to pick the best speakers and give them as much support as they can to give the most interesting talks. If you spend any time at all browsing talks on the TED site you'll start to pick out the traits that make these talks so good.

The talks are laser-focussed. The speakers know exactly what they want to tell the audience. They only have eighteen minutes each so they carefully plan how they are going to spend that time. Even so, you'll notice that they probably cover less material than the speakers you are more used to seeing. A TED speaker might make only three or four big points in that time.

The talks are almost all phrased as a story. And it will often be a story that the speaker was personally involved in. Telling a story is a

great way to engage an audience and if you can make it personal then your connection to the story will make the audience enjoy it more. There will usually be humour in the talk, but very few TED speakers tell jokes. It's far more likely that it will be situational humour that arises directly from the story being told.

The slides for TED talks are always incredibly well-designed. You're probably used to slides that are just lists of bullet points - often far too many bullet points on one slide. If your slides are full of words then your audience will spend most of the time reading your slides rather than listening to you. Good slides contain images that illustrate the story. You'll see there are very few words on TED slides. When there are words on a slide, it's likely that there will only be two or three and they'll be in a massive font. If you ever find yourself saying "those of you at the back probably can't read this" then you've tried to squeeze too many words onto your slides.

I'm not saying that everyone will be able to deliver TED-quality presentations. But I am saying that if you think about your presentations in the same way that TED speakers do, then your presentations will improve and you stand a very good chance of being one of the most memorable speakers wherever you are speaking.

Public Speaking Practicalities

Of course, if you want to give talks, then you need to find places that will let you speak. There are a number of approaches you can take.

- Speak at your company. There are a couple of possibilities here. You could offer to give a lunchtime "brown bag" session talking about some new technology you're trying out or some interesting feature you've just added to your system. Or you could give a more formal talk at one of the big company meetings that we mentioned before. Of course, speaking to people in your company doesn't really count as external self-marketing, but you can see it as practice before you start looking for external speaking opportunities.

- Speak at a local meet-up group. If you search on a site like meetup.com, you will find dozens of meetings taking place near you. Many of them will be monthly and all of them will need speakers for their meetings. It's likely that some of them will be covering subjects that you are interested in, so why not see if they have a spot for you to speak in. Find a likely-looking group and join it. Go along to a couple of meetings and see what kind of people attend and what sort of talks they have. If, after two or three meetings, you think you have something useful to share with the group, then approach the organisers and offer them a talk. Some groups might be less happy to have talks from inexperienced speakers, but I'm confident that you'll find some group nearby that are interested to hear what you have to say. And once you've talked at one group, you'll find it easier to have your suggestions accepted at another one.

- A lot of local user groups are now using sites like MeetUp to organise their meetings, but it's still far from all of them. There will be a number of other technical user groups that you can find by Googling for them. You might well find they are hold their meetings at a local university or, perhaps, at the offices of a local company that likes to promote the fact that it uses certain technologies as it then makes them look more attractive to potential employees who are part of the communities surrounding those technologies. Once you've identified a potential user group then follow the same process as for MeetUp groups - go to a few meetings to see what the group is like before proposing a talk to the organisers.

- At the top of the speaking tree, we find conferences. Obviously, speaking at a large global conference is going to be out of the reach of most new speakers, but if you speak at meeting like we've discussed in the previous couple of paragraphs, there will soon come a time when you feel you're ready for the big time. It's worth realising that the term "conference" covers a really wide of events - from one day workshops that are put on by local user groups to huge corporate conferences that are run by

massive companies and can cost a small fortune to attend. As with user groups, you should start by searching for conferences in your areas of expertise. They will all have web sites showing you where and when the next event is taking place. And in most cases they will give you the timetable for the Call for Participation (CFP) which is how most conferences ask potential speakers to submit ideas for talks.