Kaianga - Project

Shaun Alexander - 18041577 - 247310 - 2020 Sem 2

▶ Table of Contents

Introduction

To build any software you need the right tools, and you need flexibility, the idea that you can sit at any terminal and start coding in a way that is familiar to you gives way to this project. Kainga is Te Reo Moari for home, and this project is about having a familiar developer environment no matter where you are.

Inspiration

The inspiration came because I have multiple OS's that I develop on, Windows, Linux and MacOS, both for work and home, I used to use VS Code but my remote connection for work was over a Citrix Session and some how VS Code caused the Citrx session to crash so I had to get used to different tools. I tried Eclipse because most of the rest of my team used it, but I had dabbled with it at uni and hated it, and using it for work didn't change my feelings for it. I tried Vim because some of my uni friends had always fanboyed about it but I struggled to get the packages downloaded through the protected work network (I now realise that was probably more related to proxy settings than Vim, but the ship has sailed). One of the old beards at work (disclaimer: they don't really have a beard, I'm more referring to their unix wizardry level) uses emacs and kept suggesting I use that, I gave it a go and was scared away, as most people are, then in time I got used to it, then I fell to the *evil* side combining Vim keybindings with emacs power, and then it was time to delve into the beautiful preconfigured emacs that is spacemacs. I knew within hours of trying it that I wanted to use it as my development tool, I set up configuring it in my Linux environment and loved it, I went to use it on my Windows environment and at first glance everything seemed to be the same out of the box which was very promising.

However there was a snag, not everything worked as well in windows, Spacemacs/Emacs functions that relied on unix commands like grep didn't work in the windows ecosystem, they required different configuration, but I wanted one .emacs to rule them all I didn't want to have a linux configuration, a windows configuration and then a MacOS configuration.

It was around now that I had discovered containerisation for another Uni paper I was taking on cloud and IoT. It seemed a Docker container that could run on all machines that housed the emacs/spacemeacs configuration that suited me was the answer, there were some initial hurdles, such as getting a GUI on a container you can read on to learn more about the probelms faced and their solutions.

Problem Statement

The issue I find is that I use different devices and on these are different platforms, and the developer experience is different on every machine, across Linux, Mac OSx and Windows I end up installing three different sets of tools to try and have some kind of workflow consistency across my devices. My editor of choice has become GNU Emacs, I'm a big fan of the opensource community and love how configurable it is but it has one downside, it's almost too configurable for someone starting out, and as such a community driven customisation called Spacemacs exists. Spacemacs is combining Vim key bindings with a set of packages to get a user going with Emacs very quickly. Spacemacs is the configuration of Emacs that I use. And I want to use this in the same way across all my devices.

Overview

The development process involved a lot of trial and error, alot of documentation reading from Docker, Python docs, Stack Overflow and the documentation for the included features for kainga mentioned in the readme. Lots of time spent in the man pages for bash, xterm, and Xvnc.

There was an interesting turn about a quater of the way through this project where I could actually work inside the container and the environment I was developing, so a kind of inception started to take place where I was developing the container and environment inside of the very container and environment I was developing…

At first the project was one big monolithic stack, with a container managing everything up until user interaction, but I realised that there was usefullness in having user config setup seperate to the container because this gave the oppourtunity to also apply the config directly to a compatable machine without the need for containers. I also decided that seperating installation config from runtime config was another good idea which lead to the following break down.

In the end the project boiled down to three main areas, - Machine Infrastructure - The underlying computer, whether that is a physical machine, a virtual machine or a container - Initialisation Config - The installation of software and loading of the config required for the expected programs - Runtime Config - The software customisations that an individual user expects, the things that personalise their environment

For each part of the heirachy we'll look into the projects requirements, decisions made and and issues and solutions we encountered along the way and finally any limitaitons or improvements we could make in the current set up.

Machine Infrastructure

Requirements

The requirement is to be an easily reproducable config/software setup across multiple machines of multiple OS's, but also to be relativly fast to setup and not be too resource intensive in both terms of CPU and memory but also in terms of persistent storage space (HDD, SSD's etc.).

I was doing research into containerisation for a paper I was taking, 158355 Cloud and IoT paper. This led me to discover Docker and containerisation and I imediatley saw it was the answer for what I was trying to achive, VM's are great but heavy, and you can't always install them on every machine. Containers are much more lightweight and portable and mostly installable everywhere so I persued this avenue. But this approach wasn't totally easy to setup.

The minimum the system needs to support is an Xserver, a VNC connection for viewing the Xwindows remotely and an SSH server to allow a secure tunnel of encrypted traffick to the VNC server.

Outcomes

I decided to use Docker because it has a Host Server offerring for all mainstream OS's (Linux, macOS, and Windows). During the development I considered PodMan from Red Hat for a time because they supported systemd which might have helped with running multile background tasks, but they only supported linux hosts which clashed with the ideal of being a multi OS solution (there might be a way to still pursue this, see the Limitations and Future Considerations for this section)

One decision that is worth mentioning here is the decision to use a Linux image inside the container, this came down to two things, I chose Debian because of my familiarity

with Debian from experiments with Raspberry Pi's and Raspbian also the fact that Emacs is best supported on Linux and the Spacemacs experience felt the most seemless on a Linux operating system.

The process involved lots of trial and error, Docker allows you to clone images but from a security point of view I wanted to make sure that I knew everything that I had chosen to install and why it was configured as such. Below is a list of issues and solutions that have all been implemented in the Dockerfile, the build instructions to Docker, to be able to build the same container every time.

Particular attention was taken to keep the container image as small as possible and to only install what was required in order to achieve the minimum functionality of a container that provided an Xserver, and a VNC connection of an encrypted SSH connection.

The results of this work are known as Matapihi, Te Reo Moari for window, as this container becomes the VNC window into the GUI Xserver environment running on the container.

Matapihi is the container image and the initialisation script that runs on it, the process is as follows. 1. Config file is populated with relevant information such as image version number, secret file locations and other build args 2. Image is built using specialised script (python file `docker_compose_with_secrets.py`) 3. Image is run as a Contiainer (see kainga readme for more info) 4. User connects with SSH and performs some security enhancments, and sets up the SSH tunnel to the container 5. The user connects a VNC viewer through the SHH tunnel to the container 6. This for the first time launches the initialisation script which prompts users for scripts to run at login, these scripts need to be blocking, example launching firefox or in this case emacs in the foreground. 7. The Xserver will stay up untill the user exits the running program 8. When a user reconnects the same start up script that was previously loaded will run again, it's important that the start up scripts (and exit script) are idempotent for desired results.

This provides the infrastructure platform for the rest of the environment to be installed on top of.

Issues and Solutions

- **Spacemacs is better in a GUI**
  Docker containers are mainly set up to host shell applications, as such the base images don't contain an X server for rendering GUI applications and the console connections are all text based. While emacs and Spacemacs both can run in a shell the user experience is much nicer in a GUI.

  Solution
  The solution is to install an Xserver onto the Docker image and a virtual network computing (VNC) server to be able to connect as a remote desktop and interact with the chosen application running in a Docker container. My first avenue was to install a full Desktop environment onto the linux core and a VNC server to be able to control it remotely. This was expensive in terms of size, around 2.5gb and consumed a lot of memory during operation, at that size I didn't think it would be very scaleable and needed a different approach. Then I discovered x11vnc a Linux package that doesn't require a desktop environment but can just forward the Xserver output and allow interactions, so instead of remoting into the full desktop environment, we remote into just the chosen applications window. But x11vnc isn't an Xserver so we need an Xserver for Spacemacs to render in and then to forward it to the VNC client, xvfb is such a package, it can make a virtual frame buffer which removes the need for the desktop environment and Spacemacs can render into this virtual frame and x11vnc can serve this to any vnc client connecting to the VNCserver.

Whilst this solution worked well there was an even better solution in the form of Xvnc server, an X and VNC server in one package, this saved a futher 100mb in the continer image and has the added benefit of beinga able to resize the Xserver to suit the display size of a compatable VNC viewer. Using the aforementioned x11vnc and xvfb worked well to determine the proof of concept as I was working in multiple areas at once, Docker configuration, Xsession Configuration and user config loading, so whilst I used x11vnc and xvfb for most of the project due to the seperation of concerns with this heirachy it made it very easy to change the undelying X and VNC server with realtivly little effort. One of the main reasons for changing was upstream maintenance of the x11vnc package, it appears the main contributor is no longer contactable and it has been stagnant for some years where as the Xvnc project is well supported and used by the Debian package of xtigervnc-standalone-server used in this implementation

- **Screen resolution**
  One of the ideal user experiences would be to have the Xwindow screen resolution automatically set to the size of the VNC viewer used to connect to the VNC server.

  Solution
  One of the advantages of switching to Xtigervnc is that it allows for this very thing to happen, but currently only for tigervnc clients but it certainly works from the containers point of view. When it comes to user land an improvement would be to resize all windows when a screen resolution is adjusted, this is discussed in more detail in the Limitations and Future Considerations of the Installation Config Section

- **Security**
  So much could be written about security here, theres two things to consider, security of the build process and security of the conatainer at runtime, where also connecting to the container over the wire. For the build process the use of passwords would mean that any password used at build time would be available to anybody that later pulled the image from Dockerhub. For runtime there was alot of information about forwarding X servers being a security risk and slower than their VNC (Over SSH counterparts). One of the many articles discussing this can be found here.

  Solution
  The solution for part one was to use Docker Build Secrets at the time of developing there was no way to use these with docker copose so I wrote my own build script (`docker_compose_with_secrets.py`) which would read a config file and build the right arguments to pass to a Docker Build command. This allowed me to pass in file paths that contained the secrets rather than the secrets themselves so no meta data is left in the docker layer of the images. Later I decided it was safer to force users to renew their passwords on initial run so the use of there is less need for the script but would still serve a use if you wanted to build a container with hidden secrets but still host it online in a public domain.

  For the second of these issues with the GUI connection to the container VNC required using self signed certificates or a Certificate Authority issued Cert for the full scale encrypted traffic using the VNC Server implementation but its also well documented that you can forward ports over the internets defacto encryption software SSH. So the SSH server handles the encrypted traffic connection and user authentication, and at the VNC client logon the client needs to authenticate again with another password for the VNC server.

- **Build Automation**
  Docker is wonderful but you can end up needing to pass in a lot of commands to build the images securely.

  Solution

There is docker compose but as mentioned above there was a need for developing my own build script to incorporate Docker Build Secrets, this also had the advantage of being able to read from a config file so I didn't need to remember all the arguments I needed to pass into the build each time I wanted to test a container.

- **Supporting Mac IO HCI**
  Mac Laptops and keyboards/mice dont have some common keys or buttons that Linux applications expect, namely the middle mouse button or the insert button, both of these are expected to be able to paste into an xterm window.

  Solution
  After scouring the xterm manual and experimenting with the options regarding custom key bindings and Xresources configuration I was able to determine a way to add right click and ctrl + shift + v paste to support Mac hardware using the Xresources file.

Limitations and Future Considerations

- **docker console at root**
  Currently if you log into the container through the docker host you will be granted acess as root without needing to enter a password, this is a side effect of needing to run an SSH server as part of the startup command, the SSH server needs to be started as root but we run the Xserver in userland. I'm not sure how much of an issue this is given that if somebody had access to the docker host like that they would have access to more sensitive areas of your computer than this container. I would need to look at configuring this a different way so that we could launch the SSH server earlier in the docker file.
- **No vnc**
  No Vnc is a client for VNC connections that can run in a browser, this could be a way to allow even easier access to the VNC server, it could be an interesting develoment if the container also served the VNC client software over the internet to allow VNC connections through the browser.
- **Multiple Clients**
  I can see a use case for wanting to connect to the container again, to a different Xserver display, for example if you had one project open in one VNC client and you wanted to seperate it totally from your work on another project, or you wanted to take advantage of multiple screens by having a VNC client runnning at fullscreen on both physical displays. I see this as a lower priority improvement as the nature of containers is that if you want to duplicate the functionality you can just spin up another container and access it independently.
- **podman systemd**
  One of the limitaitons of Docker Containers is that the images don't support systemd very well, the idea is that each container should be running an individual service, an ethos we are clearly breaking in this implementation, unless you argue that the service we are offereing is a home like environment wherever you go. Systemd is useful because it can start multiple programs and restart them on failure, and Podman (a similar containerisation technology to Docker) supports use of systemd as this article shows. Running systemd in a container could help deal with the first limitation about accessing the console through the docker dashboard as root by allowing us to launch the SSH server as root, but have the app entry point started as the user. This would also avoid the need to have the Xvnc server running in a loop because it could restart the VNC program on exit. I steered away from podman because it only runs on Linux hosts and I wanted to be able to run the kainga suite on any OS, but containerisation is supported by the open container initiative which both docker and podman are members of so in theory I should be able to define and build my container from a Podman image on a linux host and then run it within a docker host environment on Windows or Mac OSX.
- **SSH script to rename VNC passwd**

At the moment it's quite a manual process for the user to SSH into the container to set a new password for VNC and to regenerate new SSH keys for the matapihi container, it would be good to to capture this into a script that the user could run more easily.

- **Certs SSH Keys**
  It would be nice to also add instructions (or a script) for adding the public SSH key of the physical machine to the container and to turn off password authentication so that only known hosts could connect to the containers
- **Machine Names**
  Docker images contain a random unique string as the hostname, it could be nice to generate this as a user set hostname with a unique suffix if required.

Installation Config

Requirements

The installation config needs to provide a way of being able to replicate a systems installed software, it shouldn't rely on any software that may be installed at the Machine Infrastructure stage, this results in duplicate installation of some programs in some cases but most of the time the installation is skipped if it already exists. This concept allows for maximum portability, the ability to run on a container, a virtual machine or a bare metal host.

Another requirement is that allthough I'm setting up this system for my benefit I would like other users to be able to use the same base settings and achive a similar setup if they desired, this has the challenge of allowing me to have some accounts (such as public SSH keys shared with github using my private access token) set up but have other users skip these steps and still have a functional user experience.

Outcomes

I realised early on that one giant bash script to set up a system would be slow and that a lot of set up tasks could be run in parrallel. I also realised that to be as configurable as possible the configuration files should be seperated from the code so this lead to a program that runs scripts and a configuration repository that is used to tell the program what to run and when.

In this approach I developed a program that would read from a config file and launch scripts, to help with the speed of the process it uses multiple threads where possible and if there is no requirement from a script for IO then it will run in the background and print it's results upon completion. This achieves the speed we desire, but also allows the users software environment to be refreshed everytime the user reconnects to the machine/container.

The second part to this process was the coniguration files, I developed a basic scheme to be able to specify tasks in groups, then allow groups to be run once pre-requisite groups had been completed. I also saw the need for subgroups, tasks that are very closely related to one another, to be run sequentially one after the other.

As a matter of keeping the environments as fresh as possible across different hosts a third party software called gitwatch (see the readme for details) which automatically watches the config files and pushes them to github so at next login the environment is refreshed with the latest developments hosted on github. The idea is to be able to get up from one machine after exiting the programs running and start the system on another machine and be presented with exactly the same environment you finished with on the last machine.

The results of this work are known as wakahiki Te Reo Moari for crane as this program lifts up all the install scripts and deploys/runs them on the machine/container, also kainga-conf kainga meaing home in Te Reo Moari and this

being the config to setup our machine/container software environment to be the same as we expect everywhere we go.

Issues and Solutions

- **Package installs and multiple threads**
  One of the things for installing new Linux systems is the time it takes to install all the packages, I experimented with running these in parralell unsuccessfully.

  Solution
  The way the package manager locks resources is not easily polled, the lock file always exists so it's not enough to simply check if the file exists or not, because of this trying to tweak the package manager to run on multiple threads was unsuccessfull, packages exist that do this and could be a candidate for the future but for now I set the package install scripts to grab the IO semaphore in wkahiki so that they can only run as a single thread.

- **IO and blocking**
  Because I wanrted the installs to happen as fast as possible and IO slows down programs I wanted subporcesses that didin't need user input to run fully in the background and to only display their output once at the end of there runtime. The issue here was that some programs only required input if something went wrong, or to confirm an overwriting action in the second run. The issue here is that if a program in the background reuests IO and isn't connected to a TTY it will suspend the program, this allows the user to manually set the suspended program to run in the foreground in order to interact with it.

  Solution
  The solution here was to determine which subprocessess needed IO even if only on program errors and to set them with prompt=true in the config so that they would grab the IO semaphore and run attached to the TTY.

- **Private repos**
  Initially I had matapihi and wakhiki as private GH repos and I wanted to clone the repos at runtime so that I could develop on them immediatly when launching into a container during the dev process. I would be prompted for a username and password each time for each private repo

  Solution
  To be able to pull the repos without being prompted for a password each time I set up a script that would generate a SSH key for me and then upload that to github for me using my personal access token that it would prompt me for during the install, this meant one pass phrase and I could clone/pull all my private repos. I used the config in my dotfiles to set my global git config to only use SSH for github remote connections, however this script solution is what caused the following issue and solution process.

- **Github Keys and Stow and Git config**
  Still in progress, basically if you're not me you don't upload your SSH key to my github, but my github config says do all pulling and pushing over ssh so after the dotfiles are stowed you need an SSH key uploaded to github in order to be able to pull or push so this causes futher failures in the rest of the github clones/pulls specified in the config if you don't upload a key.

  Solution
  There is two fixes here, prompt for user on GH key upload so people can upload their own key to their own account if they have set up a github access token. If the user chooses to skip the key upload process then change the global git config to be HTTPS instead of SSH by default. This may have a downstream impact on gitwatch, as in they wont be able to automatically push, but they should fork the dotfiles and kainga-conf repos and use it as their own if they want to beable to do

that.

- **Idempotent Scripts**
  Re running matapihi initialisation would cause errors for some of the scripts causing wakahiki to prompt the user if they want to abort or continue. The rerun would happen anytime you exited the matapihi program and then reconnected the VNC client.

  Solution
  The solution was to go through and inspect all the scripts that wakahiki called as specified by the kainga-conf files and modify all the scripts to all be idempotent, in a lot of cases this was as simple as checking if a certain file existed or service was running and if not, create the file or start the service.

Limitations and Future Considerations

- **Git Submodules**
  I looked into using Git to house the scripts located in kainaga-conf/bin because some of those scripts are not only useful at install time but also at run time, I thought submodules would be a great way to have the bin scripts as a seperate module this proved to difficult to maintain automatically to a satisfactory level, so I reverted to just symlinking the bin files to the ~/bin/my-bin folder which has the advantage of not taking up the diskspace twice and the scripts always being exactly the same in one place and the other. In hindsight I should probably call GNU Stow on the bin files with different parameters.
- **AltTab bug**
  There is a bug in th euse of Alttab in relation to this kainga package, I don't think it's a bug with Alttab itself but just in how I use it in this setup, the bug is that when knew Xwindows are launched even though they maybe in the foreground they don't always get the keyboard input untill you alt tab to bring it to the foreground even though it visually is. Likely there is some Xserver or Alttab setting that forces the Xwindow on top to always have the focus.
- **Screen resolution auto update on viewer size change**
  With the ideal scenario of the Xserver onthe container updating its display resolution to match that of the VNC viwere client It would be good to make all Xwindows also be notified to change their dispaly to be the new fullsize after the resolution changes. I've implemented a simple resize tool named wnrz that uses some built in functionilities of the Xserver packages to set the resolution and windows to fullscreen in this new resolution so it would be a case of figuring out what even triggers occur when the resize happens and then running wnrz with the appropriate display size.

Runtime Config

Requirements

This requirement is similar to the Installation Config but isn't specific to software installation and whats being run on a computer but the users customisations of the softawre run on the computer, in the linux world these are mostly stored in dotfiles (files starting with '.' also know as hidden files). So the requirement is to be able to back all of these up and redeploy them with ease on new machines/containers and they shouldn't have any dependencies on the software installed. For example there may be a dotfile related to a piece of software, but if that software is not installed the dotfiles would be more like plain text files than config files, as in they would do nothing.

Outcomes

We needed two things from this Runtime Config, one is to set up the environment as

expected in a bash shell this is usually using the resource `.bashrc` which generally adds things to the path to allow a user to call executables to run. But also to store runtime config and provide versioning of the config for other installed software such as the `.gitignore_global` (a file that tells git what to ignore) and `.gitconfig` (a file that sets up user preferences for git in a session) more on how this is achieved is described in the Issues and Solutions for this section below.

The results of this work are known as dotfiles which is a GNU Linux concept of storing all your personal configuration items inside hidden files which in unix land start with a `.` hence the name dotfiles.

Issues and Solutions

- **Dotfiles**
  In the unix-like world dot files are stored in a users home directory and used to store local customisations for user preferences for all sorts of programs, the problem is that they are not stored in a standard way and in order to be able to have repeatable user experience we want to be able to store them in a git repository which also allows for verioning if a new configuration doesn't result in desired affect for example.

  Solution
  Fortunatley the GNU team are to our rescue again, with GNU Stow. Stow is a symlink farm manager. What this means is that you can set up a standard folder with a specific stow structure and it will move your dotfiles into this folder then create and symlink back to the original location for the programs to reference. The advantage here is that the dotfiles folder can be managed by git for versioning and pushed to a public repo on GitHub for replicating to any machine that has internet access.

- **Including files from the root of the stow folder**
  As part of the package setup I went through and added license file to all the repositories and relevant source files. But this lead to an interesting situtaion that conflicted with how GNU Stow works, it doen't expect there to be any files in the directory that stow is called from, only directories this caused anerror running the stow part of the config

  Solution
  Use the Unix supported concept of globs for directory wildcarding in the form of `*/` to skip any file thats not a folder in the dotfiles directory. It even was my first ever stack overflow answer and got marked as the accepted answer (at the time of writing)

- **Additions to the .bashrc**
  This is a dificult one because the system generates a .bashrc for the user and likely has some system specific configurations, and you want to add your own customisations but you don't want to forgo any of the system generated defaults.

  Solution
  I solved this by implementing a script which appends to the end of a file, it uses git to merge the files so that if the change is already in the .bashrc then it won't add it again, and it simply adds a call to check if the file .my-bashrc exists and then sources that file too, and so in the file .my-bashrc is where I put all my personal environmrnt settings.

- **Idempotently adding directories to the Path**
  Because the potential exists to source the .bashrc multiple times during a session, I didn't want it to be always appending to the end of the path, I was concerned about (and experienced) multiple additions of the same directory to the end of the path.

Solution
I implemented a script that I stored in my bash functions to only add to the path if it didn't already exist in the path, inspiration for the script came from here

Limitations and Future Considerations

- **Spacemacs Submodule**
  Currently I'm using the develop branch of Spacemacs, this branch allows the latest build but doesn't provide an easy way to update, you need to perform a git pull. I wanted to include it in my dotfiles for tracking of the changes and to be able to revert if I or upstream introduced a breaking change, this however had the unintended side affect of my dotfiles not tracking the child git module included in emacs.d, I circumnavigated this by deleting the .git folder for the emacs.d folder but this means that I can no longer use git pull in that folder to update spacemacs, I should instead set this up as a git submodule and add steps to pull the spacemacs when the VNC client connects also, for one of the other kainga-conf changes I investigated git submodules and decided they weren't very usefull in that situation, but this situation is what they are designed for and with some addition to the pull operation in the kainag-conf this would be quite easily achievable

Other Considerations

Security

While basic steps have been taken to achive a base level of security, encryption over the wire and passwords for sudo operations and logging in there is myriad of other potential hardeneing techniques that should be researched and applied for this to become a production level system.

Dependency on Debian

As it stands most of the scripts are set to run with a Debian flavoured linux, it would be good to add support for other Linux flavours, mainly red hat based images because of the use inside commercial enterprises, this could be achived most likely by adjusting the package scripts to determine what flavour we are on and then run the appropriate packaging install and updating command.

Other Use Cases

I see this setup as a proof of concept that you can use containers to run GUI applications inside of, I can imagine this having real potential for developers to write programs that can be used across different operating systems seemlessly without the user knowing that the program they are interacting with is actually running on a container.

Another novel idea is to run thie in a cloud environment and do development from local underresourced machines by connecting through to more powerful machines running in the cloud, this could also help companies with infrastructure costs by not having to maintain thin clients, dev machines and cloud machine by simply cutting out the dev machine in between the thin client and the cloud machines.

Conclusion

This set of programs developed for and collected together in the package kainga and the packages that they install solves the initial problem of being able to have the same configuration across different machines when it comes to my preferred developer environment. It is a proof of concept, and I believe it is a success, I will

continue to use it for me personally and would be happy to see others take it up as well. As it stands it’s not entirely polished but definatley useable, I’ll likely continue to tweak on it here and there as time allows in order to keep it meeting my needs, but would love for it to be picked up by the open source community and would be keen to support it in that space.