A Major Project Final Report on

Utility Software for Management of Self-hosted Web Services using Docker Containers

Submitted in Partial fulfillment of the requirements for

the degree of Bachelors of Engineering in Software Engineering

under Pokhara University

Submitted by:

Ayush Jha, 15707

Aakankshya Pradhanang, 15793

Dipendra Timalsina, 15754

Date:

08 / 08 / 2076

Department of Software Engineering

Nepal College of Information Technology

Balkumari, Lalitpur, Nepal.

Abstract

The project titled “Utility Software for Management of Self-hosted Web Services using Docker Containers” provides tools to install and maintain free and open-source self-hosted web services[1] on a Virtual Private Server. Web services include services for document hosting, image hosting, online-office software, web-based task management software, social networking services etc. These services will be packaged as docker images. The services will be orchestrated using docker-compose[2] files and Nginx[3] will be used as the gateway to access the services. Each service shall be accessed via a subdomain over https. The project will also include the development of a centralized repository of indices to web-services. DockerHub[4] will be used to store actual docker images for the services. An index of each service, including it’s name, description, DockerHub tag, will be stored in text files on a public server. Installation, removal, and maintenance of the web-services will be done via a command line utility which will be developed as part of the project.

Keywords: self-hosting, Docker, Nginx, services, repository, utility

Table of Contents

Table of Figures

Index of Tables

1) Introduction

With more and more of the world adopting multi-platform technologies, the world wide web has seen an emergence of open standards, protocols and frameworks that have shaped the software that we use in our daily lives. Enterprise software has seen an even bigger overhaul of technologies. From UI design patterns to development strategies, enterprise grade software have become bigger, and have gathered an even larger set of dependencies along the way. These dependencies make installation, maintenance and upgrade of mission-critical business software very expensive. Thus, security updates have slow adoption rates, vulnerabilities aren't patched and end users have to deal with outdated software.

Amidst the aforementioned problems, containerization technologies have emerged as a "one solution for all problems". One could be a skeptic when dealing with containerization, it's use of system resources, and it's promises of abstraction – most system administrators prefer to be as close to bare metal as possible. Yet, in an ongoing effort to streamline infrastructures, maximize server resources, and ensure that applications run smoothly and securely, different approaches have shaken up the traditional server-side architecture—things like the cloud and virtualization. More and more businesses are migrating applications away from traditional IT set-ups in the search of better performance, more efficiency, and more competitive operating costs.

At the end of the day, a business has only two ways to measure the "goodness" of a technology stack – efficiency and cost. Containerization technology wins on both fronts. So let's take a step back and look at containerization in depth.

The dictionary meaning of containerization is **:** a shipping method in which a large amount of material (such as merchandise) is packaged into large standardized containers. Software containerization closely resembles it's dictionary meaning. In software containerization, a set of services are placed into a contained environment with predefined and known exits. Inside the environment, the pieces of software may interact as they like, but interaction with anything outside the environment is heavily

controlled. This provides a way to manage side effects from the software inside the container. The benefit of using containers is the ability to control a module's interaction with the rest of the system. This comes at a price – resources. Since each component of the system needs it's own container, the amount of resources required to run a component increases. But with the rise of cheap resourceful servers, this problem has become unimportant. Another use of containers is dependency management. Let us assume that a VPS has an installation of Nextcloud for use by the development team. After a while, the management team also also want their own Nextcloud instance. If one tries to install two instances of Nextcloud on the same server, dependency conflicts will cause unexpected crashes, unwanted bugs and slow down the progress of both development and management teams. There are two solutions to this problem, the first solution is to get two different servers for the two team. This is very wasteful because servers are expensive. A cheaper and faster solution would be to use containerization technology, such as Docker, to provide containerized instances of Nextcloud to each team. The ports for accessing each instance can be managed, and a reverse proxy can be used to pass traffic to their respective ports.

The true power of containerization is displayed when used in conjuncture with micro-services, or self-hosted solutions. In either case, instances can be dynamically managed. New instanced can be spawned based on rules depending on resource usage, traffic etc. This is a valuable utility, which – if automated, can provide a clean and simple way to manage complex systems such as file hosting software, git repository management software, project management software etc. With all these software running on the same servers, dependency clashes are inevitable. Thus, with the use of containerization technology, these services can be managed in a clean fashion without the need for extensive packaging, deployment of pseudo-isolation techniques, or access to heavy duty servers.

Self-hosted services were mentioned in the previous paragraph. Let's dive into the term in depth. Self-hosted is a FOSS jargon which implies hosting a piece of software on one's own hardware. While the specifics of the term "self-hosted" is up for debate,

the most widely accepted definition of self hosted service is : any service, software system which is hosted on the user's own hardware. I take a more liberal approach to defining self-hosted services : self hosted services are a category of software which are installed by users on bare-metal/virtual hardware which they have authority over. I shall be using this terminology of self-hosted services for the span of this report.

The third and final part of the project which the reader may need an introduction to would be the term "utility software". With the rising trend of agnostic nomenclature in the software field, most all technology can be categorized as either an app or a library/ framework. Thus, I believe that a proper definition of "utility software" is a necessary understanding whilst progressing with this report.

Utility software is system software designed to help to analyze, configure, optimize or maintain a computer[5]. In simple terms, any utility software is a helper that automates, redirects, formats, configures and/or optimizes computer system/s. A very popular utility software is apt-get, a package manager which automates the process of downloading and installing packages with dpkg. But a utility software may not be limited by the scope of the aforementioned definition. It may provide additional features, such as reporting mechanisms, debugging tools, manual generation tools etc. As long as a piece of software helps the user make better utilization of existing system/s, said software is a utility software.

While the above explanations correlating to the different part of this report are focused at beginners, the reader is encouraged to further research these terms for a much clearer understanding of containerization technologies. The bibliography at the end of this report is a good resource to help the reader get started.

I) Problem Statement

This project deals with privacy issues tied to using hosted services, such as DropBox, Google Drive etc. While third party hosted services are convenient to use, most of these service providers do not respect their user's privacy. Thus, self-hosted services provide a privacy-friendly alternative. Concern over lack of privacy by corporate service providers is growing, and while attempts have been made to create alternatives, they haven't been successful since installation and maintenance of self-hosting web-services[6] is complicated and time-consuming.

With the emergence of free and open-source mobile operating systems like librem 5, Eelo, etc, there is a growing market for self-hosted service providers, yet there aren't many projects that provide easy-to-use utilities to install self-hosted services on private servers.

The problem with self-hosting web-services is the complexity of setting up reverse proxies, managing certificates, debugging bugs that occur when multiple services get installed on the same machine.

This project provides utilities to ease the installation and maintenance of web-services, so that even novices can setup these services on their own servers.

II) Project Objectives

The objectives of the project “Utility Software for Management of Self-hosted Web Services using Docker Containers” are as follows

- Provide a standard way to install self-hosted services via use of Docker images.
- Create a repository that hosts indices to updated versions of docker images for web-services[7]
- Manage Nginx server[8] configurations to provide custom subdomains for each service
- Provide command-line utilities to search, install, maintain, remove services

III) Significance of the Study

The system will provide a set of easy to use utilities to search, install, maintain, remove, update web-services[9] from a central repository. The repository will include meta-information about the web-services. This project will provide the following benefits to anyone self-hosting web-services:

- Ease of use in installing and managing services
- Easily search for web-services to install
- Easily setup HTTPS server to access the services.
- Use of docker- compose[10] for better control over service resources and access to host machine.

2) Scope and Limitations

The scope of this project is to create a Utility Software for Management of Self-hosted Web Services using Docker Containers. With this system user will be able to -

1. Easily Install and manage web-services[11] such as nextcloud, jitsi, etc on a private server.
2. Manage these services via a CLI utility.
3. Use a centralized repository for the web-services for easy installation and removal.

We will be creating a single service and storing it on a repository. The repository for the project will be hosted on Gitlab, with the docker images being hosted on DockerHub[12]. The CLI will also be a minimal implementation, and we will not focus on speed, efficiency or reliability of the utility.

3) Literature Review

While performing requirement analysis for this project, we looked into a set of technologies that almost meet all our requirements. These software provide the building block for our project, and thus need to be discussed. Even though some of the discussed software weren't used in this project, they provided useful guidelines, concepts and design understanding which guided us towards a better system, and helped us form a clear picture of the final outcome. These systems also gave us a perspective into the practical possibilities, feasibility and current trends regarding container based solutions.

While installing separate web-services[13] is easy, most web-services need to be modified to let it coexist with other services, since they use the same port number, same file system and many times, same DBMS.

Currently, there are a few ways to setup multiple web-services on the same private server. We have made a list of software which we researched while in the requirement analysis and design phases of this project.

I) Snap / Flatpak package managers

Snap[14] and Flatpak[15] are containerized package managers[16], in the sense that all the dependencies for the package are bundled into a single chroot environment and access to anything outside the environment is extensively controlled.

Both Snap and Flatpak provide a sandbox environment in which the software can perform all the tasks it wants. A huge benefit of sandbox environments is it's ability to provide different dependency versions for different software. Let us assume that there are two software, S-1 and S-2 on a system. Let both these software be dependent on a dependency D. If S-1 is dependent on version 1.0 of the dependency D, and S-2 is dependent on version 2.0 of dependency D – both these software cannot be installed on the same software without manual configuration. In a sandbox environment, S-1 and S-2 can exist on the same system, but since they are in their own respective sandboxes, they can have different versions of the dependency without any conflict at all.,

Web-services can be easily installed, mostly with a single command, using this method, but the management of web-services is still tedious and this method adds new complexities to the management process, such as accessing files outside it's container.

II) Traditional package Managers (apt-get, pacman, yum, dnf etc.)

Traditional package managers assume the existence on one system with shared dependencies. This makes package management much easier, but it also disregards complexities. Traditional package managers are on almost all Linux distributions, so their availability is not an issue, but their lack of containerization or sand boxing is the main reason we chose to not use them in our project.

While the use of traditional package managers reduces complexities, it also makes updating services less reliable and causes dependency issues when installing multiple web-services at once. This method also has the port-conflict problem between web-services.

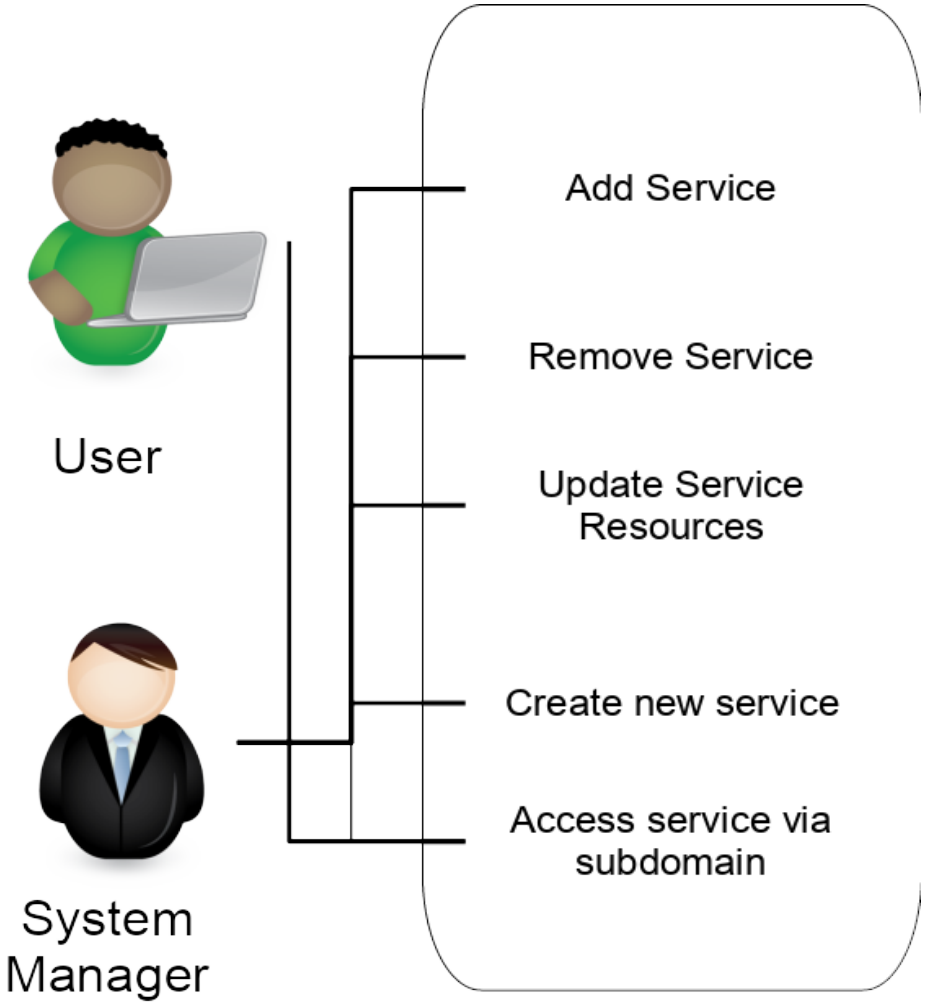

Figure 1: Usecase Diagram

III) Docker Containers with Docker-Compose for Orchestration

Docker is a protocol and software of the same name. Docker provides a thin base image, over which any Linux distro can be installed. A docker image is a set of stacked operations which provide the required environment to run a specific software. For example, a docker image based on a debian image, with openjdk installed is

suitable for running java software. The software may be accessed via controlled sharing of the filesystem, or via port mappings between the host machine and the docker contianer.

Docker-Compose is a python based utility software built on top of docker to automate the creation of docker services, networks, volumes. Docker-Compose also provides mechanisms to start, stop and log specific services. This makes docker-compose an invaluable tool to orchestrate Docker contianers. Docker-Compose uses YAML configuration files, and aims to be easy to configure and maintain. While Docker uses it's own configuration language, Dockerfile – docker-compose uses a widely known human readable configuration language, YAML.

This method is the closest to the method we are using for this project. While docker containers are easy-to-use and manageable, and ports can be mapped between host machine and container, finding good docker containers is still hard. The user will either have to create their own docker container or download it from DockerHub[17]. Also, the user will have to manually write the Dockerfile and Docker-Compose[18] for container orchestration. The user will also have to configure the reverse-proxy for each new service.

Thus, when evaluating other products, our project, Utility Software for Management of Self-hosted Web Services using Docker Containers will make it easier for novice users to setup their own home server with as many web-services as they want. Our project will also make it easy to configure, manage and remove individual services as per the need.

4) Methodology analysis and validation scheme

As we follow AUP along with TDD for the development of the software, validation and performance analysis is inseparable part of the methodology. Test phase of AUP ensures that the software is valid and meets the requirement. Similarly, performance is also measured by these integration tests as well as unit tests.

Agile Unified Process

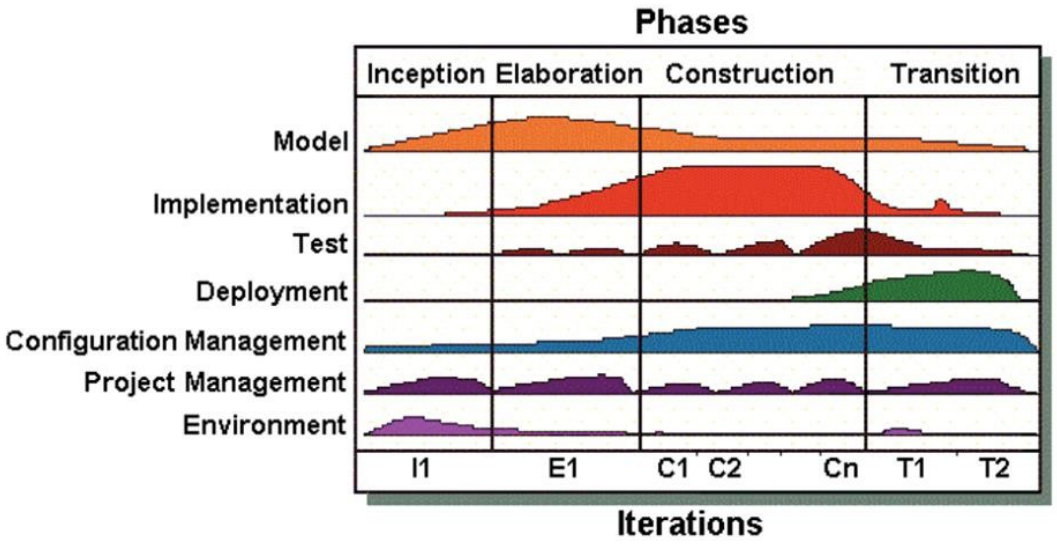

Figure 2: Agile Unified process [0]

AUP has following phases:

I) Model

We discuss the business of the project with the college representative, and analyze the requirements. Then, we discuss the viable solution with the team.

II) Implementation

We shall follow TDD approach for implementation. We write unit tests for a requirement. Then add production code to make the unit test pass. We will repeat this step until the requirements are met.

III) Test

As unit tests are written in implementation phase, we also add some integration tests. These tests will validate if all the units work when integrated together.

IV) Deployment

We will be using the CI/CD provided by Gitlab to make sure about the frequent

releases.

V) Configuration Management

Each release will have different versions, and we keep track of each of them.

VI) Project Management

We will have regular meeting with team members. This helps to know about the progress and status of the project.

VII) Environment

In our regular meeting with team members, any required resources are discussed and made available as needed.

5) Tools and technologies

Since this project focuses on the creation of a utility software, a variation of scripting languages need to be used to provide the adequate automation and configuration necessary to complete it's requirements. On that end, we must provide breifs on the languages used, design patterns chosen, orchestration tools used, as well as discuss the techniques used to perform secondary tasks such as backup management, request handling etc. The following chapter discusses the mentioned topics in detail.

I) Languages Used

Since the utility is developed to run on Linux servers, bash scripting is the preferred scripting language. This decision was based on a few reasons. Foremost, Bash is ubiquitous. It is the defacto shell for almost all Linux distro, especially all the "popular" linux distros like Ubuntu, Arch, Debian, RedHat etc. The other reason the ability to manipulate files with ease. Bash provides the foundation to perform fast and complex modifications to files on the fly. This is a must when automating any system.

Along with bash, Awk scripts have been used (mostly inline) to perform line level modifications. Use of awk is at a minimum, but this outcome is not intentional and porting parts of the project to Awk is in our future plans.

A large part of the project deals with Docker and Docker-Compose, thus YAML scripts encompass a large portion of the project. Docker-Compose is scripted in YAML, and since each service must have a docker-compose.yml file – YAML is also used in this project.

II) Artifact Management

The term "artifact" is used for the output produced by Gitlab CI/CD. As of present, CI/CD has not been deployed for this project, but future plans exist to make use of CI/CD to produce docker images and store them in the Gitlab registry.

III) Design Patterns

Looking at the current state of the project, no design patterns were strictly followed. The project is currently a bodged together piece of spaghetti code consisting mainly of an ad-hoc template system which generates scripts that are tied together via bash scripts and Nginx standard directories. Plans for a better design are planned for future versions, but that is highly unlikely at the moment, because further work on the project will probably cease.

IV) Orchestration

Orchestration is at the core of this project. Orchestration is used to manage a set of containers that form a service, and a set of services that form a client. Currently, orchestration is done via docker-compose, since all the containers are contained inside a single VPS. If multi-VPS environments need to be supported, then there are two options – Docker Swarm, Kubernetes.

V) Keeping Backups

Simply starting a service has no value if there is no persistency. This is where backups come into play. There are two kinds of backups – contianer backups and volume backups. Container backups are done by Docker, in which docker takes a snapshot of a container and places the blob into a zip file. This file can be moved across servers and run anywhere as a container. The other form of backups is volume backup. In this form of backup, only the data is backed up. A set of directories from inside the container for required service are mapped to the host machine. These directories on the host machine are then periodically backed up.

VI) Handling HTTP/S Requests

This project uses Nginx as a reverse proxy to our services. A subdomain is automatically selected based on the client and service name. For example If a client, Ram, is running a service, Odoo, then a domain name odoo.ram.siyasang.com will be

generated and placed into the Nginx configuration along with appropriate reverse proxy configuration. This process is automated by by this project. The Nginx server itself is run inside a docker container, and can be reloaded via a Makefile in the root directory of the project.

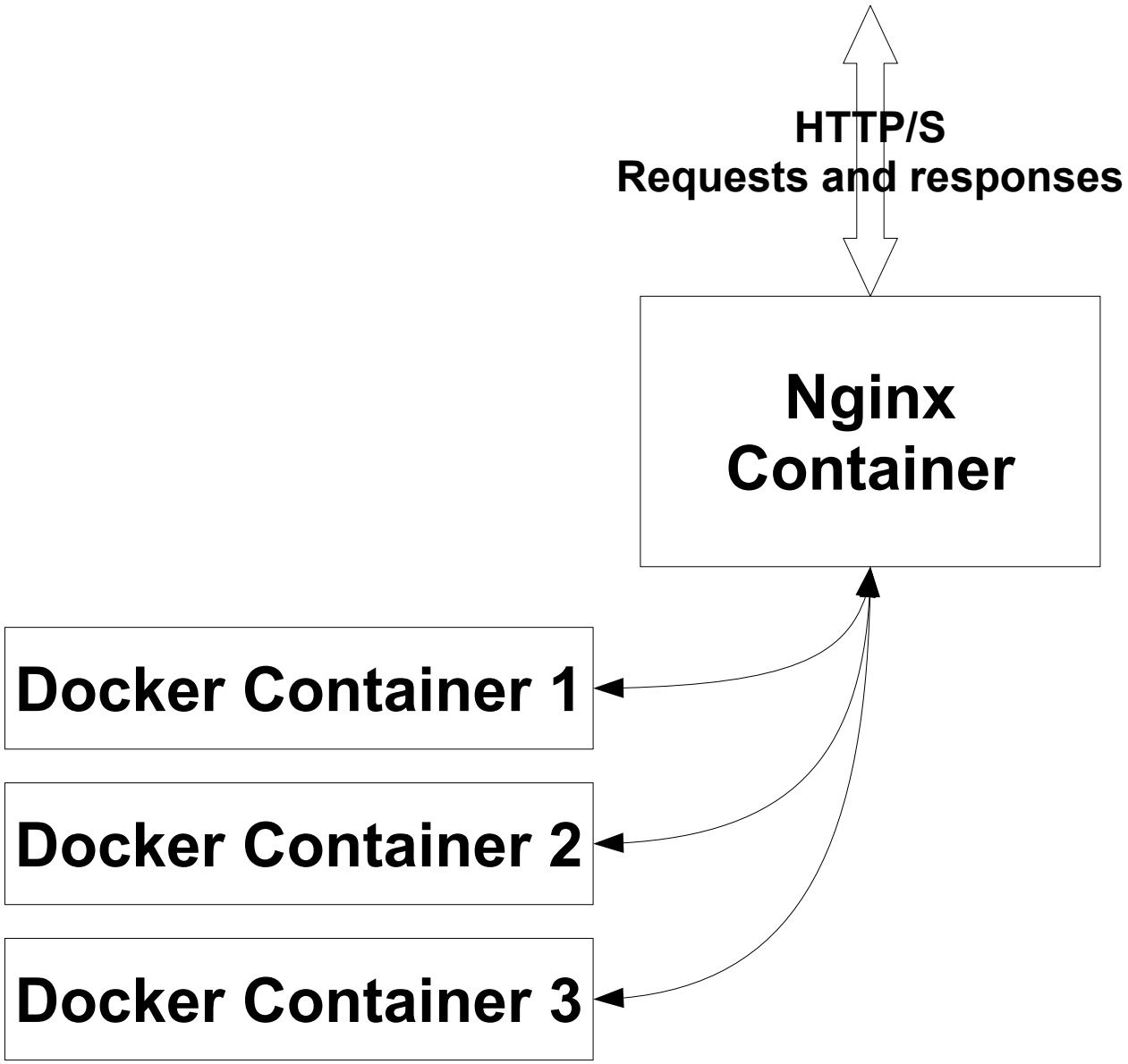

Figure 3: Handling HTTP/S Requests and Responses

VII) Additional Tools and technologies

Certbot is used for SSL certification from LetsEncrypt[19]. LetsEncrypt is a certification system which provides free SSL certificates to websites. It also supports wildcard subdomains. Certbot is written in python and can be downloaded from the Python repository.

Similarly, Gitlab is used for git repository management. This choice was made due to gitlab's CI/CD and docker image registry features.

6) Requirements

Upon looking into the various tools specified in the literature review, we made a list of requirements for our product.

- One user may have multiple services running for them
- One user can access the services via a subdomain
- All the services must be containerized, along with their dependencies – such as DBMS, In-Memory databases such as Redis, FileSystems etc.
- The resources allocated to each service must be configurable. Resources include CPU time, memory and disk space.

7) Usecase Diagram

Use Case Name : Add Service

Actors : System Manager, Client

Preconditions:

- Manager has access to the requires servers
- Client has ordered a new service
- Required service can be created for the client

Post - conditions : New service will be created for the client

Normal flow :

1. Client requests a new service
2. Specifics of the service are noted by the system manager
3. System manager creates a service with the given specifics

Alternate flow :

1. If the service is not available, the client's request is rejected
2. If the service is already running for the client, a conformation for a duplicate service is sent to the client before proceeding ahead.

Use Case Name : Remove Service

Actors : System Manager, Client

Preconditions:

- Manager has access to the requires servers
- Client has ordered removal of service
- Required service can be removed for the client

Post - conditions : Service will be removed from the system

Normal flow :

1. Client requests removal of service
2. Manager backs up current state of the service
3. Manager stop the docker service
4. Manager removes the nginx configuration for said service

Alternate flow :

1. If the service is not available, the client's request is rejected

2. If the service is already removed, the task is aborted.

Use Case Name : Update Service

Actors : System Manager, Client

Preconditions:

- Manager has access to the requires servers
- Client has ordered an update of service
- Required service can be updated for the client

Post - conditions : Service will be updated

Normal flow :

1. Client requests update of service
2. Manager backs up current state of the service
3. Manager stop the docker service
4. Manager updates the resources, or configuration files for the services
5. Manager restarts the docker service

Alternate flow :

1. If the service is not available, the client's request is rejected

8) Conclusion and future

The current version of the project is a proof of concept that accomplishes the following goals.

- One user may have multiple services running for them
- One user can access the services via a subdomain
- All the services must be containerized, along with their dependencies – such as DBMS, In-Memory databases such as Redis, FileSystems etc.
- The resources allocated to each service must be configurable. Resources include CPU time, memory and disk space.

The following features are not available yet.

- GUI for management of services
- Automatic creation of subdomains

9) Project task and time schedule

Task Schedule

Table 1: Project Task and time schedule

Model	Ayush Jha, Aakankshya Pradhanang, , Dipendra Timilsina	
Implementation	Repository	Ayush Jha
	CLI Utility	Ayush Jha
Test	Aakankshya Pradhanang	
Deployment	Ayush Jha, Aakankshya Pradhanang, Dipendra Timilsina	
Configuration Management	Ayush Jha, Aakankshya Pradhanang, Dipendra Timilsina	
Project Management	Ayush Jha, Aakankshya Pradhanang, Dipendra Timilsina	
Environment	Ayush Jha, Aakankshya Pradhanang, Dipendra Timilsina	

10) Bibliography

1: David Booth, W3C Fellow / Hewlett-Packard Hugo Haas, W3C Francis McCabe, Fujitsu Labs of America Eric Newcomer (until October 2003), Iona Michael Champion (until March 2003), Software AG Chris Ferris (until March 2003), IBM David Orchard (until March 2003), BEA Systems, Web Services Architecture, ; 2017

2: Docker Contributors, Overview of Docker Compose, https://docs.docker.com/compose/; Ongoing

3: Nginx contributors, About Nginx, https://nginx.org/en/; Ongoing

4: Docker Contributors, Docker Hub Quickstart, https://docs.docker.com/docker-hub/; Ongoing

5: Parsons, June Jamrich; Oja, Dan, New Perspectives on Computer Concepts 2014: Comprehensive ,

6: David Booth, W3C Fellow / Hewlett-Packard Hugo Haas, W3C Francis McCabe, Fujitsu Labs of America Eric Newcomer (until October 2003), Iona Michael Champion (until March 2003), Software AG Chris Ferris (until March 2003), IBM David Orchard (until March 2003), BEA Systems, Web Services Architecture, ; 2017

7: David Booth, W3C Fellow / Hewlett-Packard Hugo Haas, W3C Francis McCabe, Fujitsu Labs of America Eric Newcomer (until October 2003), Iona Michael Champion (until March 2003), Software AG Chris Ferris (until March 2003), IBM David Orchard (until March 2003), BEA Systems, Web Services Architecture, ; 2017

8: Nginx contributors, About Nginx, https://nginx.org/en/; Ongoing

9: David Booth, W3C Fellow / Hewlett-Packard Hugo Haas, W3C Francis McCabe, Fujitsu Labs of America Eric Newcomer (until October 2003), Iona Michael Champion (until March 2003), Software AG Chris Ferris (until March 2003), IBM David Orchard (until March 2003), BEA Systems, Web Services Architecture, ; 2017

10: Docker Contributors, Overview of Docker Compose, https://docs.docker.com/compose/; Ongoing

11: David Booth, W3C Fellow / Hewlett-Packard Hugo Haas, W3C Francis McCabe, Fujitsu Labs of America Eric Newcomer (until October 2003), Iona Michael Champion (until March 2003), Software AG Chris Ferris (until March 2003), IBM David Orchard (until March 2003), BEA Systems, Web Services Architecture, ; 2017

12: Docker Contributors, Docker Hub Quickstart, https://docs.docker.com/docker-hub/; Ongoing

13: David Booth, W3C Fellow / Hewlett-Packard Hugo Haas, W3C Francis McCabe, Fujitsu Labs of America Eric Newcomer (until October 2003), Iona Michael Champion (until March 2003), Software AG Chris Ferris (until March 2003), IBM David Orchard (until March 2003), BEA Systems, Web Services Architecture, ; 2017

14: Snap Contributors, Basic Snap Usage, https://tutorials.ubuntu.com/tutorial/basic-snap-usage#0; Ongoing

15: Flatpak Contributors, Flatpak Reference, http://docs.flatpak.org/en/latest/reference.html; Ongoing

16: , Flatpak: A Containerized Approach to Developing Linux Applications, https://www.linux.com/news/flatpak-containerized-approach-developing-linux-applications/; 2016

17: Docker Contributors, Docker Hub Quickstart, https://docs.docker.com/docker-hub/; Ongoing

18: Docker Contributors, Overview of Docker Compose, https://docs.docker.com/compose/; Ongoing

0: , The Agile Unified Process (AUP), www.ambysoft.com/unifiedprocess/agileUP.html; 2006

19: LetsEncrypt Team, LetsEncrypt, Ongoing, https://letsencrypt.org