DevOps from Scratch (Technical Support)

Mark Bools

November 14, 2020

Document ID: A000000001
Last Modified: 2023-10-23

Contents

Contents **iii**

1 How to. . . **1**
- 1.1 . . . read this book 1
- 1.2 . . . get the most from this book 2
- 1.3 . . . manage your workspace 2

2 Setting Up Your Environment **7**
- 2.1 VirtualBox 7
- 2.2 Vagrant 7
- 2.3 git 8
- 2.4 Installing the host tools 8

3 Our Starting Point **9**
- 3.1 Ideation 9

4 DevOps from 20,000 feet **13**
- 4.1 The DevOps Infinite Cycle 14

5 Core Concepts **17**
- 5.1 Cohesion 17
- 5.2 Coupling 17
- 5.3 Abstraction 17
- 5.4 Separation of Concerns 17
- 5.5 Scope 17
- 5.6 Context 17
- 5.7 Contingency 18
- 5.8 Entropy 18
- 5.9 Parsimony 18

6 Virtualisation **19**
- 6.1 Creating a Virtual Server with `VBoxManage` 19
- 6.2 Setting up a simple virtual machine 19
- 6.3 And now the easy way 22

7 Infrastructure as Code **25**
- 7.1 Less Talk, More Do! 26
- 7.2 What about the data? 30

8 The Master Server **31**
8.1 Preliminaries 31
8.2 Base server and operating system 31
8.3 Vagrant SSH 33
8.4 What versus How 36
8.5 Our core configuration tool 38
8.6 Something is missing? 44

9 Requirements **45**
9.1 What are requirements for? 46
9.2 Uses of requirements 48
9.3 How to capture our requirements 49
9.4 Starting a conversation 50
9.5 Testable requirements 50

10 Master Server Requirements—round one **53**

11 Testing I **55**
11.1 The Purpose of Testing 55
11.2 Testing Principles 55
11.3 Requirements as tests 57

12 Security I **59**
12.1 Risk 59
12.2 Architecture 59

13 Architecture I **61**

14 Firewall **63**
14.1 What is a firewall? 63
14.2 What does a firewall do? 64

15 Repositories **65**

16 Managing Data **67**

Bibliography **69**

A Brief History of "devops" **71**

Index **73**

Chapter 1

How to. . .

This chapter offers some guidance on getting the most from this book.

1.1 . . . read this book

If I was being flippant I might say, "with your eyes" but I'm bigger than that so instead I will suggest some ways you might use this material.

This book is organised as a single narrative course centred around the technology that supports DevOps. If you start on page one, read through each page and follow along with all the material, you should end up being proficient with the entire IT System Lifecycle Management.

This book contains mistakes. Deliberate mistakes (and no doubt some mistakes that I did not intend, that's life). Sometimes we will do things more than once. WTF? In most educational material we are presented with 'perfect' solutions, this is not realistic. The real world sucks. It changes constantly and today's ideal solution is okay but tomorrow the boss (or customer) decides new technology is desirable how do we handle this? This is were some soft skills may be required to persuade them to change their mind. Assuming we cannot persuade them to change we need to plan and execute a migration from the older solution to the new solution. Rather than avoid this sort of complexity this course takes it head on.

This course is focussed on concepts rather than specific technologies. Yes, we use specific tools but the aim is to understand the approach to DevOps, learning how to confront and solve problems rather than simply 'press this button', 'put this value here' type tutorials (again, yes, there is some of this sort of material, especially early on, but the focus always remains on the concepts involved not merely how to finish specific tasks). Following tutorials is fine but generally we only learn one thing, how to do the one task in the tutorial. This is of limited use. Learning the core idea behind the tutorial means you can apply what is learned in novel circumstances.

For example, we use VirtualBox as our virtual machine hypervisor. What if your organisation uses VMWare or Windows Hyper-V? No problem if you understand the core concepts of virtualisation it should be a simple matter of translating your knowledge to a new set of operations (figuring out which menu items or commands get you the result you want). If you understand the core concepts then you can figure out the button pressing parts by reading the

manual for the tool. Its like understanding variables in programming, once you know what a variable is then figuring out variables in programming language X is largely a matter of syntax.

Don't have time to follow along from the start? Perhaps you already know enough about networks and feel confident skipping that material. No problem. At the start of each section you will find details of how to create an appropriate environment for that section (see §1.3). These 'checkpoints' also mean that if you mess up you can simply throw your environment away, recreate it using the closest checkpoint and continue with the course. In fact I encourage you to mess up your environment. You will learn much more by 'playing'. So set up and environment, mess around with your own ideas and then, when you are ready to do the next part of the course, either restore your own saved snapshots or tear down the environment and build a new one using the provided checkpoint code.

If you are more interested in the philosophical and organisational aspect of DevOps then take a look at this book's companion *Devops from Scratch (Organisation)*[Boo20a]

Having difficulty? No problem. Ask for help. Someone in the community may help and since I am in the community I can also help clarify things. If enough people are confused by the material then obviously I messed up and need to rewrite that material to be more clear; it shall be done.

Other books are available with more detailed material. Trying to cover all of the many complex topics under the IT System Lifecycle Management rubric would make this book not only much larger but also less focussed. My solution is to write books to deep-dive into related topics and reference those books from this one where appropriate. This way I hope you will find all the guidance you need either here or in one of the supporting books.

All of these books undergo constant maintenance (hopefully improvement), my only goal is to make the material more clear and more accessible over time.

1.2 ... get the most from this book

Firstly, forget DevOps. Seriously, ignore it. Although this book uses the term DevOps (mostly for marketing reasons) it is really more general, it is about how to do IT System Lifecycle Management more effectively. DevOps is a distraction at worst and only a small part of successful IT System Lifecycle Management at best.

Do the examples! You'll get much more from the material by following along and investigating for yourself.

1.3 ... manage your workspace

We are going to be doing a lot of practical work throughout this book so it is worthwhile considering how we will manage our workspace.

A lot of this section will only make sense once you have read about the tools Git, VirtualBox, and Vagrant but I'm assembling basic advice here to make it easier to refer back to later.

1.3.1 Initial setup of your workspace

If you follow this book from page one to the end you should find your workspace is always in sync with whatever the book is dealing with but practically many of you will jump to sections of particular interest, skipping many sections. Anticipating this I've put in plenty of checkpoints. All of the material (including checkpoints) is held in a single Git repository, so I recommend getting that first.

Assuming you have installed Git (see §2.4) you can create your project workspace.

```bash
mkdir dfs
cd dfs
git clone --depth 1 https://gitlab.com/saltyvagrant.classes/dfs-material.git course-material
mkdir classroom
mkdir archive
```

Your `dfs` workspace now contains three directories; `course-material` holds all this book's accompanying material, `classroom` is where you will follow along with the course, and `archive` is where we will store various backup files.

Throughout this book I use the `WSR` directory[1] to refer to the root of your workspace. Any path that does not explicitly start from the `WSR` root assumes you are following instructions from the last checkpoint and are relative to whichever directory you should be in at the time.

```bash
cd WSR
cd WSR/classroom
cd ../course-material
cd WSR
cd classroom
```

Lines 1, 2, and 4 each start with `WSR` and are therefore not really relative to your current working directory. You should take these to be absolute directories rooted at your workspace root `WSR`. If your workspace is at `/home/fred/xyz` then `WSR/classroom` should be read as `/home/fred/xyz/classroom`.

Line 3 is relative to your current working directory (in this example `WSR/classroom`) and is referring to `WSR/course-material` (since the parent of `WSR/classroom` it `WSR` and `course-material` is to be found directly under this directory).

Line 5 is again relative to your current working directory. As you just moved to `WSR` (line 4) this refers to your `classroom` directory under that root.

[1] If you are using Microsoft Windows as your host you will need to convert '/' to '\' in paths whenever working in the host workspace. (It is precisely because of this sort of "conversion" nonsense that we use the guest workspace most of the time.)

1.3.2 Regular host workspace activities

There are a number of actions you may want to repeat throughout this course. Rather than repeat them in full each time I present them here and simply refer to these entries as necessary.

1.3.2.1 Checkpoint Classroom

You will need to checkpoint the classroom at least once, when you start the course. If you get lost in the material you can reset your classroom to one of the checkpoints in the book. This will clean up you `classroom` directory ensuring you are ready to proceed with the book's follow-along lessons.

1. Shutdown any running classroom Virtual Machine (VM)[2] (if one is currently set up).

   ```bash
   cd WSR/classroom
   vagrant halt
   ```

2. Backup your current classroom

   ```bash
   cd WSR
   mv classroom archive/classroom_<date>
   ```

   Replace `<date>` with the date of the backup (I recommend using a `YYYMMDD` format as this sorts properly). For example, if today where December 3rd 2020 and I wanted to backup my classroom I would us the following[3].

   ```bash
   cd WSR
   mv classroom archive/classroom_20201203
   ```

3. Copy the relevant material from `course-material`

   ```bash
   cd WSR
   cp course-material/<checkpoint>/ classroom
   ```

   Replacing `<checkpoint>` with the name if the checkpoint from which you want to proceed.

4. Start up the classroom

[2] A segmented presentation or emulation of a physical computer allowing multiple 'guest' machines to share the physical resources of the 'host' computer.

[3] Windows users should use `move` rather than `mv`

```bash
cd WSR/classroom
vagrant up
```

This will start up any VM required for the classes.

1.3.2.2 Snapshot Classroom

If you are following the advice given above and ‘playing’ with your classrooms then I suggest your take a snapshot of your environment just before you start to play. This way you can quickly reset your classroom back you a point where it is ready for you to continue following along with this course.

1. Follow along with this course.
2. Decide to ‘play’ for a while so take a snapshot.

   ```bash
   cd WSR/classroom
   vagrant snapshot save class_<date>
   ```

   Replace `<date>` with the date of the snapshot (I recommend using a `YYMMDD` format as this sorts properly). For example, if today where December 3rd 2020 and I wanted to backup my classroom I would use the following.

   ```bash
   cd WSR/classroom
   vagrant snapshot save class_201203
   ```

3. Play with your classroom environment.
4. Decide to resume the course as described in this book.
5. Restore your classroom to the saved snapshot.

   ```bash
   cd WSR/classroom
   vagrant snapshot restore class_<date>
   ```

   Where `<date>` is the date of the snapshot to be restored. For example, to restore the snapshot from December 3rd 2020 we created earlier I would use the following.

   ```bash
   cd WSR/classroom
   vagrant snapshot restore class_201203
   ```

6. Resume course.

One other useful snapshot command is `list`, this can be used to show your previously saved snapshots (useful if, like me, your forget this sort of thing).

```bash
cd WSR/classroom
vagrant snapshot list
```

1.3.2.3 Update material

This book, and consequently the accompanying material, is continually being updated[4]. Most updates will be to the *guest* workspace consequently the host workspace will rarely need updating, the following procedure will update your host workspace and bring your guest systems up to date.

```bash
cd WSR/course-material
git pull
```

This will update the course material in the host workspace. If you have created a `classroom` then you may need to re-copy the relevant course material and re-provision the virtual machine.

```bash
cd WSR
cp -rf course-material/<checkpoint>/* classroom
cd classroom
vagrant up --provision
```

Line 2 copies the relevant checkpoint files (obviously replacing `<checkpoint>` with the actual checkpoint directory you want to use). Line 4 will update any guest virtual machines (even if they already exist or are running).

[4]If you have downloaded the PDF version of this book then you should download the latest version at the same time you update the course material, otherwise they will get out of sync.

Chapter 2

Setting Up Your Environment

In order that we are all seeing the same environment as we progress through the following material you will need to install three applications onto your computer (the host computer):

- `VirtualBox`
- `vagrant`
- `git`

Let's take a look at each and discuss why they are required.

2.1 VirtualBox

VirtualBox is Oracle's virtual machine application. This allows us to run a virtual (guest) machine on our host computer. This in turn means that even if you are running, for example, a Windows PC you will be able to run the Linux servers required to follow along with this material.

Virtualisation also isolates our host computer from the machines that we use. This has the advantage that no matter how badly we mess up the virtual environment it will have no effect on our host computer and any change to our host computer will have no effect on the virtual machines[1].

2.2 Vagrant

Vagrant is HashiCorp's command line tool for managing virtual machines. Vagrant provided a simple consistent method for defining virtual machines as code. This means we can all easily set up the same virtual machine environment without the need to rely on following complex set up instructions.

As with many topics covered in this course, there is a more detailed book covering Vagrant *Vagrant from Scratch*[Boo20e].

[1] This is not 100% true, but close enough for our purposes here.

2.3 git

Git has become the *de facto* standard in version control tools. Git is a powerful tool, unfortunately its history means it has a bloated command line interface that is often daunting and confusing to newcomers. Fear not! We will initially use `git` commands to obtain some files and nothing more (so you can just type the commands with no need to understand them) but as we progress we will explain the `git` command line and if you are interested in learning Git in detail there is a complete book on the topic *Git from Scratch*[Boo20b].

2.4 Installing the host tools

I have prepared some brief installation videos but to get the most up-to-date instructions for installing these host tools follow the instructions on their web sites.

Chapter 3

Our Starting Point

We installed the host tools for our technical 'classroom' in Chapter 2, now we consider the environment in which our organisation works in a broader, softer sense.

To give ourselves something concrete to work on we are going to develop 'The Devops Guild', a complex of services (some using existing systems that we will integrate and customise, some we will develop from scratch) that will form a website supporting DevOps professionals.

Like all initial ideas we're not entirely sure how to create this service but we know that we will need to be able to create and tear-down experimental systems. We're also pretty sure we want to use cloud services to host our system because this allows us to scale efficiently.

We are keen open source advocates, so not only will we use open source solutions where possible we will also contribute back to the open source community. That said, we're a pragmatist first, so we won't rabidly pursue an open source solution if a cost effective non-open source solution presents itself.

To keep initial costs down further we want to use our existing computers where possible.

Initially we will be working from a home office, but recognise that we will need more computing power than we own. Access to 24×7 up-time and good network availability will be important as we progress. We currently have a laptop and an internet connection, and that's pretty much it.

3.1 Ideation

The first step in any project is the formation of the idea.

When I first get an idea for a project it is either specific (these tend to be smaller projects that are easily conceived as a single highly focussed idea) or nebulous (these tend to be more aspirational, general, and poorly conceived at the start). This project is the latter type.

The process of fleshing out these more general project ideas tends to be cyclic.

1. Have idea

2. Use existing knowledge to formulate plan

3. Experiment
4. Evaluate
5. Refine
6. Discover new tools and techniques
7. Repeat steps 1–6

It is seldom as linear as this and rarely a smooth process, particularly for larger projects where we are dealing with many people (and often many teams) all with different perspectives on the project. In my professional life I have rarely worked on a large project that has been delivered as it was originally conceived.

3.1.1 Intitial Ideas for 'The DevOps Guild'

The guild will be run as a cooperative, administered and developed by the membership.

Here are some initial ideas for site features.

- Static site. There will be a framework of static site content binding the various components together.
- Wiki. Registered members can edit. Possibly more than one (publicly readable and internal guild business)
- Resume. Members can maintain there resume on site. This resume can be made public and site provides 'pretty printing'.
  - Resume syncing? Possibly offering updating of resume on other sites (assuming API availability).
- Portfolio. Can be made public or privately linked to support resume, job applications, or when seeking Guild progression.
  - Linking to certifications held.
  - Linking to members GitHub etc.
  - Guild repository.
  - Guild certification.
- Forums
- Micro blogging
- Chat
- Training. Members can develop training and deliver through Guild site. Training can be free or monetised.
- Mentoring. Members provide mentoring and can earn Gilt (on site tokens) and Reputation (on site measure of members participation).
- Membership. Anyone can register as a member.

- Fees. Members will be required to pay an annual fee to access site features (Minimum fee to be set by membership). These fees will support provisioning, development, and administration costs allowing the site (and Guild) to remain ad and sponsor independent.
  - Pay-what-you-want. Fees will be at the discretion of member (subject to minimum fee).
  - Fee incentives. Top payment in a month gets lifetime membership (tie breaker; earliest wins). Next five highest get two years membership instead of one.
  - Member can choose whether to make their fee visible to public, members only, or keep it private. (Fee will always be private during month it is paid to avoid compromising incentives.)
  - Members can choose to display their 'fee rank' (position in the month they subscribed, overall position).
- Gilt. On site tokens. Can be earned and exchanged with other members. Can be purchased. Can be used to vote on new features etc. Can be used to pay for training (when provider allows) of membership fees.
- Guild 'ranking'. Members can gain ranking (promotion by peer review).
  - Apprentice. Any member can become an Apprentice.
  - Journeyman. An Apprentice meeting Guild's minimum standards can apply to have their portfolio peer reviewed, if accepted they progress to Journeyman.
  - Craftsman. A Journeyman meeting Guild's standards can apply to have their portfolio peer reviewed, if accepted they progress to Craftsman.
  - Master Craftsman. A Craftsman meeting Guild's standards can apply to have their portfolio peer reviewed, if accepted they progress to Master Craftsman.
  - All standards are set by Guild members.
  - All peer reviews are performed by randomly assigned reviewers selected from higher ranks. As far as practicable all participants (applicant and reviewers) are anonymous.
- Request For Comment (RFC). These will be used to solicit input from members on all aspects of the site development and Guild operation.

Author Note

Turn general notes into something more constructive.

Chapter 4

DevOps from 20,000 feet

This is the obligatory 'what is DevOps' chapter. The problem is DevOps has been so abused as a term that it is largely useless but here goes.

The core idea of DevOps pre-dates the term DevOps, indeed at the start of computing there was no significant distinction between user, developers, or operators as users were also the developers and the operators. As computers matured from academic and military environments into more commercial settings users split off into a clearly distinct group[1]. The people developing systems also started to separate from the people operating the systems, particularly in the mainframe days where these vast machines required the constant attention of operators who loaded punched cards and paper tape, and removed printouts for delivery to developers or users. The advent of Teletype terminals and subsequently video display terminals made it possible for user to interact with systems blissfully ignorant of the operators behind the scenes. Similarly developers became increasingly independent of the operators, able to write, compile, and run their creations independent of the operators keeping things running behind the scenes.

Once developers had completed their work they would hand over the finished product to operators who would then take over the loading and operation of the system on behalf of users. This generally worked well in the early days simply because both developers and operators worked within the same organisation. The increased commodification of software though increased the distance between developer and operator. Indeed operators would often take a software product and install it knowing only that it was an IBM product, never knowing or interacting with the developer. Because this gap developed between commercial suppliers of software and those who operated the systems delivering value to customers so too there tended to be an increasing gap between developers and operators even within a single organisation; why have one rule for operators when working with an external supplier and another when working with internal developers?

This divorce between developers and operators made a certain sense when delivering packages software, that is software that was a simple stand-alone system. Developers had little concern about working well with others. The operating system ensured the various programs running on the computer were separated from one another so operators had few concerns. Products tended to

[1] Yes, I am massively over simplifying things here, but I think the point stands.

go through long life-cycles and this allowed for extensive testing (and testing was somewhat simpler as the software operated largely on its own).

This situation did not last long though. Increased complexity meant increased specialisation and consequently increased interaction between components of a system. No longer would a development team write a system largely from scratch but they would take various commercial products and build them into a system. Each building block would be treated as a black-box and development teams where hostage to the delivery cycle of the vendor to fix problems (resulting in code being developed by these teams to 'work around' perceived deficiencies in the commercial packages).

Long story short, the increased complexity of systems also increased the complexity of delivering systems from development teams into operation. No longer a simple 'here it is' the delivery became a complex set of operations the install various off-the-shelf systems, configure them, then install the custom components, test they worked, etc. All this complexity resulted in either longer delivery times (and wicked documents describing—hopefully—how the installation should be performed), or more commonly development projects would build out the first operational system and deliver the entire system to the operations team (saving the complexity of delivering build instructions to operators). This second approach often became the norm.

Maintaining development teams after the initial system was delivered starts to become expensive as more and more systems are delivered. Consequently it is common for operations teams to take over maintenance of systems. Problems arise when knowledge held by the developers evaporates with the development team, seldom being effectively communicated to the operations team. This leads to poor understanding of the system and consequently a struggle to diagnose and correct problems.

This problem is exacerbated by the advent of systems supporting web based products and services. The rapid development and delivery of these systems means constant rapid change.

Enter DevOps, a call to return to a culture in which developer and operators work closely together in both developing and maintaining systems. DevOps philosophy complements the Agile movement which encourages close work between users and developers.

DevOps is about creating as frictionless a cycle as possible for the development, validation, delivery, and monitoring of systems.

4.1 The DevOps Infinite Cycle

No doubt you've come across the DevOps steps illustrated as a sort of infinity symbols. This suggests that DevOps is an endless cycles of these steps.

- Plan
- Code
- Build
- Test
- Release

- Deploy
- Operate
- Monitor

As an IT professional you will find yourself in this cycle and more importantly you will often be problem solving in the middle of this cycle. It is a luxury to be neatly involved from the planning stage onward and even under these ideal circumstances you will soon find yourself troubleshooting throughout the process. This is the attraction of being a generalist!

Chapter 5

Core Concepts

There are several concepts that crop up across the design and management of IT that are so universally applicable that it is worth learning them regardless of your specific interests.

This chapter introduces these core concepts.

5.1 Cohesion

Cohesion refers to how closely elements are related to one another. In generally it is desirable to keep related things together and unrelated things separate.

5.2 Coupling

Coupling refers to how tightly to elements depend upon one another. In general we want elements to be as independent as practicable.

5.3 Abstraction

Abstraction is the process of extracting the essential from the incidental.

5.4 Separation of Concerns

Separation of concerns is a general principal that employs abstraction to increased cohesion and reduce coupling. The general idea is for elements to 'mind their own business', performing a well bounded function (or set of factions).

5.5 Scope

Scope is the 'range of applicability' of an element.

5.6 Context

Everything operates in a context. Most of the time in IT the context is well defined.

5.7 Contingency

Probably best summarised as 'it depends', contingency is the idea that we often must account for things changing.

5.8 Entropy

'Things degrade over time', or perhaps more accurately 'without concerted effort to prevent it, thing get worse over time'. In software circles this is colloquially known as 'bit rot', the idea that without specific work to avoid it a software system's structure will, over time, become more complex, more difficult to maintain, and more prone to error.

5.9 Parsimony

The 'KISS' (Keep It Simple, Stupid) principle. Do not make things more complex than required. The more parts a system has the more opportunity the more things there are to go wrong, so it makes sense to use as few things as possible.

Chapter 6

Virtualisation

There was a time when virtualisation was considered a rather exotic solution for IT systems but nowadays it is considered essential.

In Chapter 2 we installed VirtualBox Oracle's virtualisation tool. We will be using VirtualBox for all our virtualisation. We also installed Vagrant a handy tool from HashiCorp that simplifies the programmatic definition of our local virtual machines. In this chapter we look at creating a virtual server both manually and using Vagrant.

6.1 Creating a Virtual Server with `VBoxManage`

`VBoxManage` is the Command Line Interface (CLI)[1] to VirtualBox. I am using the CLI rather than the more common Graphical User Interface (GUI)[2,3] for two reasons:

1. Education; using the CLI discloses more about how a VM is put together in VirtualBox, this leads to a better understanding of virtualisation.

2. System as Code; a central tenet of the technical aspects of DevOps is the system as code. Graphical interfaces are not easy to handle programmatically whereas a CLI is ideal for scripting.

Readers interested in more detail of virtualisation and VirtualBox specifically may find *VirtualBox from Scratch*[Boo20f] useful.

6.2 Setting up a simple virtual machine

Let's create our first virtual machine.

[1] Application interface using the computers command shell.
[2] User interface using a visual desktop metaphor.
[3] If you are interested in using the VirtualBox GUI there are plenty of examples online.

Host operating system

The following commands should work on MacOS, Linux, or Windows 10[a]

[a] Assuming you have a recently up-to-date Windows 10 installation.

First create a directory within which we will create our virtual machine and make this our current working directory.

```bash
mkdir vbclass
cd vbclass
```

Now open a command line terminal and download the installation media we need to build our server. Since we are building a Debian server we download a Debian installation ISO.

```bash
curl -O https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/\
   debian-10.9.0-amd64-netinst.iso -L
```

`curl` is generally available as a default command line tool nowadays but you may need to install it on your host (or just download the ISO via your browser).

Now we run a series of `VBoxManage` commands to setup the virtual hardware for out virtual machine.

```bash
VBoxManage createvm --name "vbdemo" --register --basefolder "$(pwd)"
VBoxManage createhd --filename vbdemo/vbdemo.vdi --size 20000
VBoxManage storagectl "vbdemo" --name "SATA Controller" --add sata
VBoxManage storageattach "vbdemo" --storagectl "SATA Controller" --port 0 --device 0 --type hdd --medium vbdemo/vbdemo.vdi
VBoxManage storageattach "vbdemo" --storagectl "SATA Controller" --port 1 --device 0 --type dvddrive --medium ./debian-10.9.0-amd64-netinst.iso
VBoxManage modifyvm "vbdemo" --memory 8192
VBoxManage modifyvm "vbdemo" --nic1 bridged --bridgeadapter1 en1
VBoxManage modifyvm "vbdemo" --ostype Debian
```

We start by creating the base VM settings file and registering the VM with the VirtualBox library (line 1). The settings file is an XML file called `vbdemo.vbox` in a `vbdemo` subdirectory for the current working directory. You

will also notice that looking in the VirtualBox library that the new VM is registered.

Once this base VM settings file is available we can start adding custom hardware to our virtual machine.

Line 2 creates a virtual disk 20GB in size in the same directory as the settings file. This new disk is empty when created.

Line 3 creates a SATA storage controller in our new VM and then lines 4 and 5 associate disks with this controller. Line 4 associates the virtual disk we just created in line 2 and line 5 associates the Debian installation ISO downloaded previously. Line 5 is the equivalent of loading a Debian installation CD into a CD drive attached to the virtual machine (note the `--type dvddrive` option on line 5).

Line 6 gives our memory a boost to 8GB, you may need to modify this to suit your host machine (it's generally a bad idea to assign more virtual memory to a virtual machine than you have physical memory on your host machine).

Line 7 configures a network interface card (nic) on the virtual machine. In this set up we are simply creating a bridged network interface, which means the virtual machine will we network device `en1` as if it were attached to your local network.

Line 8 tells VirtualBox to expect a Debian operating system on this virtual machine.

After running these commands we have the following situation.

- A directory containing a virtual machine setting file (`vbdemo/vbdemo.vbox`) and a virtual disk (`vbdemo/vbdemo.vdi`).
- The settings file has been modified so that the virtual machine:
  - has 8 GB RAM
  - a SATA controler with two drives attached:
    * the virtual 20GB hard drive (`vbdemo/vbdemo.vdi`)
    * a DVD drive containing the Debian installer (`\nolinkurl{./debian-10.6.0-amd64-netinst.iso}`
  - a network interface card (`en1`) 'wired' to your host computer's network

When we start this virtual machine the hardware will present as specified in the settings file and, as the hard drive has nothing installed on it, the system will boot from the virtual DVD drive, beginning the Debian installer.

Start the virtual machine.

```bash
VBoxHeadless --startvm "vbdemo" &
```

This will start the virtual machine 'headless', without a virtual display attached. This may seem odd, but in future we will access our virtual machines through SSH so none of the machines will be configured to use a display other than its console. This is the situation you are most likely to encounter professionally.

Every machine has a console. The console is a screen and keyboard directly attached to the machine. In a data centre these screens and keyboards are

typically physical devices installed into the racks holding the servers and they are used by data centre personnel to monitor and control servers in the rack. Data centres are also commonly configured to allow remote access to consoles in addition to the physical console devices, saving personnel from needing to enter the physical data centre to access the console. In virtual environments (such as ours) the console is accessed virtually.

- Using the VirtualBox library GUI.
- Using the Remote Desktop Protocol (RDP)[4] which VirtualBox makes available on port 3389.

Once on the virtual machine's console you will see the Debian installer and you can walk through the installation steps as you would for a physical machine.

At the end of the installation you will be prompted to remove the installation CD and reboot. The Debian installation ISO mounted in the virtual DVD drive will be automatically 'ejected' as the virtual machine reboots, so just select to reboot the virtual machine and you will see for virtual machine restart into a Debian console prompting you for the username and password to login.

6.3 And now the easy way

We just set up a Debian server manually using the VirtualBox CLI. This is useful to learn the nuts and bolts behind setting up a virtual machine and using a CLI means we could put these commands into a script file and run them whenever we wanted to create a new machine.

We have not discussed how to automate the installation of Debian itself, this is a topic dealt with in *Packer from Scratch*[Boo20c], but we don't need to consider this now as we're going to turn our attention to the use of Vagrant.

6.3.1 Introduction to Vagrant

Vagrant is a command line tool for managing virtual machines[5].

Hang on? Isn't that what VirtualBox does?

No. VirtualBox provides the facility to *run* virtual machines, Vagrant *manages* virtual machines.

Vagrant is capable of managing virtual machines running on several different virtualisation tools; VirtualBox, VMWare, and Hyper-V being the main ones used to run virtual machines on a local host. In Vagrant terminology these virtualisation tools are 'providers'. Vagrant has a plugin architecture, so developers are free to create their own provider if the existing ones are unsuitable.

Another key concept to Vagrant is the 'box'. A box is a packaged up base virtual machine that is used by Vagrant to provide one or more virtual machines to your project.

[4] A network display protocol developed by Microsoft.

[5] We are using Vagrant as a tool in this book but I have written another book dealing with Vargant in more detail, *Vagrant from Scratch*[Boo20e]

Let's start defining a Vagrant system. This system will consist of a single virtual machine. We base our virtual machine on a box supplied by the bento project[6]. Since we started out with a Debian server above let's continue and set up a Debian server using Vagrant.

To create a Vagrant managed virtual machine we need to write a `Vagrantfile`. A `Vagrantfile` is a configuration file used by Vagrant, written in Ruby (a fact that we will exploit later when defining more complex Vagrant setups).

Vagrant can create a `Vagrantfile` for us as a starting point. This is helpful when starting out but I'm sure you will soon find this template unnecessary, even irritating because of all the comment lines it contains. That said, let's create our first `Vagrantfile` using Vagrant.

```bash
mkdir myvagrant
cd myvagrant
vagrant init bento/debian-10
```

Simple as that, the `init` command instructs Vagrant to initialise a `Vagrantfile` such that it specifies a virtual machine based on the `bento/debian-10` box.

Where does Vagrant find these boxes? By default they are hosted on the HashiCorp Vagrant Cloud server.

Before taking a look at the `Vagrantfile` let's start our virtual machine.

```bash
vagrant up
```

You should now see Vagrant downloading the `bento/debian-10` base box, make a clone copy for our project, and start up the new virtual machine. At the end of all this we have a running machine but we are returned to our command line prompt.

As with our manually constructed virtual machine the Vagrant machine is started headless[7]. We access our new virtual machine using SSH, fortunately Vagrant makes this trivial by supplying the `vagrant ssh` command.

```bash
vagrant ssh
```

You should be immediately connected to the Debian virtual machine we just created. Your command line prompt while on this virtual server will be `vagrant@debian-10:~\$`

To leave the server (but have it remain running) simply logout.

```bash
exit
```

[6]The bento project is run by the people who develop the Chef configuration management system. I have found these boxes to be quite sound for my needs.

[7]Virtual machines can be started with a display, but we almost exclusively want headless servers, so Vagrant's default behaviour is ideal.

You will be returned to your host computer's command prompt.

Chapter 7

Infrastructure as Code

Central to our approach is the idea of infrastructure as code. What does this mean?

Hopefully we are all familiar with the fact that underlying the websites, applications, databases and other software in our charge is code. This code is transformed by other software (typically a compiler or interpreter) into a form that is ultimately executed under the management of an operating system (another piece of software). This code is, at least in well run modern environments, held in a version control system.

It seems peculiar then that all of the surrounding configuration, setup, and monitoring necessary for maintaining a system should be recorded in documents that are followed and maintained (hopefully consistently and accurately) by engineers. These documents are often called 'run books' and there are even tools that attempt to partially automate these run books[1]. This never works!

This problem is solved by replacing the documentation and manual execution with code that is executed by software designed to manage infrastructure. This may seem odd, sending software to look after software, but bear with me.

Using code to manage our system also provides other significant advantages; it is repeatable, and it can be put into version control. This may seem like a slight thing after all the manual documentation can be put under version control too. The difference is the repeatability. With the best will in the world humans suck at writing and following documentation. This is a combination of the document being unclear or incomplete and the engineer following the document interpreting it incorrectly or, with the best of intentions, 'correcting' the document as it is being implemented (often without updating the document).

We have removed the human element between writing the system specification and its implementation. Any problems with the interpretation of the infrastructure code can be corrected and the code re-run until the problems are resolved. In other words our infrastructure becomes subject to a cycle of debugging, just like software. Our objective, if it is not already obvious, is to produce instructions that can be executed without human intervention in a repeatable manner such that we can create or recreate our infrastructure.

Another often overlooked advantage of this infrastructure as code is speed. Anyone who has been involved in the manual deployment of a system will know it is not only error-prone (consequently one is never sure one has the

[1] Tools like RunMe.

system as specified) but it is also slow, painfully slow, and labour intensive. Infrastructure as code is front loaded, meaning most effort is invested at the beginning of the project. Once at the point of deployment it (ideally) is a push-button deployment. It can still take hours to deploy a system from scratch, but compared to the days or weeks involved in a manual deployment this is not a long time.

7.1 Less Talk, More Do!

Complexity ahead!

Although the steps to setup this 'first server' are trivial the material explaining how this setup works is more complex and you may need to come back to it after reading more of the book. I have tried to keep is simple (both in implementation and description) but some of you may feel overwhelmed. DON'T PANIC! If you have trouble just skip ahead and come back when you've learned more.

Setup to Follow Along

If you have not done so already, setup according to §1.3.1 and, if you need to archive any current class files §1.3.2.1.

```bash
cd dfs
```

On Mac/Linux:

```bash
cp -r dfs-material/dfs040cp010 classroom
```

On Windows:

```bash
xcopy dfs-material\dfs040cp010 classroom /E
```

All Platforms:

```bash
cd classroom
vagrant up
```

Although this may take a while to run I think we can agree that it's simple enough[2]?

[2] If you have something already occupying port 35555 on your host computer you will get an error message from Vagrant that the port cannot be mapped. In this case edit `Vagrantfile` and change the line `website_port=35555`, changing 35555 to a free port

You have just created a web server with simple static website, set up a basic firewall, and run some smoke tests on the server to ensure basic functionality. Furthermore, the system you end up with will be functionally the same[3] as mine. We have a repeatable process.

On your host computer, open a Web browser and access `http://localhost:35555`. This should display the home page of our sample website.

That process illustrates just part of the power of infrastructure as code.

Let's take a look at this example.

Before we take that closer look...

- This is a simple example. Only one server is created the 'system' it implements is simple and the tests are all run locally.
- This is a very simple configuration, no attempt has been made to make this 'production ready'.
- Tests are far from exhaustive and intended as a simple illustration.
- In the following discussion I am not presenting all of the details, just the highlights. The rest of this book will flesh out the details, how the various tools are used, and more importantly it will offer guidance on how to approach solving some of the problems we encounter in DevOps.

7.1.1 The source

The first thing to note is that the entire 'specification' for this system is held under version control in a Git repository. Any problems detected with the system would be corrected in this repository and then redeployed from there (more on this later). No modification would be made to this system directly (by, for example, someone logging on to the system directly)[4].

This idea that our servers should be 'untouched by human hands' is perhaps the most foreign idea for most people. We are so used to logging on to servers to diagnose and fix problems that being told this is no longer the way to work is often greeted with 'how can we possibly support our customers this way?' Not only is this possible it is essential.

7.1.2 The `Vagrantfile`

This file tells the `vagrant` software how to create and initialise our server.

This `Vagrantfile` is the highest level of our local system specification but contains very little of the actual server configuration. This is deliberate. We want as much of the configuration to be reusable in different contexts; building

[3] I say 'functionally' because there may be some variation in specific versions of some packages installed. For example, although we have fixed the version of the Python `pytest` package this may in turn install dependencies and these may be limited to a range of versions rather than one specific version. Whether this lack of precision is important is context dependent.

[4] Strictly, of course, in this simple example we are logging on the server to run the `salt` and `git` commands, but that's all and we could run these remotely if we wanted

a physical server, building a cloud server, as well as building a local Vagrant server.

To this end we include in the `Vagrantfile` only those elements peculiar to setting up a Vagrant server. Notably, the line that create a `grains` file for the `salt` configuration:

```
Vagrantfile
15 config.vm.provision "shell", inline: "mkdir -p /etc/salt;
   [ -f /etc/salt/grains ] || echo 'is_vagrant: True' >
   /etc/salt/grains ; sed -i -e
   '/^is_vagrant/{s/is_vagrant:.*/is_vagrant:
   True/;:a;n;ba;q}' -e '$ais_vagrant: True'
   /etc/salt/grains"
```

This is broken into two parts. The first creates the `grains` file if it does not exist:

```
bash
1 mkdir -p /etc/salt; [ -f /etc/salt/grains ] || echo
  'is_vagrant: True' > /etc/salt/grains ;
```

The second is redundant when the provisioner is first run, it is included in case the provisioners are re-run. This second line ensures that the `is_vagrant` grain is set correctly.

```
bash
1 sed -i -e '/^is_vagrant/{s/is_vagrant:.*/is_vagrant:
  True/;:a;n;ba;q}' -e '$ais_vagrant: True'
  /etc/salt/grains
```

This line tells Salt that this server is a Vagrant managed server. This is required so we can make adjustments to the server configuration. In this instance we ensure that the firewall allows `ssh` connections. We could have used other features of the server (such as the existence of the `vagrant` user account) but attempting to 'fingerprint' the server like this is prone to all manner of difficulties (what if someone creates a server with a `vagrant` account). Being explicit using a grain like this is better.

7.1.2.1 Salt provisioner

Given that the Vagrant Salt provisioner can be used to perform a similar function to the provisioning script described in §7.1.3, why not use it? In general I want my provisioning method to be portable so that I can use the same provisioner to build a 'real' server (physical or virtual) that I use to build the development and test server. This flexibility is denied us if we lock ourselves in using a Vagrant specific provisioner configuration.

Does this mean I would never use the Vagrant provided `salt` provisioner? Not at all. As I said, 'it depends'. I *would* use it if I knew that the machine being specified was uniquely a Vagrant machine, or when I knew that everything

in my specification can be provided by a `salt state.highstate`. I'll describe some of the issues I see with the current setup in §7.1.3.

7.1.3 The provisioning script

The `provisioning/build` script is a simple beast that replicates much of the Vagrant `salt` provisioner.

Installing `git` allows us to pull the formulas for `iptables` and `nginx`. I could have installed the `salt` configuration and then invoked a state that installed the formulas before using `state.highstate` to complete the machine setup, but that seems unnecessary and does not eliminate the need for a script[5].

Issue: I have not specified the versions of these formula to be used. Instead we simply pull the 'latest' available. For a non-critical system this is fine but as we will see this may not be appropriate to your project.

Next the script installs `salt` using the `bootstrap-script` provided by SaltStack. This is essentially what the Vagrant Salt provisioner would do too.

Then, the `salt` configuration is put in place and the Salt minion configuration (again, Vagrant will do these tasks too but relying on the Vagrant provisioner locks the solution into Vagrant while this script can be run on any Debian base system).

Finally we run `salt` to assert the highstate, this completes the setup of the machine.

Having set up the server we now deploy the website. To keep things simple this is a static web site and is controlled as source in a Git repository. The website is not deployed using `salt` but it could be.

Now that everything is deployed we can run some smoke tests. To do this I simply run the `infratest`s held alongside the configuration.

Issue: I prefer to have tests hosted on an external server even if they are run locally but in a single server system such as we've just created that is not practicable.

7.1.4 Salt configuration

The majority of the 'hard work' of setting up the server is done by `salt`. We cover Salt configuration in more detail as we work through this book (and in much more detail in *Saltstack from Scratch*[Boo20d]), here I will just outline briefly the components used in this example.

Salt's configuration is provided in two parts:

states These specify the desired state of our server, they are held under `/srv/salt` (custom states) and `/srv/formula` (standardised states)[6].

pillar Files under `/srv/pillar` provide data which is subsequently used when processing states.

This configuration is made of of three parts:

[5] I could also have set up a Git file system—discussed in detail in [Boo20d]—but I think the current approach is simpler for new users.

[6] The custom/standard distinction is my own, but I think characterises the difference well.

- Standard tools
- Firewall
- Nginx

Standard tools installs Git and Tree. We have installed Git already in the provisioning script, but this install specifies the version of Git that we consider correct. This ensures our server has a consistent configuration.

The firewall is set up using the Linux net filter using the `iptables` tool.

Nginx is the tool that will server our website.

7.2 What about the data?

There is an obvious gap in our system rebuild; data.

We can build our server simply enough as outlined in the previous section, but what about the data we enter? Admittedly, not an issue in our simple example (beyond treating the web site as standing data) but certainly an issue in general.

There are different classes of data controlled and generated by our system. Broadly we can consider three classes of data:

- Transient—this is data that we can generally afford to lose. Or, more accurately, its loss has little impact on our business. I include in this classification things like log files[7].
- Standing—this is data that changes seldom if at all. An example might be the tax rate applied on our shopping website.
- Dynamic—this is what most people consider 'data'. This is all the information we capture that, if lost, would have a material effect on our business.

The management of data is a complex topic and we will discuss it at length in Chapter 16.

[7] The exception to this being in regulated environments where log file data often forms part of audit data and consequently becomes of significant value and consequently is treated as dynamic data.

Chapter 8

The Master Server

In building our system we need to start somewhere. It is tempting to start with a development environment but we will instead start by building a server to provide some basic facilities.

- A version control repository—into which all our sources (this includes our documentation) will be placed
- A 'bug' tracker—to keep track of issues and defects.
- A build system—a 'doer of things' to automate the transformation of source into product and ultimately to build our infrastructure.

Although I said we will be starting by building a server rather than a development environment we will still be creating a virtual server *for* development and testing.

8.1 Preliminaries

If you have not done so already, get the material for this book to make it easier to follow along (see §1.3.1)

8.2 Base server and operating system

The master server will be an x86 machine with 8GB RAM, 4 cores, and 250GB SSD[1]. Onto this hardware we will install the Debian operating system. At the time of writing the latest stable version of the Debian operating system is Debian-11 codenamed 'Bullseye'.

The `Vagrantfile` for our base server will therefore look like this;

[1]You might be thinking, "that's a pretty weird specification for a server!" And you're right. It just happens to be the specification of a spare machine I have lying around (recall, we're working with what we have available.)

Vagrantfile

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  config.vm.box = "debian/bullseye64"
  config.vm.box_version = "v11.20220912.1"
  config.vm.provider "virtualbox" do |vb|
    vb.memory=8192
    vb.cpus=4
  end
end
```

Try it now. Create a new directory, move into that directory, create a new `Vagrantfile` and enter the content above[2], save the file and then `vagrant up`. You should end up with a new VirtualBox server set up with the Debian operating system installed.

The keen-eyed amongst you will have noticed that there is no mention in this `Vagrantfile` of disk size. The Vagrant box will provide the initial system disk, changing this size of this disk involves more than a simple directive in the `Vagrantfile`. For now we will live with the disk provided and address this issue if the need arises.

Another key feature of this `Vagrantfile` is the `config.vm.box_version` line. This 'locks' our configuration to a specific version of the base box. This ensures that any configuration work we do after this is building on a known starting configuration. It also means that anyone using our `Vagrantfile` is sure to also deal with the correct starting point.

If we omit line 6 then Vagrant will use the latest `debian/bullseye64` available on the Vagrant Cloud box catalogue. As we develop our initial solution this may be an option we want to use, but as soon as we are to share our configuration with others providing the version is important.

'Locking' our version like this is important for consistency later but has an associated downside. As our configuration ages this version of the Vagrant box becomes increasingly out of date with respect to the Debian releases. We can deal with these issues in a number of ways, among them the following.

- Review and update the 'locked' version of the box periodically.
- Build our own custom Vagrant box.
- Update the Vagrant VM as part of our subsequent configuration.

Each of these has pros and cons and we will consider each later. This 'lock' versus 'free' version issue will be a recurring one. For now we have a potentially more serious issue to deal with, the Vagrant box we rely upon is hosted on the Vagrant box catalogue hosted by HashiCorp and there is no guarantee that this box will remain available indefinitely. Worse, we have no control over that box's availability, the Debian team or HashiCorp may choose to deprecate it.

[2] You may need to reduce the assigned memory and cpu count, depending on your host machine's specification.

This raises a general issue; we should, so far as practicable, make copies of all resources we rely upon such that we can maintain direct control over them. For the box it would simply mean cloning the `debian/bullseye64` box into our own Vagrant box catalogue and then using this local Vagrant box catalogue as our primary source. Since we do not currently have such a catalogue we should add this to our backlog.

It is also important to emphasise that our development and test configuration is based on a virtual machine configuration suitable for use with Vagrant on our desktop machine, it is not necessarily a precise match with the base configuration we will have on our target system. How should we deal with this? One obvious solution is to carefully control matters such that these differences are eliminated. Another, perhaps more pragmatic, approach for our purposes here is to test for the important features that we must have in our base configuration and to make our configuration appropriately adaptable. We will investigate these options as we proceed.

Part of the Vagrant specification for a box requires that an SSH server be provided to allow Vagrant to communicate with the VM to both provide access (via `vagrant ssh`) and for further configuration, to which we now turn our attention.

While Vagrant requires SSH access we will likely *not* install SSH access on the majority of our 'real' servers. So, one of the major discrepancies we have between our Vagrant development and test system, and our formal test and production system is the presence of SSH. We can minimize the impact of this discrepancy by severely limiting access to SSH. We will address this shortly.

Insecure by design

Vagrant boxes will generally not have much security by default (see §8.3). This is by design. Vagrant is intended primarily as a development tool and is not suitable for controlling or deploying production systems, consequently you will find that most generic Vagrant boxes, such as `debian/bullseye64`, are fairly 'bare bones', consisting of a largely default installation. However we want our local system to reflect our formal test and production environments as closely as practicable[a]. We will secure our servers as much as is practicable given the aforementioned Vagrant requirements.

[a] This is a game of diminishing returns. As our system grows there will come a point where replicating it for developers locally will be impractical, however we can reproduce the features essential to specific teams (although this requires a level of understanding of the system seldom found—but our mission is to foster this level of understanding, so let's continue).

8.3 Vagrant SSH

Vagrant boxes set up SSH with a non-privileged user account `vagrant`. The `vagrant` account is configured for both password (also `vagrant`) access and a public/private key pair is preloaded to allow Vagrant SSH access to the server without the need for messy password integrations.

The upshot of this setup is that Vagrant controlled servers are inherently insecure by default as the username/password pair (`vagrant/vagrant`) is common knowledge.

The normal use-case for Vagrant renders this insecurity moot because we would not normally make these machines accessible outside our host computer and certainly not on a public network. The difference is significant to our current setup because as we secure our server we need to ensure access to the `vagrant` account via SSH so that Vagrant continues to work.

We could secure our Vagrant setup further by changing the `vagrant` account password, but frankly this is overkill for our purposes. Our Vagrant systems are intended only for use in development on individual host computers, which should themselves have host firewalls preventing unwanted network access to the Vagrant systems.

The most important thing we need to do though is isolate, so far as practicable, any Vagrant specific configuration such that it does not interfere adversely with our development. In other words, we want to avoid making assumptions that are only applicable in our development Vagrant environment. Since our users (developers) may be unaware of these issues, we need to isolate them as far as practicable as part of our configuration.

8.3.1 Vagrant provisioning

As our first step in configuration let's do something simple. We will need the Git system installed on our new server so this is a good simple thing for us to install.

Modify your `Vagrantfile`, inserting the `config.vm.provision` instruction.

Vagrantfile

```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3
4  Vagrant.configure("2") do |config|
5    config.vm.box = "debian/bullseye64"
6    config.vm.box_version = "v11.20220912.1"
7    config.vm.provider "virtualbox" do |vb|
8      vb.memory=8192
9      vb.cpus=4
10   end
11   config.vm.provision "shell", inline: "apt-get install
      -y git"
12 end
```

The new line (11) instructs Vagrant to use its SSH connection to run the shell command `apt-get install -y git` on the guest operating system (our new server).

By default Vagrant will run all such provisioning shell commands `sudo`, that is with elevated 'super user' privileges. (The `vagrant` account is an unprivileged account but is configured as a `sudo` user, see `sudo` users.)

Assuming you still have your Vagrant VM running from earlier, you can have vagrant 're-provision' the VM rather than needing to destroy it and start over. On your host computer, while in the same directory as the `Vagrantfile`[3].

bash
```
1  vagrant up --provision
```

By default (without the `--provision` option) `vagrant up` will start a halted or suspended Vagrant VM without running any `provision` entries. *With* the `--provision` option the VM is started (if halted or suspended) and any `provision` commands in the `Vagrantfile` are run.

In our example the inline shell command `apt-get install -y git` will be run and the Git package will be installed on the VM.

If the VM is already running, that is not halted or suspended, the `provision` entries in the `Vagrantfile` are still run against the running VM.

If the VM does not exist (it has not been previously created with `vagrant up` or it has been destroyed with `vagrant destroy`) then the VM will be created as normal and the `provision` entries will be run as part of the creation of the VM.

This simple approach to configuring our VM seem okay for simple things but has one major problem, it is not portable. Suppose we want to apply this configuration to another server, one not controlled by Vagrant. We would need to somehow extract all the `config.vm.provision` directives to apply the relevant configuration. Not very practical.

What we need to do is decouple the configuration actions from the Vagrant mechanism that invokes those actions.

8.3.2 Vagrant provision by script

Modify your `Vagrantfile` again, replacing the `config.vm.provision` directive.

Vagrantfile
```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3
4  Vagrant.configure("2") do |config|
5    config.vm.box = "debian/bullseye64"
6    config.vm.box_version = "v11.20220912.1"
7    config.vm.provider "virtualbox" do |vb|
8      vb.memory=8192
9      vb.cpus=4
10   end
11   config.vm.provision "shell", path: "scripts/configure""
12 end
```

[3]Technically you can be in the same directory as `Vagrantfile` or any of it's subdirectories, but that's a lot to type every time.

We have added line 11 to copy a file (`scripts/configure`) from the host computer to the VM (into `/tmp/vagrant-shell/configure`) and then execute this script on the guest VM.

Next we need to create the `configure` script. On the host computer, in the directory containing the `Vagrantfile`.

bash
```bash
mkdir scripts
vi scripts/configure
```

Use whatever your preferred text editor is (I'm using `vi` here).

Enter the following into the `configure` file.

configure
```bash
#!/usr/bin/env bash
# vi :set ft=bash:

set -euo pipefail

apt-get install -y git
```

The first line ensures that Linux invokes the correct interpreter when none is specified. The second line is a 'mode' line telling `vi` that this is a `bash` script (not important if you do not use `vi`, but I do so adding this mode line is habit for me). Line 4 ensures the script fails early and hard if it has any errors (not strictly useful in such a short script, but a good habit to acquire).

Line 6 is the important line and reproduces the install of the Git package.

This may all seem rather overkill, and it is for such a trivial example, but it illustrates an important principal. Moving our configuration instructions into script means we can copy that script to any system we want to configure and run it. This configuration is now independent of Vagrant, relying solely on Linux script interpreters[4]. The only parts of our configuration process that are tied to Vagrant are the `provision` directive, these contain no configuration information other than which script to upload and run for this particular VM.

8.4 What versus How

Notice that our configuration is really a script detailing 'how' to impose our configuration. To use this configuration we need to know some things about our target system. For example, we need to know that the system supports installation of packages using `apt-get`, and we need the system to run `bash` shell scripts.

The second requirements (the need to run `bash`) could be a simple prerequisite, we are using `bash` as our configuration tool. The former though is more questionable.

[4]Well, it also depends on the APT repositories too, but let's keep things simple.

What if I want to run this configuration on an Arch based distribution? These distributions use `pacman` rather than `apt`. What about on RedHat distributions where `dnf` is the preferred package manager?

The issue is this; my configuration is really saying ‘I want to ensure Git is available’. This requirement is independent of the underlying operating system or distribution or that operating system. All I really want is to have Git available after I have applied my configuration.

Configuration management tends, therefore, to be based on a declarative system. The configuration is a statement of ‘what’ should be true on the system if it is configured correctly. The configuration manager is unconcerned with ‘how’ the configuration is asserted and only concerned ‘that’ it is asserted. In other words, I don’t care about which installation method is used to install Git I only care that once my configuration is applied Git is available.

In a trivial sense my script could be something like the following (if you’re following along there is no need to make these changes, I’m just illustrating a point).

configure

```
1  #!/usr/bin/env bash
2  # vi :set ft=bash:
3
4  . config-tool.sh
5
6  set -euo pipefail
7
8  git-installed
```

Our configuration now just states ‘after running this configuration the system must have `git-installed`’, or put another way, ‘any system that meets the requirements of this configuration must have `git-installed`’. All of the detail about how the configuration tool should verify that Git is in fact installed or how if should be installed if it is not already, is irrelevant to the configuration itself.

A naive implementation of `config-tool.sh` might look something like the following.

config-tool.sh

```bash
#!/usr/bin/env bash
# vi :set ft=bash:

set -euo pipefail


# Figure out package tool
PKG=""
for pm in "apt dnf pacman"; do
  if command -v "${pm}"; then
    PKG="${pm}"
    break
  fi
done

git-installed () {
  # If git IS installed, we're done
  command -v git && return

  # Otherwise try to install it
  case "${PKG}" in
  "apt")
    apt -y install git
    ;;
  "dnf")
    dnf -y install git
    ;;
  "pacman")
    pacman --sync --noconfirm git
    ;;
  *)
    echo "No package manager found" >2
    exit 1
  esac
}
```

Obviously this script is deficient in many ways (not being very generalised, not accounting for different package names, not covering many distributions, not handling errors, etc.) but in principle it allows us to say `git-installed` (the thing we want to be true) in our configuration and leave all the messy details (of how to make it true) to be figured out by the configuration tool.

As you might imagine, we are not going to be writing our own configuration management system!

8.5 Our core configuration tool

In the previous section we converted our initial configuration steps into a Bash script. This decouples our initial configuration from a Vagrant specific format

(the `Vagrantfile`) and places it into a more portable form that can be used to provision not only Vagrant machines but also cloud servers or physical machines. This has the benefit of making our configuration something defined independent of the underlying implementation of our server.

We also showed the separation of the configuration from how to achieve that configuration. This is an important abstraction allowing us to focus on 'what' our system should look like rather than 'how' to make our system look the way we want. (This is obviously an ideal and reality being the complex mess it is seldom this clean cut in real life. But, hey, that's what we're here to learn about!)

The Bash script is certainly a move in the right direction and there will be many more such scripts required to set up our servers, but Bash scripts are tough to get right and are seldom concise in expressing all the minor variations required when configuring multiple servers. Fortunately there are specialised tools for managing our server configurations.

There are many tools to choose from in the configuration management space. Which you choose may depend on a number of factors.

- Does your team already have experience using a particular tool? This could result in a default decision based on current experience, the tool may be adopted because people are comfortable using it, or rejected because of past problems with the tool.
- Does your team have deep knowledge of a particular language? This can influence tool choice because particular tools are written using, or are designed to integrate with, specific languages. For example Puppet is Ruby based, while Saltstack is Python based.
- Cost.
- Supported platforms.
- Legacy configuration. Either a need to adopt existing configuration or to migrate from a legacy configuration.

We will use Saltstack for the following reasons:

- It is freely available.
- It is based on Python and Python is pretty much the *defacto* standard scripting language on Linux (and is preinstalled[5] on most Debian installations, including this `debian/bullseye64`).
- It is so much more than a configuration management tool, it can be used for monitoring and self-healing of systems, features we will use much later in this course.
- It has tools for deploying cloud servers, which we will use later in this course.
- It can be used standalone, over SSH, or as a full bus-oriented master/minion system. These options are discussed briefly in later sections and more fully in *Saltstack from Scratch*[Boo20d].

[5]Python being preinstalled is less of an issue for Saltstack since version 3006 a `onedir` package includes the appropriate Python interpreter for Salt.

8.5.1 Installing Salt

We have seen how trivial installing a Debian package can be when we installed Git. Debian repositories do have a set of Salt packages but they tend to be older versions of Salt and we would prefer to have a more recent version (and have the option to keep our systems up-to-date with the latest releases), so we will not be using the Debian package repository version.

Fortunately SaltStack provide a shell script for installing various SaltStack components. In a similar approach to that used in §8.3.2 we can provide the SaltStack script and run it to install the Salt components we require. We will start with a naive implementation and gradual refine it into a more robust implementation, along with discussion of each refinement.

Edit the `Vagrantfile`.

Vagrantfile

```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3
4  Vagrant.configure("2") do |config|
5    config.vm.box = "debian/bullseye64"
6    config.vm.box_version = "v11.20220912.1"
7    config.vm.provider "virtualbox" do |vb|
8      vb.memory=8192
9      vb.cpus=4
10   end
11   config.vm.provision "shell", path: "scripts/configure""
12   config.vm.provision shell, inline: "cd /tmp && curl -o
   bootstrap-salt.sh -L https://bootstrap.saltstack.com &&
   sh bootstrap-salt.sh -M git master"
13 end
```

We are adding just one line (line 12) but it's doing a lot of work. There are three commands to be run; change to the `/tmp` directory, download (`curl`) the script from the SaltStack website, and finally run that script.

As before we can run this additional provision line using the `--provision` option. On your host computer, in the same directory as the `Vagrantfile`.

bash

```
1  vagrant up --provision
```

This run will take some time as the Salt installation script does a lot of work for us.

While it works, let's consider what we just did (and why it's not a particularly good approach).

We downloaded a script from the internet and ran it into our VM without any checks. This is a security risk on two levels; the source of the script (the `bootstrap.saltstack.com` website) could have been compromised and the script could have been tampered with, secondly, we have no idea what the script is actually doing, where is it sourcing the installation from, what (if any) precautions are taken to ensure the script installs the proper files.

This level of trust may be okay for our 'quick and dirty' development environment, but they are unacceptable for a production environment.

In Chapter 15 we discuss repositories in more detail but for now we should note that sourcing directly from a public website is a security risk. Whether we consider it a reasonable risk comes down to our risk tolerance (see §12.1.1) and this will vary according to context (we may accept more risk in an isolated development environment but less in a live production environment).

The first step in reducing our risk is simple in principle, we download the script to a local file system, review it, and then use this reviewed copy as the source for our configuration. Simple in principle, more complex in practice. Performing such a review is a non-trivial exercise. It is important that this process results in a controlled copy of the script that can be used for delivery but also for comparison when updates to the upstream (original source) are made.

Furthermore, in this case the script itself refers to other repository objects and if we are to be highly risk averse these must also be vetted and local repositories used. Chapter 15 discussed these observations in more detail, for now we will set most of them aside in order to progress with our configuration work (rest assured though, we will return to this issue).

8.5.2 Additional Salt setup

The current Salt setup will not work!

Salt, as installed, is running a Master/Minion setup where each Minion controls a target configuration (in this case the new VM) and the Master coordinates a set of Minions. In order for Minions to find the appropriate Master they use a DNS lookup. By default the Minion will look for the domain name `salt`.

The Master and Minion are running as two services on the VM. We can view their current state using `systemctl`.

bash

```bash
systemctl status salt-master
systemctl status salt-minion
```

You will see that the Minion has failed to start, this is because we have not yet configured a domain name `salt`, so the Minion will be unable to find the Master (even though it happens in this case to be the same VM).

To fix this we need to define a domain name `salt` that the Minion can find and resolve, and then we need to restart the Minion so that is can find and connect to the Master.

To fix this quickly we can add a line to our `/etc/hosts` file to resolve domain name `salt` to the VM itself. As the Master and Minion are running on the same VM we can use the local loopback device `lo`, this has the IP Address `127.0.0.1` (this IP Address is a standard 'this machine' IP address[6]). We could simply edit our `/etc/hosts` file, but this would be useless to anyone building this VM from our `Vagrantfile` in the future. Following the principals

[6] Technically the address block 127.0.0.0/8 is reserved for loopback addresses ([see CVH13, table 4])

of infrastructure as code, this change must be written into our configuration. The modification to the `/etc/hosts` should exist before we install Salt so that it is available before the Minion tries to find the Master for the first time. The obvious place to do this in our current configuration is the `configure` script we started earlier.

configure

```bash
#!/usr/bin/env bash
# vi :set ft=bash:

set -euo pipefail

apt-get install -y git sed

sed -i -e '/[[:space:]]salt\([[:space:]]\|$\)/ {:1;n;bl}' 
 -e '/localhost/ s/$/ salt/' /etc/hosts
```

`sed` is a fairly standard Linux tool available in most distributions (it is available in the `debian/bullseye64` box already). So, why add it to the `install` on line 6? This is a precaution. If our `configure` script is run on a distribution that does not have `sed` installed we want to ensure that it is installed before trying to use it on line 8. If Git or sed are already installed attempting to install them a second time will not cause any error.

Line 8 needs some explanation. This is the line that adds `salt` as a domain name. It may be tempting to try something line `echo "127.0.0.1 salt" >> /etc/hosts` to append a suitable line to the `/etc/hosts` file. But consider what this would do if the `configure` script were run multiple times (as it has been already). Every run would add another `127.0.0.1 salt` line to our `/etc/hosts` file. Not good.

The more complex `sed` command avoids this problem. The `salt` domain name is added only if no suitable entry already exists.

This idea that the script should result in the same output each time is called 'idempotence'. A fancy word meaning 'repeated application without change to the result beyond the first run'. Put another way, any idempotent operation has the same result on our system as the first time we run it.

Our configuration is not entirely idempotent yet. If we run this repeatedly there is a possibility of different results for the following reasons.

- `apt-get install` will install the latest available version of each package, so if either Git or sed packages are updated in the repository between runs then the version installed on our VM will be upgraded; our configuration results in different versions of these packages being installed, breaking idempotence.

- We have not specified a version to the Salt installation script. As with the packages this may result in the Salt version changing if the source repository is updated between our runs of the installation script.

- We have not reviewed the salt installation script to identify if it is idempotent.

The `bootstrap-salt.sh` and `configure` scripts can be combined into one script by adding the call to the `bootstrap-salt.sh` into the `configure` script, then remove the `config.vm.provision` line that calls this script from the `Vagrantfile`. As a final bit of cleanup for this version we change the name of the `configure` script to `bootstrap-masterserver`. The resulting files are shown next.

bootstrap-masterserver

```
1  #!/usr/bin/env bash
2  # vi :set ft=bash:
3
4  set -euo pipefail
5
6  apt-get install -y git sed
7
8  sed -i -e '/[[:space:]]salt\([[:space:]]\|$\)/ {:l;n;bl}'
   -e '/localhost/ s/$/ salt/' /etc/hosts
9
10 pushd /tmp
11 curl -o bootstrap-salt.sh -L
   https://bootstrap.saltstack.com
12 sh bootstrap-salt.sh -M git master
13 popd
```

Vagrantfile

```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3
4  Vagrant.configure("2") do |config|
5    config.vm.box = "debian/bullseye64"
6    config.vm.box_version = "v11.20220912.1"
7    config.vm.provider "virtualbox" do |vb|
8      vb.memory=8192
9      vb.cpus=4
10   end
11   config.vm.provision "shell", path:
     "scripts/bootstrap-masterserver"
12 end
```

We now have one clean script to take a base Linux (Debian) install and add in tools to facilitate our full configuration.

On the plus side, this script is simple enough that it can be modified for alternate base systems without much difficulty and it does so little that debugging it, should the need arise, will be trivial.

On the downside, we are reliant on `bootstrap-salt.sh` which, while not catastrophic, leaves us open to some risks. We are currently pulling the latest version direct from SaltStack's server which means it could change without warning, it also means any compromise to this script or the SaltStack domain

could compromise any server built from this `bootstrap-masterserver` script. For now I think these risks are within my comfort zone so let us proceed.

8.6 Something is missing?

The eagle-eyed amongst you will have noticed a significant omission from our progress so far. Requirements. Well, more specifically we've dived in to building our system and no one has laid out exactly what we are trying to build. Sure we have some vague notion of what we're aiming at but most projects produce depressing reams of requirements before a line of code is written, let alone building any support infrastructure.

The odd thing is that, at least in my experience, these project requirements seldom cover basic project infrastructure. There tends to be an implicit assumption that the project team will just build 'stuff' they need. Most of the time this seems to work... More or less. Often these support infrastructure elements are an invisible cost to the project. A 'build manager' or even 'build team' (the title varies) will spend many hours tending the support elements, fixing problems as they arise, building out additional capacity, rebuilding broken systems, etc. Without any requirements it is impossible for the project to assess whether needs are being met in an efficient manner, and so any associated cost is absorbed into the 'build team' overhead.

Does this mean we should spend an age developing detailed requirements? No. But it does seem odd that there is a tendency for teams to promote the short life cycle, rapid feedback, Agile approach to customers but fails to adopt the same standards for the internal systems.

Let's fix that.

Chapter 9

Requirements

Author Note

This is early draft material, barely more than a stream of consciousness and notes.

Time, mind, and space.

Mind to mind versus mind model mismatch.

The trick is to balance all these factors against available resources.

There is no mileage in spending time developing too specific requirements, but too general requirements will result in wasted downstream effort.

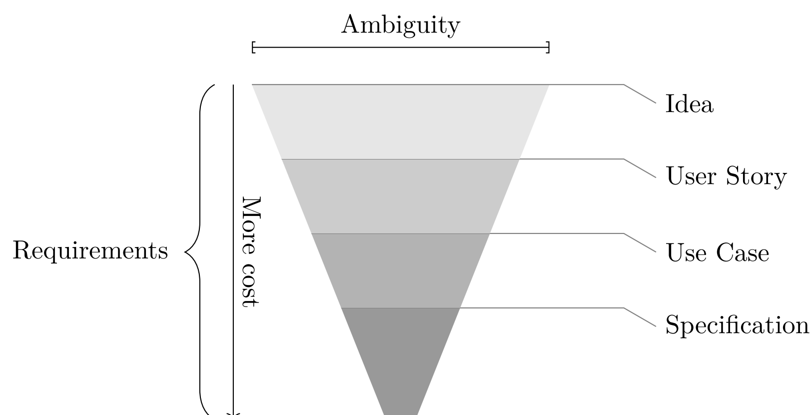

Figure 9.1: The requirements funnel

Requirements communicate the customer's needs, wants, and desires to the developers.

Requirements are always expressed in the language of the consumer[1], the 'domain language'.

[1] I use the slightly awkward terminology 'consumer' and 'producer' because many other terms like 'user' or 'client' are overloaded (people tend to think of 'user' as a person while 'client' is, for better or worse, bound to 'server').

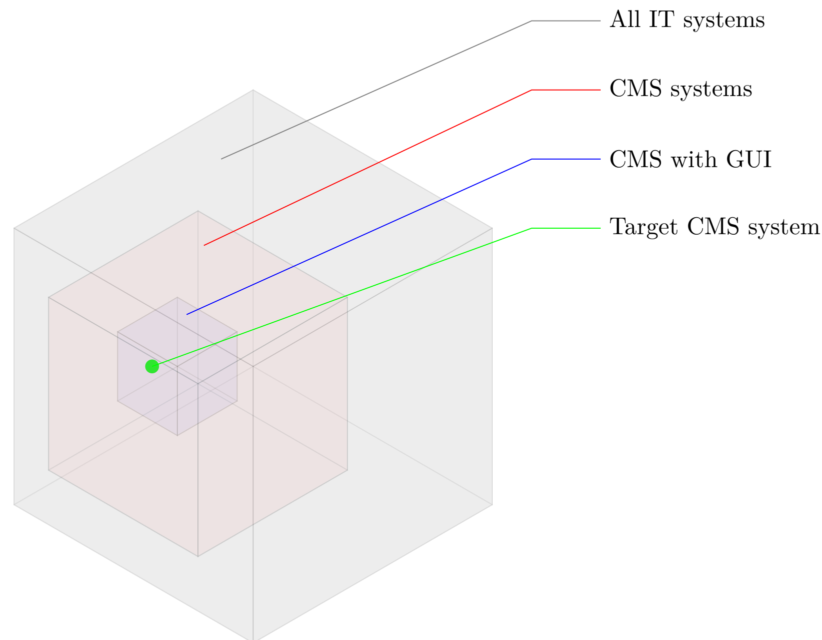

Figure 9.2: The design space

Language is slippery, it is easy for misunderstandings to happen even under the best conditions. We maintain a glossary for all terms used in our requirements, this is the definitive record of meaning in our project.

Requirements need to be specific enough that the development team can deliver the requirements in a timely and accurate manner. This obviously means the level of detail will depend on the relationship between the development team and the consumer. When the consumer and developer are close, for example working side by side, the documented requirements can be less specific[2].

Requirements are constraints in the 'space of possible solutions' Figure 9.2.

9.1 What are requirements for?

Why write requirements?

Short answer; communication.

Requirements communicate information between the person[3] who needs something to the person providing that something. Ideally this communication is unambiguous and complete. That is, the provider never asks the questions, 'what exactly does this requirement mean?' or 'what else do you need?'. Nor does the requester feel the need to ask, 'can we just add. . . '.

Requirements vary considerably in complexity. On a one man project it is common for no requirements to be written. There is no communication

[2]Assuming there are no external project drivers, such as regulations.

[3]I use 'person' here but 'agent' or 'entity' may be more accurate because requirements may also be imposed on our project from systems or organisations that interact with our system, not just humans.

problem, as both the requester and the provider are the same person[4] the requirements can change without notice, there is no need to be specific, there is no problem of ambiguity (or at least the resolution of ambiguity is simple).

On a project with multiple participants the need for clear durable communication increases in proportion to the number of participants (one argument for small teams). One requester and one provider can likely be dealt with informally. Projects tend to be small and communication can be as regular and close as the two participants desire. Once several requesters are involved it becomes essential for the provider to ensure that there is consensus among the requesters. Failing to do so is almost certain to degenerate into chaos as disagreements among requesters drive the provider mad as they pull in different directions.

These communications are compounded further when multiple providers are involved (exponentially so when those providers work in different organisations).

Communication operates over space and time so although it may seem pointless to document requirements when only two or three people are involved remember that in future you may be dealing with more people (not to mention the future selves of the original participants). These 'future participants' may benefit greatly from well documented requirements (not to mention the history of decisions made in reaching those requirements).

If you're working in a team these requirements help to ensure everyone is working to achieve the same result. I think of requirements as guiding lines that specify the essential properties of the system. Within this specification we have plenty of room to be creative but when all is said and done we must meet this specification before all else. These requirements help ensure we are not wasting resources on unnecessary activity.

I want to stress here that looking back on these requirements we may see violations. This is because requirements change. It is likely that the requirements the project has now will change as the project progresses, so we expect the system to sometimes violate these early requirements. This is not a failure on our part, it is simply the way the world works; you cannot see beyond the horizon.

9.1.1 What drives requirements?

People!

Specifically people's goals.

People have goals, these goals drive requirements that in turn drive the need for system development (new features on existing systems or new systems).

Let's generalise this a little. Goals may be personal, 'I want to. . . ' or collective, 'we want to. . . '. Collective goals often stem from business objective, e.g. 'Increase sales 10% during the next fiscal year' is a business objective that may motivate several people within an organisation to start a project and hence write some requirements.

[4]There are, even in one person projects, at least two participants; you now and future you. I have certainly had future me running foul of past me's laziness in not documenting properly.

9.2 Uses of requirements

Beyond their function as an aid to communication requirements have other functions.

9.2.1 Requirements as contract elements

Requirements often form a part of a contract for delivery. This leads to people insisting on the need for a 'complete' set of requirements (and the resulting interminable, and often flawed, requirements phase). After all, if the requirements are not established in excruciating detail beforehand, how can we price up and control the project?

Quite simply you cannot, but then again, even with these painfully acquired detailed upfront requirements, most projects[citation needed] fail to deliver. Why? Because requirements change, especially in light of experience. Also, people are better at telling you what is wrong with the delivered system than they are imagining what they need up front[5].

Which is more productive? Do something quick and realise it's wrong, correct course, repeat. Or, spend months trying to write perfect requirements, develop something based on the wrong requirement, deliver something that does not meet the user's needs but meets the documented requirements. This was realised long before computers were created; models, galley proofs, and prototypes are all attempts to provide customers with something concrete to evaluate and critique before committing to expensive production runs. While these prototypes etc. were intended to be disposed of[6], current IT development encourages continual refinement; taking each product and refining it through a deliberate process (in Agile approaches this is the often misunderstood[7] 'refactoring' process).

9.2.2 Decouple Architecture

We will discuss coupling throughout this course, requirements afford an opportunity to identify boundaries to each component in our system and treat these as stakeholders. This approach has two main benefits: once the stakeholder's needs are identified (the other component's API, for example) no additional interaction should be required during requirement capture, secondly it encourages decoupling in our architecture. Clearly defining system component boundaries is one step in decoupling components.

[5]The old joke springs to mind, "To get the answer to a question online, don't post the question, post the wrong answer. You'll quickly get many people eager to point out that you're wrong and what the correct answer should be."

[6]The maxim 'build one to throw away (you will anyway)' is often quoted from the original edition of *The Mythical Man Month and Other Essays on Software Engineering* [Fre74], Brooks meant we should build prototypes so that we can get early feedback. Twenty years later [Fre95, Chapter 19] Brooks revised this advice saying it was 'too simplistic' and 'implicitly assumes the classical waterfall model of software construction'.

[7]I say 'misunderstood' because too many fail to treat refactoring as a definite step, instead muddling it it with development of new features.

9.3 How to capture our requirements

There are myriad ways to capture and control requirements from simple text files to dedicated software management systems. You may be compelled to use a specific solution (company standards) but for this project we have free rein. Guided by the idea of maintaining simplicity until we need to make things complex we will start with text files. If future demands require that we use a more complex system for managing our requirements we should be able to reformat them relatively easily (generally I find it easier to move from simpler formats to more complex).

The majority of our initial requirements will be technical. Our users being technical too. That said we may have plenty of non-technical people too; programme office team, often project management is non-technical (or at least not deeply technical). There will also be plenty of people who require information from your project (or providing to your project) that will not be technical; customer, users, human resources, etc.

Eliciting and decomposing requirements is a skill unto itself (just take a look at the many books on the topic; search for 'requirements' or 'business analyst'). But we'll have a stab at a simplified version.

There are a few things we can be fairly sure about. We need a way to store documentation (including these requirements). The documentation needs to be accessible. Members of the project need to be able to read and update documents. We want to be able to track the history of these documents. We need to be certain that documentation is controlled and accurately identified. I prefer a layered approach to document control.

- Latest release.
- Current draft
- Version control repository

The latest release is the document from which the project must work. It is read only and published for everyone to access.

The current draft is a utility so that the author can access the document. This copy is only required for non-technical authors who cannot (or will not) use the version control repository. This copy may be unnecessary in many projects but my experience has been that it is often a fruitless task to try to get everyone using a version control tool when they have been used to using a shared file system.

The version control repository holds the complete history of all documentation on the project. Depending on the tool used and the nature of your project or organisation you may need to have more than one repository. Some documents may be confidential or restricted. It may be unacceptable to your organisation for these documents to be at risk of exposure (even to other team members—examples include contract and other legal documents, details budgets and costs). My opinion is that project documentation should be release as permissively as possible.

To be clear, the version control repository need not be a specific tool. It is possible to hold all documentation in a set of directories, using file name conventions to maintain a version history.

You should be realising at this point that our requirements need not, and arguably should not, say anything about the final solution (exception being where such a solution is externally imposed).

9.4 Starting a conversation

The core to eliciting good requirements is having good conversations.

This is the function of 'stories' in the Agile method. Stories are a useful formalisation of conversations between requesters and producers.

Once these conversations, and consequently the associated stories, are mature enough (notice, not 'complete', they are never really complete), we are ready to formalise these stories into requirements. (On projects where Agile is being fully implemented—a customer advocate is on hand—the story may be appropriate as a requirement.)

This is an oft repeated error. Stories are generally not good requirements! They are close, but win no cigar. A story will typically yield more than one requirement. Not all requirements will have a story, but some requirements may lead to a story being generated if the development team need to clarify something (this clarification may lead to yet more requirements).

Stories have acceptance criteria, these are the closest to requirements.

9.4.1 Traceability—sexy!

Before we get too far into this let's talk about traceability. One of the problems faced by a project of any size is traceability. How do we know that we have done the right thing and that we have done everything asked for and only the things asked for[8]?

For any given piece of the final system I want to be able to say, 'this feature was requested in requirement *by*'. If I cannot answer this basic question then why does this feature exist? Why did we expend resources to create it? Why did we adopt the cost of increased product complexity and maintenance?

All the cycling through requirements and stories in §9.4 leads to a history for requirements, this is another dimension of traceability.

If you work in a regulated environment you will find a need to maintain detailed audit-able traceable records. Generally, the more regulated your environment the stricter your traceability requirements. That said, regardless of any external requirements your project will benefit from effective traceability.

Traceability requires that we clearly and unambiguously identify each project asset. Identity schemes are an extended topic, there are as many schemes as there are projects (or so it often seems).

9.5 Testable requirements

Requirements must be testable. Some guidance suggests requirements must be SMART (Specific, Measurable, Attainable, Realistic, and Testable and Traceable). I suggest this is a reasonable guideline but, as with most advice, you need to intelligently apply this guidance. Remember, guidance is not law.

[8]Delivering more than requested means we are 'giving away' project resources.

We can test manually or automatically; automatic testing is vastly preferable for the simple reason that automatic tests are more likely to be run and run more frequently.

9.5.1 Features versus design

Ousterhout [Ous19] suggests that Test Driven Development (TDD)[9] is harmful because it results in 'tactical' development rather that 'strategic' design decisions. This largely depends on how you interpret TDD. If you treat TDD as automated specification verification then it's efficacy depends on the specification. If you specify (requirements) at a design level then TDD will work well. It is true that, as presented by most advocates, TDD will result in technical solutions. I'm also sympathetic to Ousterhout's promotion of design over features.

9.5.2 Test automation

If we can write tests to confirm our requirements have been met, and then automate those tests, we are golden. These automated tests can be run repeatedly as our project moves forward, acting as a ratchet, preventing any backsliding.

9.5.3 Tests we cannot automate

Some requirements cannot be automated, or it may be impractical to automate them. For example, aesthetic design requirements are often subtle and a matter of opinion. They are important, but practically impossible to automate. Even if we somehow establish a baseline (say record the GUI and save this as a 'gold standard', then replay the tests and compare the result to this gold standard. Fine, but even a slight change to the GUI will throw this comparison out and we will have to manually assess a new gold standard. Repeat this too often and the automation of tests start to lose its usefulness.).

Although we may not be able to *perform* a test automatically it is worthwhile setting things up so that test results are *confirmed* automatically (making this confirmation a condition of passing the product). For example, we could maintain a spreadsheet where each line holds a manual test and next to that test the latest result, we can then write a utility to 'test' that all the result fields are 'pass' before declaring the product ready for the next phase.

[9] A development methodology in which tests are written before the implementation code.

Chapter 10

Master Server Requirements—round one

Author Note

This is early draft material, barely more than a stream of consciousness and notes.

We took a general look at requirements in Chapter 9. Before diving into requirements we should first think about why we want them. We start with a need, usually in the form of a problem, usually a problem blocking progress toward some goal. Now our project needs:

1. A central repository for our:
   - changes (issues etc.),
   - documentation,
   - code, and
   - configuration.
2. A means of deploying configuration (build out of machines, ensuring those machines remain consistent with project needs, etc.)

The central repository does not have to be one repository, but should be accessible to all members of the project team.

The current requirements are all technical and focussed on the system development support system. There is a low communication barrier between the customer and the development team (they are basically the same in this context).

We are fortunate that we have no operational constraints at the moment. If we had an operational environment to think about we would need to account for this in our requirements. Perhaps taking in documentation and instructions about how to deliver into production, starting the liaison with operations (getting suitable representatives from operations on the project to ensure that whatever we intend to deliver will be acceptable).

- "The master server must have Salt installed"

- "We want SSH installed so we can log on in order to control the Salt master"

These are questionable requirements as they specify technical solutions. However this is to be expected in a technical domain. Arguably we could step back and express the first requirement as "I need a way to manage the system configuration", but this feels unnecessary at this point.

Before finalizing these requirements though we have others to satisfy;

- "I need to store documents"
- "I need to be able to identify specific versions of documents"
- "I need to be able to build and test versions of the system"
- "I need to be able to control different environments"

In other words our project needs some document management before anything else.

From these high level "needs" we need to derive requirements specific enough that our development team can implement them. This is a rather vague statement, but necessarily so because the amount of effort required will vary according to circumstance.

If we are specifying requirements internally (developer specifying the build system for use by the development team, for example) then we need to invest less effort than if we were writing requirements between different organisations or parts of the business (the marketing department writing requirements for a new campaign management system to be implemented by an external development agency).

Chapter 11

Testing I

11.1 The Purpose of Testing

There are several reasons to perform testing:

- Check that requirements/specification has been met.
- Exercise solutions to try to break them.
- Continuously evaluate solutions to ensure we don't regress.

Each of these can be sub-divided into:

Functional Testing what the code does.

Non-functional Testing how the code does it (such as how fast it performs, the user experience, etc.)

When should we write our tests? The answer depends on the purpose of the test. Tests to check our solution meets requirements are best written first. The process of writing these tests serves to help us understand the requirements. Tests to 'break' our system are best written after our implementation as they help us sure up our solution, uncovering extremes under which our solution fails. Finally, tests to monitor our solution can only be written once our solution is operating.

11.2 Testing Principles

All testing should employ some simple principles.

- Isolate the thing under test.
- Control variables that effect the thing under test.
- Confirm expectations of the thing under test.
- Follow the country code (leave things as you found them).

11.2.1 Isolate the thing under test

Regardless of what we are testing (entire systems down to single functions) we need to understand the boundary of what we are testing and isolate the thing we are testing. This isolation typically requires us to simulate some input at the boundary of the item under test and capture the output (or effect) of the thing under test. These 'simulated boundaries' vary in complexity according to what is being tested. A simple function can be tested with a series of direct calls but higher level user acceptance testing may involve simulating complex external service interfaces (for example, when testing an online store we may need to simulate the external payment processor).

This is often easier said than done, especially for non-functional system level requirements. It is often impractical to isolated a system in a way that will reflect the conditions needed to confirm a non-functional requirement. If you have, for example, an issue with an API being overwhelmed so you decide to rate limit the endpoint. How to best test this? It is simple enough to confirm that the rate limit is correctly set in your configuration, but does this solve the endpoint being overwhelmed? We might test this in a controlled test environment by hitting the endpoint with high load, the problem is that the issue may only occur when overall load is high, or when other processes on the web server are consuming resources, and so on. The best we can hope for is to reproduce *some* set or circumstances that reliably reproduces the issue *before* our change and then confirm that *after* our change these same tests pass. The problem is that these sort of tests tend to be fragile even with the most careful isolation (and controlling of variables, see §11.2.2). It is best to mark such tests as 'fragile' and run them as a special subset of testing where we know to expect some issues (i.e. they are not part of an automated pipeline but run with operator oversight). These sort of tests may also be candidates for monitoring rather than testing.

11.2.2 Control variables

Related to the first principle, we must ensure that we fully control any variable condition in both the environment and the interface of the item under test. This requires a full appreciation of the thing under test so that we can identify all the material variables that may effect our test. For example, when testing a function that sums two numbers we will control the inputs, capture the output, and confirm we get the correct result. This function is running in a complex environment with a specific version of a language compiler, CPU characteristics (type, clock speed, etc.), memory, and so on, but these are not variables likely to effect our current test so we need not control for them. If however we are performance testing this same function to ensure it can meet performance criteria (such as summing one million floating point numbers with ten decimal place accuracy each second) then we may need to control things like compiler version, CPU and memory characteristics, as these may now have an influence on our test outcome.

11.2.3 Confirm expectations

This is the essence of testing. Given some controlled condition does the thing under test meet our expectations? The implication of this principle being that we have clear expectations (these being our requirements expressed in the problem domain language). It may be that our tests *are* the requirements for lower level testing. The higher level our testing the more likely our requirements will need to be translated from the language of the problem domain into the technical language or the solution.

11.2.4 Leave things as you find them

This is an extension of controlling variables. As each test concludes we should ensure we leave things as they were before the test. This simplifies the setup for all subsequent tests and means we can run individual tests or run tests in any order (a powerful capability as our system grows).

11.3 Requirements as tests

In Chapter 10 we started developing some requirements for our development system. This is a good start but as they stand these requirements are written in plain English (or at least a structured version of plain English) and as such they need to be confirmed manually. This is inefficient and error prone.

We need to provide an automated way of checking these requirements.

Chapter 12

Security I

12.1 Risk

12.1.1 Risk tolerance

12.2 Architecture

12.2.0.1 Perimeter

12.2.0.2 Zero Trust

Although not a new idea Zero Trust [Mar94] has become an increasingly popular concept in computer system security. Whereas, in the past, systems were architect around the idea of perimeter security wherein any user or device inside the perimeter was assumed to be, at least somewhat, trustworthy in zero trust systems all devices and users are untrusted regardless of their relation to other elements of the system. Every attempt to transact with another part of the system requires the user or device to authenticate and authorise.

Chapter 13

Architecture I

Author Note

Brief introduction to what architecture is, how it fits into DevOps, and some brief examples.

Chapter 14

Firewall

Now that we have our configuration management system in place we can start defining our configuration.

The first task is to start securing our server. Security should always be defined as restrictive (most secure) first and then only relaxed enough to allow key functions required for the system to operate.

On our final system this currently means, for the master server, that some users will require access to drive Salt (initiating Salt operations) and Salt itself needs network access for communication between the Salt Master and any Minions we define.

On our development system we have the additional SSH requirements for Vargant to work, as noted in §8.3.

Given these two slight differences our configuration must account for environmental differences. Environments are first class concepts in Salt so we will look at the Salt environment system, but I don't think this system is all that helpful and prefer another approach, which I investigate more fully as we proceed.

With that said let's look at our first configuration the server's host firewall.

14.1 What is a firewall?

Any computer attached to a network is vulnerable to attack. A firewall is just one of the tools available to protect a network and the computers on it.

Broadly speaking firewalls come in three types:

Network firewalls These are typically dedicated network devices placed at key points in a network to protect parts of that network, often between untrusted and trusted networks (e.g. your company network and the internet).

Host firewalls These are software that runs on a computer and typically protect that machine only.

Application firewalls These are software, or subsystems, that control input and output of specific applications or services running on a host computer.

It is possible to have hybrid computers that act as both an network firewall, protecting whole or part of a network, but running other software too.

14.2 What does a firewall do?

In general a firewall inspects every network packet passing through it and, based on the 'firewall rules', determine what to do with that packet; drop it, accept it, mark it, pass it to the destination, pass it for further processing, and so on.

Chapter 15

Repositories

Chapter 16

Managing Data

Bibliography

[Fre74] Brooks Jr. Frederick. *The Mythical Man Month and Other Essays on Software Engineering*. Addison Wesley Longman Publishing Co, 1974.

[Mar94] Stephen Paul Marsh. "Formalising trust as a computational concept". In: (1994).

[Fre95] Brooks Jr. Frederick. *Mythical Man-Month, The: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley Professional, 1995. ISBN: 9780201835953.

[CVH13] M Cotton, A Vegoda, and B Haberman. Ed. by R Bonica. 2013. URL: `https://tools.ietf.org/html/rfc6890` (visited on 12/13/2020).

[Ous19] John Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, Palo Alto, CA, Jan. 2019.

[Boo20a] Mark Bools. *Devops from Scratch (Organisation)*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/devops-org/devops.html`.

[Boo20b] Mark Bools. *Git from Scratch*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/git/git.html`.

[Boo20c] Mark Bools. *Packer from Scratch*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/packer/packer.html`.

[Boo20d] Mark Bools. *Saltstack from Scratch*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/salt/salt.html`.

[Boo20e] Mark Bools. *Vagrant from Scratch*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/vagrant/vagrant.html`.

[Boo20f] Mark Bools. *VirtualBox from Scratch*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/virtualbox/virtualbox.html`.

A Brief History of “devops”

Index

vagrant, 7
virtualbox, 7