



Full-stack project containerization

Geoffrey Thielman

Haaga-Helia University of Applied Sciences

Bachelor's Thesis

2022

Thesis report

Abstract

Author

Geoffrey Thielman

Degree

Bachelor of Business Administration

Report/thesis title

Full-stack project containerization

Number of pages and appendix pages

40 + 9

Software development has undergone an overhaul when it comes to the testing and delivering software products. Hosting software for business related purposes upon virtual machines has grown to be an encumbering process with significant overhead when it comes to managing many virtual machines. The industry has shifted towards utilizing containerization technologies to solve this overhead and to leverage consistency of their products.

The author explores the concepts and role of virtual machines within its former traditional hosting role for software compared to that of containers, its successor. He suggests that containers add value to traditional virtual machines hosting with added security and increased abstraction while creating consolidating physical resources to the actual software solution rather than the operating system and overhead processes.

The introduction of a product thesis covered that of a testing tool in development that already has been taken into use with a growing pain of building installers as the end solution for the testing and delivering the product to the end-users. Slow builds have been hampering testing phases significantly due to wait times for the installer to be built. The solution previously being prone to environmental errors caused by installations in unsupported operating system by end-users which has been taking time away from developers to resolve these.

The thesis goals have been set by the author to minimize these problems by containerizing the full-stack project that makes up the testing project, to automatize its building process and to publish to an Artifactory storage. The sub-goal being to reduce build times by at least 40 % compared to its previous build solution.

Overcoming implementation difficulties and having theorized multiple methods to optimize build times with various tools, the author successfully containerized the project into isolated containers for each component. Having containerized the solutions he constructed the continuous deployment pipelines required to accommodate the need for automation.

The author concludes the project in an overall success while displaying minimum time saves of 55 % and describes direct and indirect benefits of the containerization for the future of the test tool's development.

Keywords

Containerization, DevOps, Full-Stack, Docker, Automation

Table of contents

1	Introduction	1
1.1	Problem setting.....	2
1.2	Project scope.....	3
2	Virtual machines and containers	4
2.1	Hypervisor Virtualization	4
2.1.1	Type 1 Hypervisor	5
2.1.2	Type 2 Hypervisor	5
2.2	Definition of containers	6
2.3	The big picture of containers.....	7
2.4	Docker	8
2.4.1	Dockerfiles.....	8
2.4.2	Docker Image	9
2.4.3	Docker tools for optimization.....	9
3	Project details	10
3.1	Robot Framework	10
3.2	Jenkins	10
3.3	Artifactory	11
3.4	Full-stack project description	11
3.5	Problem statement	12
3.6	Possible benefits from containerization.....	13
3.7	Project objectives	14
3.8	Implementation plan	14
3.8.1	Containerization implementation plan	15
3.8.2	Pipeline implementation plan	15
4	Implementation work	17
4.1	Database containerization process	17
4.1.1	Problems occurred	18
4.1.2	Implementation changes.....	18
4.1.3	Implementation result	19
4.2	Executor containerization process	20
4.2.1	Problems occurred	21
4.2.2	Implementation result process	21
4.3	Webserver containerization process	22

	2
4.3.1 Problems occurred	23
4.3.2 Implementation results.....	24
4.4 Pipelines construction process	26
4.4.1 GitLab Runner creation.....	26
4.4.2 Pipelines implementation.....	27
4.5 Storing container images	29
4.6 Hosting the containerized project.....	30
5 Evaluation	33
5.1 State as is with Jenkins	33
5.2 Full automatization with GitLab and Docker.....	35
5.3 Build time comparison	36
5.4 Containers brings complexity.....	37
5.5 Containers require extra security	37
6 Conclusion	39
6.1 Next steps	40
References	41
Appendices.....	47
Appendix 1. Resulting database Dockerfile	47
Appendix 2. Resulting executor 2-stage Dockerfile	48
Appendix 3. Resulting webserver 4-stage Dockerfile.....	49
Appendix 4. Resulting executor pipeline job.....	51
Appendix 5. Resulting webserver and database pipeline job.....	53
Appendix 6. Jenkins build trend.....	55

1 Introduction

Software development is in an ever growing process that continues to rise significantly with the need to fulfill a higher demand (Carey 2022). With software development comes the software development life cycle which contains phases such as planning, analyzing, designing, developing, testing, deployment, and maintenance. Two of the key phases being testing and deployment where we have to assure the software is working as intended and delivering the working solution (Demchenko 2021). Testing software can be done by implementing software tests and to have them executed in an automated fashion.

Software for intended for distribution is often packaged to an installer which then can be made publicly available. When software is distributed in this way the installation has to be tested upon multiple operating systems to ensure it works as expected. One of the biggest challenges is this assurance that the solution works well within the environment in which end-users will house the solution. The finished products of software projects are typically providing their services hosted upon virtualized software when it comes to software intended for use within business practices. This virtualized software being virtual machines which now long has replaced traditional dedicated hardware with a more elegant software-based architecture. Virtualization technologies has made the development lifecycle easier, especially when it comes to hosting multiple operating systems to install software upon for testing purposes. It allows you to deploy multiple operating systems to meet demands in a timely fashion (Carklin 2021). However, traditional virtual machines still requires a significant amount of overhead. An operating system has to be provided in the virtual machine before the software can be installed. Furthermore, the operating system itself requires some resources in order to operate functionally, this being CPU power, RAM and disk size. This combined with other software present on the operating system can take up a hefty amount of resources that could otherwise be allocated directly to the resource pool available for the running of our to be tested or hosted software (Simic 2019a). Having multiple virtual hosts running upon the same hardware can impact performance significantly. Furthermore, providing software as a service does not guarantee that end-users will host the software on a suitable environment that is recommended by the developers. This in turn can cause environmental problems with certain feature or result in a total failure to run the software. Building installers and testing them upon multiple environments is also a time costly procedure which limits developments time significantly.

Containers alleviates these problems by being able to provide a guaranteed consistent environment. Containers unlike traditional virtual machines don't add another operating system upon the host but rather leverages the host's operating system to provide an

isolated environment for execution. These containers can be optimized to reduced size by giving the option to only provide and install the bare essentials within the isolated environment. Furthermore, distribution of the software becomes easier as a total software package can be separated into multiple containers and the update process can be targeted to a specific container instead of having to update the entire solution everytime (Powell 2021). Containerization brings many advantages over traditional virtual machines and their implementation makes both development and delivery. These being the ability to easily orchestrate your solution with multiple container and limiting the amount of overhead needed. The portability of containers adds to the ease of usage, especially with its guaranteed working environment (Peltokorpi 2021, 13).

Currently within our company we are developing an advanced Test Tool. This project aims to increase testing flexibility and consolidate testing tools under one software. With a distributed system approach the project requires to be able to be deployed easily and run in most environments. Containerization of the project has been planned work as one of the stepping stones towards the productization of the project. Containers could possibly bring benefits within the development process. But due to late introduction of container virtualization within the project there could be some difficulty.

1.1 Problem setting

As indicated in the previous section, traditional virtual machines bring unnecessary amounts of overhead to make an environment ready for the project's testing and/or hosting. This also makes it so that we must package our software to be able to be directly installed by the end-users. Even stating which operating systems we support does not mean that the end-users will adhere to our advice. Furthermore, a lot of time in our development life cycle is wasted by testing upon multiple operating systems and resolving all the errors which occur in these different environments and end-users' environments. For every error that would be found and solved during our sprint's testing phase, the software package will have to be rebuilt, reinstalled and re-tested. This process interrupts the flow of development and testing outside the already long packaging times of the software.

The thesis aims to research and measure the impact of containerization as our method of building and providing the solution to end-users by packing our solution within containers. Also, this thesis aims to find out how our development workflow can be optimized by decreasing test, build, distribution and deployment time through the process of introducing containerization and automation. An additional goal is to find out what changes are to be

implemented to the project to accompany this structural change as container technologies are being introduced in an already long running project which is now near its production phase.

1.2 Project scope

The commissioned project thesis focusses on our company's full-stack project, and all its components needed to make the application work. Furthermore, the automation of its building process and method of distribution is covered in the practical part of the thesis. The thesis mainly focuses on Linux-based containers due to the limitation of windows containers due to them only being developable and runnable on windows host systems (Microsoft 2021) which requires some licensing which falls outside the scope and time frame of the thesis. Extra steps must be taken to be able to host windows containers together with Linux-based containers on the same platform (Haakman s.a.). The thesis covers the research, planning and implementation of the project's containerization. Container virtualization is researched by literature review before its technology is implemented by a proof-of-concept. This includes a theoretical implementation plan and practical implementation. The work to make the solution work within containers is documented together with all the rework required to project itself. The success of the project will be measured directly by the increase in performance, this specifically being faster build times compared to the current solution and the throughput of the automatization process consuming the build times.

2 Virtual machines and containers

IBM's first iteration of virtualization came originally from the 1950s when large-scale mainframes were being made available to schools and companies. These mainframes were widely expensive. To cut costs where possible it became vast practice for users to have access to the same data storage and compute power from any station. However, IBM released the first operating system called "VM" that allowed system administrators to have multiple virtual machines upon their System/370 mainframe (IBM 2017). It wasn't until around the 2000s when virtualization started to really take off for x86 architecture CPU's. This is when virtualization software like VMware started to release their first versions which set the trend for modern virtualization. In the early days this was mainly achieved through various software techniques. Modern-day virtualization techniques require specific hardware that supports it, this being mainly CPU's and motherboards which comes with the BIOS firmware feature to enable virtualization. (Agesen 2009). With virtualization we essentially programmatically emulate an entire computer system on top of a physical computer. Here we have a guest which is the virtual machine on top of the host or the physical computer.

There are many benefits as to why one would use virtual machines as a solution instead of multiple physical hosts. The most predominant being the cost price for the solution of virtual machines being much lower as you can host multiple virtual machines upon a single host. You can combine and allocate your compute resources to your virtual machines dynamically to meet any type of demand. With only running physical hosts you are at risk for investing heavily in wasted compute power when the demand is lower than your physical compute power. This makes virtual machines a more ideal solution for providing hosting services for various needs. The ease to manage virtual machines is the most alluring feature about the technology as it is much easier to manage than a physical server as backing up, restoring, booting, and scaling can all be done through the management tools provided by the virtualization tools (Jayaraman & Rayapudi 2012, 12-14).

2.1 Hypervisor Virtualization

Hypervisor-based virtualization is the current method of virtualization. Hypervisor-based virtualization brings an intermediate layer that is created between the virtual machine layer and the host operating system layer called the hypervisor. A hypervisor is software that manages and monitors virtual machines. This software creates the opportunity to run multiple virtual machines within a single host because it does not virtualize the physical hardware directly, but instead abstracts the computer's hardware with software. Here the

hypervisor software translates requests between the physical and virtual resources (VMware, s.a.). There are 2 types of hypervisor virtualization these being Type 1 Hypervisor or “bare-metal hypervisor” and Type 2 Hypervisor or “hosted hypervisor”.

2.1.1 Type 1 Hypervisor

Type 1 Hypervisor or the bare-metal hypervisor runs directly on the physical hardware, this meaning that there is no intermediate host operating system between the hypervisor and the physical hardware. This is the most used type of hypervisor due to its efficiency when providing requests from the virtual machine to the physical hardware as it does not need to go through any host operating system. Running directly on the physical hardware also comes with the benefit that more virtual machines can be created as no resources have to be used for any host operating system. Due to these benefits the Type 1 Hypervisor seem to be also the most secure as any vulnerabilities from a host operating system are not applicable to this solution. (Singh & Singh 2018, 19)

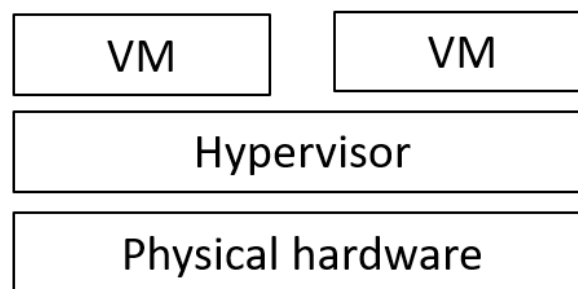


Figure 1. Type 1 Hypervisor Diagram (Adapted from Simic 2019b)

2.1.2 Type 2 Hypervisor

Type 2 Hypervisor or the hosted hypervisor is installed within the hosts operating system, hence the naming being “hosted”. This type of hypervisor is less popular within company workspace due to it being less efficient than the Type 1 Hypervisor as requests from the virtual machine must go through the hosted operating system before reaching. This type of hypervisor is more in line with how past x86 virtualization as they were a software layer within the operating system. The fact that the hypervisor is working within the host instead of directly on the physical hardware means that this type of virtualization does support a wider range of hardware. (Singh & Singh 2018, 20)

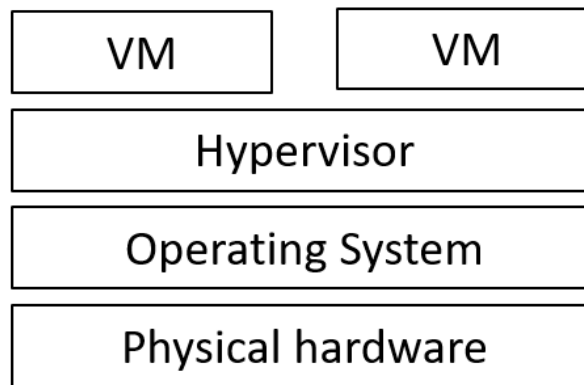


Figure 2. Type 2 Hypervisor Diagram (Adapted from Simic 2019b)

2.2 Definition of containers

A different method of virtualization is achieved with containers. Instead of virtualizing an entire virtual machine, containers virtualize an isolated environment which does not directly virtualize the physical hardware of its host, but instead virtualizes the operating system. Containers in turn share the host's operating system's kernel and does thus not require an operating system to be applied within the virtualized environment. This in term makes them more portable and efficient than traditional virtual machines. Unlike virtual machines whose goal is to provide a complete virtualized operating system solution, containers have as their goal to package code, software, and all their dependencies so that the applications can run within their own isolated environment. This is achieved by an intermediate layer that converts container images to containers at runtime, this being the engine (Docker s.a.).

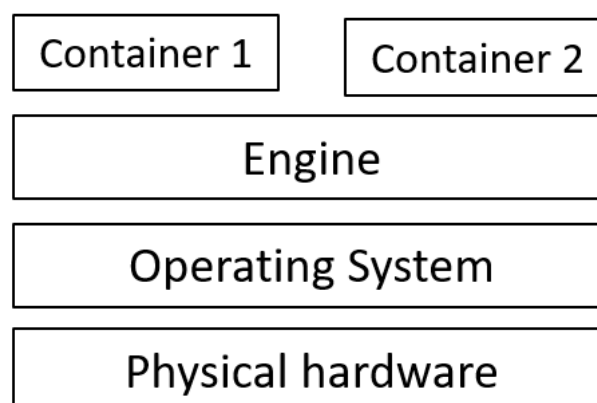


Figure 3. Container Virtualization (Adapted from Docker s.a.)

Virtualizing the environment and leveraging the host OS kernel gives containers a lot of flexibility as they can run within any supported OS as long as the container software has

been installed. This also gives the benefit that the environment within the containers does not interact with the host. It is completely isolated which allows the same containers to run on any operating system while adding an extra layer of security.

2.3 The big picture of containers

Containers provide many benefits to the software development lifecycle and has mostly replaced traditional virtualization in software development and hosting due to its ease of use. However, it is important to notice that VM's are not phased out of the industry. As a matter of fact, they are still holding their excellent usefulness outside of development and are still used as a hosting platform for various services. Containers and VM's work well together for their combined security. Especially when containers opposed to VM's are more so streamlined to enhance security controls directly (Guo 2017). While VM's add a layer of security by isolating the physical resources from the host and containers, in the case of Type 1 Hypervisors the security is more appropriate due to not having host OS vulnerabilities as there is no presence of a host OS. As Eder Michael describes (2015, 6) both technologies can be combined to improve overall security. However, this is not out of the box security, containers still have to be configured properly to achieve this extra security. Containers when correctly implemented do provide slightly more security than hosting multiple application on the same host.

The biggest advantage of packaging your software application in containers is the overall ease of management. Take the situation of hosting your application bare on a VM. Here we'd subsequently have to go through the process of installing all the dependencies of our software project and make sure the environment is properly configured to support our services, this process is tedious and requires some manual configurations to a certain extent. When utilizing containers, we use a container image file as a blueprint to define the configuration and packaging of our container, when building a container image, the file is followed to build our containers from during runtime. This blueprint will always be followed and automatically compiles the total package identically. This cuts out installing dependencies and most of the configurations in the host machine. Making containers excellent for leveraging a consistent and easy to manage environment (Dahlitz 2021).

Containerization of software has taken its key spot within the development lifecycle due to its strong benefits and portability. It is only natural then that VM's have taken a more sole hosting-based role and are not appropriate anymore for efficient software testing and hosting.

2.4 Docker

The concept of containers has been floating around for quite a while. But the subject has become stale before Docker reinvigorated the container idea when it came to the open-source market in 2013. In its original state Docker had many flaws, mostly being security issues. It wasn't until Microsoft entered the fold with offering containerized solutions and a large fundraiser that Docker became a widely used technology (Muñoz 2019). As it stands now, Docker is currently the most used technology for containers. The choice to utilize it in this project is thus natural as Docker containers can be launched individually through the command line or orchestrated through Kubernetes, Docker Compose and other orchestration software. This making it ideal for end-users to decide themselves how to orchestrate the containers according to their preferred methods. Docker provides us with Linux-based application containers (Murillo 2019).

2.4.1 Dockerfiles

Incorporating Docker within a software project is simple. The Docker engine installed by the Docker software utilizes a Dockerfile to build a container image. This file is a blueprint of how we will build our environment and run our software specifically. Within a Dockerfile you can create your own environment from scratch or utilize another image as a base image. This is done by stating a "FROM" instruction upon which you can expand your image upon in further instructions. One of these being the "RUN" instruction which will execute any given command in a new layer on top of the current image which is currently being built. Upon successful execution of the given command the image with the added layer is committed before the next instruction in the Dockerfile starts a new layer. The "COPY" and "ADD" instructions allows us to introduce files and/or directories from the host to the container. The later of the two instructions not being solely limited to the local host OS as it can also utilize network sources through the means of an URL. The "VOLUME" instruction lets us create a specific volume that will be mounted to the specified directory within the container. Volumes allow us to persist data between restarts. Docker containers are naturally isolated environments, in order to allow communication with our applications we will need to open ports for inbound traffic, this being achievable through the "EXPOSE" instruction. The final instruction within a Dockerfile is the "CMD" instruction. Here we can specify the running command which will be the starting of our software application. When extra executions need to be executed after our software application has launched, we can utilize the "ENTRYPOINT" instruction which will be executed upon the containers launch after the "CMD" instruction. Each instruction within a Dockerfile can be seen as a layer the total execution of all layers by the Docker engine will result in an image. Build arguments or "ARG" variables can be introduced which will be available

solely when building the image compared to “ENV” variables which can be utilized during both the build of the image and overwritten at runtime of a container (Jayawardana 2019).

2.4.2 Docker Image

An image is thus compiled from the Dockerfile, this image is the application and all its dependencies combined with the instructions for creating a container. This image will be executed at runtime to produce the final container which will host our application. Images range in size depending on the number of layers that are introduced, and which base image is being built upon. This size is quite important as the final image is required to be on the host which will run the container, thus making the image the distributed product for end-users to run a container from (Eschweiler 2019). It is in everyone’s best interests to reduce the image size as much as possible. This can be achieved with traditional Dockerfiles by limiting the number of instructions and combining “RUN” instructions as much as possible. Further reduction of size can be made by choosing to utilize a smaller base image. For example: Alpine based images are traditionally smaller and more optimized than an Ubuntu or Debian image. Full-fledged Python images are large in size in comparison to a Python-Slim image as the slim images contains the minimal number of packages needed to run simple applications and are built upon lightweight images themselves (Garcia 2020). You typically want to use a smaller image and add the required packages on top of it instead of using a full image with packages already included that are obsolete for your applications.

2.4.3 Docker tools for optimization

Docker also comes with various tools. Such as the ability to utilize multi-stage builds which can help us optimize our images. Multi-stage builds allow us to define stages within a Dockerfile which are independent stage builds defined within the same Dockerfile. Here we can reduce the number of layers defined in the final image by handling compiles and dependencies collection where possible in an earlier stage which then will be copied over to the final stage. These stages representing different images and only the final stage will compose the final image. All previous stages are discarded upon build completion (Ferrill 2021). When it comes to optimizing performance, we can utilize Docker BuildKit. This feature is beneficial as it enables enhanced caching of containers and its layers. Furthermore, BuildKit allows for the layers that are not dependent on other layers to run parallel which increases build performance significantly when coupled with multi-stage builds (Walker 2020). As such containerization projects should leverage these tools as much as possible.

3 Project details

The project that is subject to containerization is still actively in development and is looking to enter the productization phase. This phase being the making ready of the solution for larger distribution and a full release. Our full-stack project that's build on top of Robot Framework has been in development for quite some time. The project has naturally progressed into a working solution that is still being expanded. This development comes accompanied with various tools and technologies that are exterior to the actual software project but are present to build and enhance performance. These being Jenkins and Artifactory.

3.1 Robot Framework

Robot Framework is an open-source test-automation framework built on top of Python. This framework allows us to execute different test sets that can test a variety of components in an automated fashion. Robot Framework utilizes Robot files or script files containing keywords instead of functions to define the operations needed to execute tests. The basic Robot software is not made to interact specifically with different software or hardware but is expandable through the use of libraries. These libraries extend the use of the Robot application exceedingly as standard libraries are provided but can also be written for new applications, if need be (Robocorp 2022). The inclusion of Robot Framework makes up for the core component of our test tool that allows us to expand test boundaries significantly through the addition of libraries when building a container and at run time of the application.

3.2 Jenkins

Jenkins is an open-source automation software aimed for continuous integration and continuous deployment. This meaning that it offers pipeline solutions to test and build software when needed in an automated fashion. Jenkins can work with any type of software project through the means of plugins which can be enabled and allows for optimization within your builds. Pipeline jobs are automated processes which consist of multiple user defined steps which will be executed upon a trigger. A trigger can be a code commit to a Git repository. This software can then retrieve the repository and execute a test run and/or a build, in the case of a build job the Jenkins job can store the completed build in object storage or repository to your liking (Saurabh 2022). Jenkins is mainly used within our project to package and create our installers which we can test upon. The builds are manually executed by the developers when release testing is being conducted.

3.3 Artifactory

Artifactory is a package management software that offers its services by providing hosting solutions with repositories. Here users can create repositories for different purpose. In our use case we use repositories to cache project dependencies such as the python and node dependencies. Caching these dependencies within our internal network saves a considerable amount of time and potential costs compared to utilizing the node package repository. We are minimizing the download time here considerably by doing so. Furthermore, a repository to store our final builds allows developers and end-users to easily access the built solution to which they can retrieve them manually or automatically and deploy them. Artifactory thus having the ability to support both our development phase and build phase (Jfrog, s.a.).

3.4 Full-stack project description

Testing has become an import process within software development. Automated testing systems alleviates a lot of manual labour, but when it comes to specific areas such as research and development normal test tools don't cover the wide spectrum of needs as they have not been fully developed yet or require multiple tools to cover all areas of development needs. Our project which has created a high-level tool allows to execute test cases automatically. It is a distributed system which is highly scalable and modular as you can easily add different test environments and utilize it as a test editor. With this tool we can control any software or hardware as long as it provides some kind of communication interface.

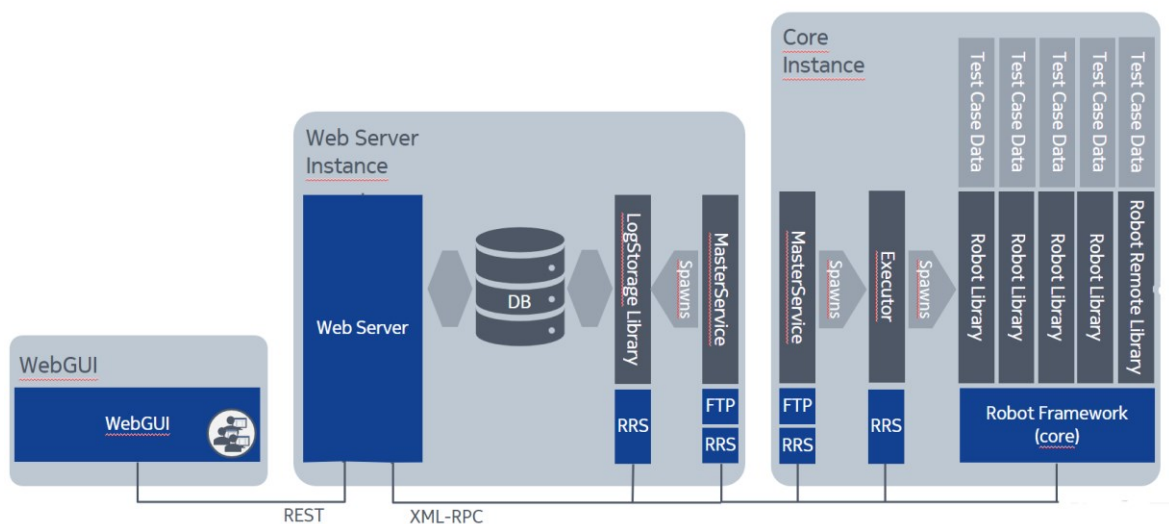


Figure 4. Project architecture of the main components.

The project is divided into multiple components which work together to provide the entire test solution. The MongoDB database as data storage solution, this database services

both the webserver and the core part of the project. Our node.js webserver is the middle component which provides the data on request through API endpoints for the WebGUI and provides the means to start/edit/stop executions on local and remote Core instances through XML-RPC requests, these being remote procedure calls encoded by utilizing XML. Robot test files are housed at the webserver and transferred through FTP when needed on by the core. The React.js WebGUI is our standard user interface and entry point for our end-users which allows us to configure our setup, choose the servers we would like to execute upon, analysis test executions, edit test cases directly, manage automation, and much more.

The Core instance or our executor instance as we also name it is a Python application built on-top of Robot Framework. It consists of a MasterService which is the high-level management of the application. It is the listening process which will initiate the executor process which spawns the needed robot libraries to execute the tests upon. The core instance is a versatile application as it is normally attached to a webserver directly but can also be deployed as a stand-alone remote service. This makes executions of different tests easier and allows for different core instances to fulfil different testing needs through different testing applications or libraries.

The project is currently ongoing which is now nearing its productization phase. This meaning that the tools are actively in testing while it's in usage and the steps are being made to finalize the project as deliverable solution. This productization phase has the goal to enhance the security and portability of the solution.

3.5 Problem statement

In the current state of our project, we package our total solution to an installer once a month when the newest version of the project is distributed. This aligns with our sprint schedules as we conduct a 1-month iterative sprint with 3 weeks development time and 1 week testing time where the main branch in the repositories is under a code freeze. This meaning that all changes and added features will be merged into the main branch and will be tested for any possible faults. The packaged installer will be deployed and tested upon different VM's with different operating systems.

A one week testing phase is quite long for development iterations, this is mainly due to limited resources available in our cloud. We have to utilize our cloud environment to load VM images on which we then install the newest version of our solution. Then we can conduct testing of the new features and changes accordingly. When a fault is found within

the new features, this has to be fixed if possible before the testing phase is completed, and the release of the new version is made. This means that once the fix has been implemented the solution has to be repackaged, installed and tested. Currently we utilize Jenkins to build our installers. In its current state it takes on average about 20 minutes to package a new installer according to our latest builds. This is due to the fact that regardless of the amount of changes the total solution has to be repackaged into a new installer. This means that at times our developers who are testing the solution are stuck waiting for the packager to complete and hampers testing significantly when multiple faults are found. What also costs a lot of time is the fact that we are testing our solution within multiple environments. This being windows, Debian, CentOS, Fedora, and Ubuntu. Naturally we want to ensure that our solution works within the environments we support. However, even with all testing is completed and we can make a new release doesn't mean that we won't receive fault notices from the teams who utilize our testing solution. Sometimes these faults are due to the environments they are installed upon.

3.6 Possible benefits from containerization

The containerization project has been on the project backlog for quite some time and is a productization requirement for both as a deliverable to end-users and its security hardening phase. It is only natural that this project must be completed as soon as possible as it will move our project closer to production and to reap the benefits for our development team and end-users.

By containerization the project we can make our deliverable solution more consistent as we are no longer just providing the software installer, but also the complete environment which will guarantee consistency within its isolated containers as the operating systems will no longer play a factor in the behaviour of our application (Pulfer 2018). Furthermore, the dependencies installation will no longer be handled by the packaging, but by the container images. If we individually pack our different components, we will be able to significantly reduce build times of the final image compared to our current Jenkins build job due to being able to only rebuild the individual images that have had changes since their last build. Furthermore, the biggest benefit to reap from this project is the overall time we can create for our developers to spend developing by reducing the testing time. As we only provide one environment per component we no longer have to test on multiple different environments and will also reduce the amount of environmental bug tickets from our current end-users.

3.7 Project objectives

As stated in the previous sections we have some clear problems we want to alleviate and the benefits of containerizing our project will fulfil this need. The clearest project pain being the testing time needed before we can provide a release and how much of a blocker the Jenkins job builds can be. Minimizing testing time and optimizing the automated jobs associated with providing the complete solution. Moving away from Jenkins build jobs dependencies and build the proper CI/CD pipelines to automatically build and store our deliverable. The container image size could provide a challenge as we no longer provide solely installers but a complete packaged environment which includes our projects components. We want to be able to reduce the size of our images as much as possible while still having the desired performance of the containers at runtime. Before phasing out the Jenkins jobs we want to make an effective difference between the build times of our current Jenkins build job and the containerized solution. An appropriate project goal would be at least a 40% reduction in build times.

As such the project goals are as follows:

- Containerize the projects components.
 - o contain size as much as possible.
- Optimize the build performance.
 - o Cut down build time as much as possible.
- Construct the pipelines so that we can automatically build and store our project's deliverables.

While containerization and constructing the automated pipelines are the main goals of the thesis project, we still want to discuss the hosting of the containers for demonstrative purposes. This however is not a project goal because we as providers and developers have no control of how end-users will orchestrate the container solution, but merely can provide an example. As such the theory of hosting the containers will be discussed.

3.8 Implementation plan

Due to the nature of the project and the possible problems that might occur during the implementation process as containerization is introduced in a late stage of the project the implementation method will follow a development work methodology. This methodology refers to the use of new information that will be generated by research in how to implement the containerization and the practical experience before and while implementing to move the project along. To achieve the end results of the project we have to split up our tasks accordingly in an order that would make logical sense, this being a containerization phase and a pipeline phase.

3.8.1 Containerization implementation plan

The containerization phase will take all the processes into account to successfully create the images of our solution. Here we have opted to utilize Docker as our containerization tool for consistency with other project and its universal usage. We want to individually package our 3 main components independently within their own isolated containers. These being the database, webserver and the executor. The order of our containerization phase will be to create the image of the database first before the executor image and finally the webserver image. Normally you would want to separate the webserver's server and client components into individual containers. However, since our server and client communications are not solely through API interfaces but also shared files in directories which both components need to access means we will have to put the entire webserver within the same image. To successfully implement our solution, we will have to take a close inspection at the Jenkins build jobs and the installer build scripts to be able to mimic our desired environment with the container images.

Where possible we want to utilize lightweight base images to build our final image upon. These being mostly slim or alpine based if applicable. These images provide the lowest size while in theory providing the same functionality. Using full environment images would result in our finalized images being over 1 gigabyte in size. This would be quite hampering for both storing the images and distributing them. Appropriate research and testing must be done to measure the viability of the images in question to assure their performance to be also satisfactory to our needs. Due to the nature of the project planning will only get us so far. Containerizing a product that has already been in development for quite some time is very touch and go as to what works and what not. As such a simple plan is created during implementation and modified according to what works and what does not by creating a proof of concept.

3.8.2 Pipeline implementation plan

Containerizing the project is the biggest chunk of work due to the nature of research and work that goes into it. The pipelines phase refers to the construction of Continuous integration and continuous deployment pipelines. We want to make it so that our images get build automatically as changed are being applied to the GitLab repositories which house the webserver and executor project, the webserver repository also houses the database scripts. Due to the nature of containerization, we have already defined how the images are built within the first phase of the project. In this phase we make it so building is automated through a pipeline job which will retrieve the repositories, stage the Docker build and finally store the completed image in an Artifactory repository. Here developers

and end-users can retrieve the completed builds. 3 pipeline jobs will have to be constructed for each of the components and only changes to the component will cause a rebuild of the image that requires the changes. Other components should not be rebuilt if no changes have been made to the project or the build job. For example: when a change has been made to the webserver repository to the webserver component then only the webserver image should be rebuilt as no changes were made to database scripts nor the executor/core repository.

We want to create one or more GitLab runners as the host where each of the pipeline jobs will be executed. Introducing a runner gives us advantages that our source code is secure within our own environment, and that we can introduce scheduling and specific configuration to optimize performance even further, if need be, in a simple manner compared to utilizing a single CI server. (Levan 2020)

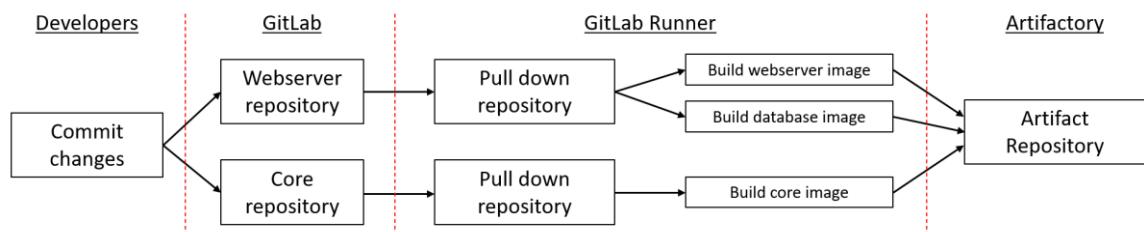


Figure 5. Basic diagram of the pipeline jobs.

When developers commit changes to the respected webserver or core repository, we want to build our Docker images and make these available to our developers so that they can test them in production mode. Furthermore, the pipeline will also create the images that would be released to our end-users upon merging all branches to the release branch.

To achieve this, we will have to take use of a configured GitLab runner as host to the building process and our GitLab to trigger the build job upon merges and commits in the GitLab repositories. Once the pipeline has been triggered the GitLab runner should pull down the changes made to the project locally. Here it will initiate the build of the images and upon completion will push the completed images to the Artifactory repository.

Artifactory repositories support different software packaging systems and automation. This repository will be the end point of our pipeline to store the completed images to which our end-users can retrieve them with the possibility to do so in an automated manner through their own deployment pipelines.

4 Implementation work

When it comes to the implementation, we start off with the containerization phase. Here we will walk through the process of planning, implementing and solving problems as they occur in real time with each component. The actual code files can be found within the appendices section of the thesis due to the large nature of the code that being is explained. The first component to be containerized would be the database as this component is required for both the executor and the webserver container. The second process would be the executor container as the webserver requires the executor to be present to execute tests. After the containerization process, we would build the pipelines required so that we can build the Docker images in an automated fashion.

4.1 Database containerization process

Currently the project utilizes MongoDB 3.6.8. However, this version of MongoDB has passed its end-of-life date meaning that it is no longer receiving future updates which is generally bad practice to implement. As such the intention is to update the MongoDB to version 4.4 if possible or at least version 4.2 which have been verified by our internal security specialists. As the default MongoDB images are quite large (400 MB), we want to opt for utilizing a normal Alpine image as the base image to build upon for our database image due to its small size and enhanced security because of the size.

As far as configuring the database goes, we can mimic our image build to the previously utilized installer script. The installer will check if MongoDB is installed within the system, if not it will install MongoDB on the system. Upon completion and verification that the MongoDB is installed and reachable the installer will start executing an import script which will create the needed database structure and insert the data required to be able to start the application.

Converting this in a containerized solution should be straight forward. Taking an Alpine base image, we'll have to define the installation commands for the MongoDB 4.4 to install the database ourselves from the Alpine apt repository in the container. As for the configuration and imports of our data through the means of scripts we can utilize a bash script as the ENTRYPOINT. This command will automatically run any file upon container launch which makes us able to run our import script upon launching the containers. This being convenient as the import script is already optimized to run the json file imports. The standard MongoDB service ports 27017 has to be exposed for inbound connections. Finally, a Docker Volume will need to be attached to the container at runtime so that the database contents persists when the container gets removed or replaced.

4.1.1 Problems occurred

At the start of implementing the mentioned plan, we came across the issue that MongoDB was not available in the Alpine 3.15 apt repository which is the current version of Alpine. Upon further investigation it was found that MongoDB 3.2 was the highest available MongoDB version available, which was in the Alpine 3.6 apt repository. Searching why this was the case has revealed that Debian, Ubuntu, Fedora, Alpine and many more open-source Linux operating system distributions have stopped their support for MongoDB versions 4 and beyond within their apt repositories which only offers open-source software. This is due the worry of MongoDB's licensing changes for their 4.0 versions and all subsequent versions which is now under the Server Side Public License or SSPL. The licensing change has caused some negative comments towards MongoDB from various open-source communities and project maintainers. This resulting in decisions to stop the inclusion of MongoDB in major distributions as stated by Tom Callaway (2019), RedHat developer at the time about the SSPL license change and how it affects the Fedora OS project.

Previous versions of MongoDB utilized the GNU AGPL v3.0 which indicates that MongoDB versions lower than 4 are free to use as long as copies of the license text and copyright notice are included, changes to the original software are indicated to users and the source code is made available if any binaries are created and distributed on the original licensed software (GNU 2007). However, SSPL requires all surrounding source code and infrastructure to be publicized or to get a commercial MongoDB license. (MongoDB 2018) This change alone is makes it usage for proprietary software invalidated. Upon checking our legal licensing review service, we confirmed that MongoDB versions 4 and upwards are not approved as an open-source software due to the SSPL license.

4.1.2 Implementation changes

After a team meeting, we unanimously decided that we will not be moving forward with MongoDB as our database solution due to the current version we are using being past their EOL. This means that the containerization of the database will fall back to our original project equivalent of MongoDB 3.6.8 under the old AGPL license as a temporarily solution till a study has been conducted to which alternative database solution shall be implemented. This process is quite long and falls outside the scope of the thesis due to the study and implementation changes within the project that follows.

Our implementation changes to use Bitnami MongoDB 3.6.8 as our base image. Bitnami still supports their older images making it especially useful for patching the environment for future security issues or bugs that might happen as MongoDB versions below 4 are not updated anymore by the developers themselves. Furthermore, Bitnami base image is based on mini-Debian which makes it a more lightweight solution than a full MongoDB image (Bitnami s.a.).

4.1.3 Implementation result

The resulted image is straight forward. We utilize the Bitnami MongoDB 3.6.8 base image. Our steps include the regular update of the package lists before installing dos2unix within the container. Dos2unix is a needed dependency within the database containers since we are developing within Windows environments, meaning that our scripts are written in Windows and could possibly have non-Unix line endings which prevents execution upon Unix systems or our Linux-containers. To prevent these from ever occurring we make sure to parse our scripts through the dos2unix functionality before executing them (Appendix 1).

Normally we would want to avoid running the final software process as a root user to enhance security within the containers. However, MongoDB is a service that requires root privileges in order to be started, creating a standard user without root privileges within the container is subsequently unnecessary.

Database images come with an entry point directory. This being the “Docker-entrypoint-initdb.d” directory. All the scripts files in this directory will be automatically executed by the engine at runtime after the container is launched. Meaning that we simply copy all our scripts over to this directory, including our json files container the data imports. This entry point only executes bash or JavaScript files so our json files will be ignored. This is ideal as our import script handles the inserts of the data within the json files. However, we'll have to modify the import script slightly to make our project root user within the authentication database before all the data is inserted as the imports and the webserver both require this user to exist to be able to connect to the database. Finally, the MongoDB service port 27017 is forwarded and a volume statement is set.

Upon running the build image, we can verify the container is created and the runtime executes our import scripts elegantly before restarting the MongoDB with authentication enabled. With removing the container and starting a new container we can verify that the

Docker volume works as intended and data persists, here our import scripts are not re-executed as the data already exists within the database.

4.2 Executor containerization process

The executor container will be the middle ground of difficulty. Not as straight forward as the database container but also not as difficult as the to be webserver container. To build the executor container we will have to create a python environment. Here we will opt to use a python slim image instead of Alpine due to how pip installs python dependencies and the fact that Alpine is quite unfavourable for Python applications where high performance is favoured.

Alpine is built to be as small as possible. Alpine has been built upon the musl libc library instead of glibc library upon which usual Linux distributions are built to achieve its small size (Alpine s.a.). Python utilizes the GNU C library for many of its low-level components. Making a C compiler a requirement, usually this is provided by the operating system as most Linux distributions have it as a requirement (GNU s.a.). However, Alpine does not have this natively within its system making it so we'd have to install C language compilers on our Alpine image. These compiler packages are quite large which would impede on the use case of Alpine with its small size. Furthermore, Turner-Trauring I. (2020) describes the situation well in the case of Docker containers when installing Python dependencies. Pip normally installs Python dependencies by downloading pre-compiled wheels. These Wheels are then utilized to install the dependencies. At the date of writing musl packaged wheels were not pre-compiled yet, meaning we'd have to download the raw source code all the dependencies and compile the wheels ourselves for musl Linux. Since the writing of this article a motion to create a standardized tag for musl Linux wheels has been accepted resulting in pip being able to provide pre-compiled musl wheels (PEP 656 2021). When creating a proof of concept of the executor container utilizing Alpine, we witnessed a 9-minute build time due to the bulk of pip wheels were not musl compiled wheels. This is indicative that although a musl Linux tag has been introduced, library developers do not offer pre-compiled wheels to the pip repository. This makes Alpine a less than suitable base image for our Executor container and a python-based slim image the best alternative. We'll have to install one general compiler which is missing in the python-slim image but we can immediately remove it after all the dependencies are installed within our environment, reclaiming the space it takes.

Our executor must be globally available within the container environment and our Jenkins build job has revealed that we compile a wheel of our executor project. Here we can opt to

use Docker's multi-stage build feature to reduce the size of our final image. In our first build-stage we will compile our executor wheel. In our final-stage we will copy over the wheel from the build-stage and install it after all its dependencies have been installed.

4.2.1 Problems occurred

When creating the blueprint for the executor image it seems to be that logs of test runs are not directly posted to a webserver or stored within our Database. Rather in our original installer-based solution it seems that the logs are saved in a local directory which the webserver then can access in the case of an attached executor. As containers are isolated from each other and by default can only communicate with each through the network interface. This means we'll have to implement a shared volume that will be implemented in both the webserver and the executor container. This is not a problem for Docker's side of the implementation, but this is quite problematic for the future of the application. Introducing a shared volume can pose a data migration problem in the future to which container will handle this migration. Furthermore, the overall structure of the logs and/or the log directories can change as development of the project is still ongoing, making a data migration of the shared volume unavoidable as else both the webserver and executor will not function properly.

4.2.2 Implementation result process

With the shared volume problem brought up to the other developers, we decided to move forward with the shared volume as a temporarily solution until both the webserver and executor implementations of the log storage service are changed to utilize MINIO technology. MINIO is an object storage software solution that can also be utilized as a stand-alone container that will act as the storage for the logs through its API functionality. (MINIO s.a.) The executor component will then provide these logs to a MINIO container from which the webserver can retrieve them. Instead of a shared volume we would place this object storage container in our total solution. Due to the large amount of code refactoring to be done to both the webserver and executor, this solution falls outside the scope of this project and will be implemented at an appropriate time (Appendix 2).

The rest of our implementation follows to the plan. We utilize a first build-stage where we compile the executor wheel after obfuscating the actual code. After the executor wheel has been compiled, we start our final image where we install all the dependencies needed in order to be able to run the executor program. We are also installing optional test libraries for our end-user's convenience. Our team has cached the pip dependencies in Artifactory since the original Jenkins build jobs have been constructed. This makes

downloading the dependencies instant. Our container must come with an environmental variable to define the local public Ip-address for communications to the webserver and log storage, in our original project implementation it was found that localhost is unreliable. The environmental variable can be overwritten by the end-users at runtime with the correct Ip-address. Our executor service ports are exposed, and the volume statement is set to preserve the test logs after container shutdown or restart. Finally, we start our executor service with the designated python command.

4.3 Webserver containerization process

The webserver container will compromise of both the client and the server. Normally you would split the client and server into sperate container in API design. However, as there are some commonly shared resources and non-API communications, we will have to combine the two components together in the same container. This is quite a hinderance in performance as node and react dependency installations and project builds are time costly procedures. This being the current process within our Jenkins job following the webserver build script.

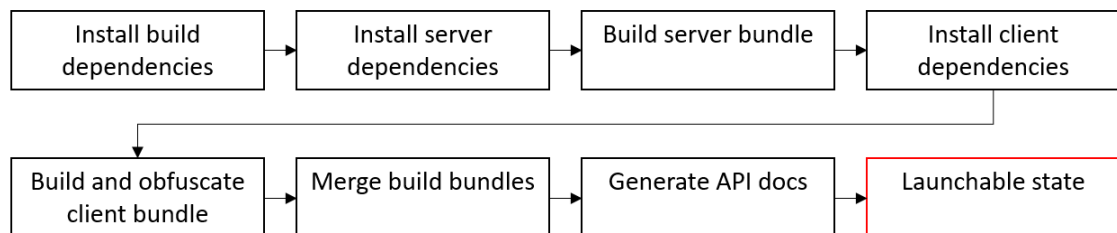


Figure 6. Simplified diagram of the current webserver build steps.

If we would follow the build script to the dot our build would be considerably slow as we would be moving to through the build phases sequentially. However, we can cut these build times down by splitting the client and server build processes in their own build-stage with multi-stage builds. Separating the client and server into their own build-stages will allow the stages to run parallel with each other up until the final stage where the needed built bundles are copied over thanks to the BuildKit feature. Here we will also benefit from the fact that only the component that has had changes will be rebuild in the feature and the cached state of the unchanged component will be utilized. This will require some rework to the build scripts as these have been created for a sequential build where first the dependencies are installed before the server will be build and finally the client will be obfuscated and built. After the building process the bundles get combined, and a binary will be created. Here we will have to split these functionalities of the build script apart into their respective container.

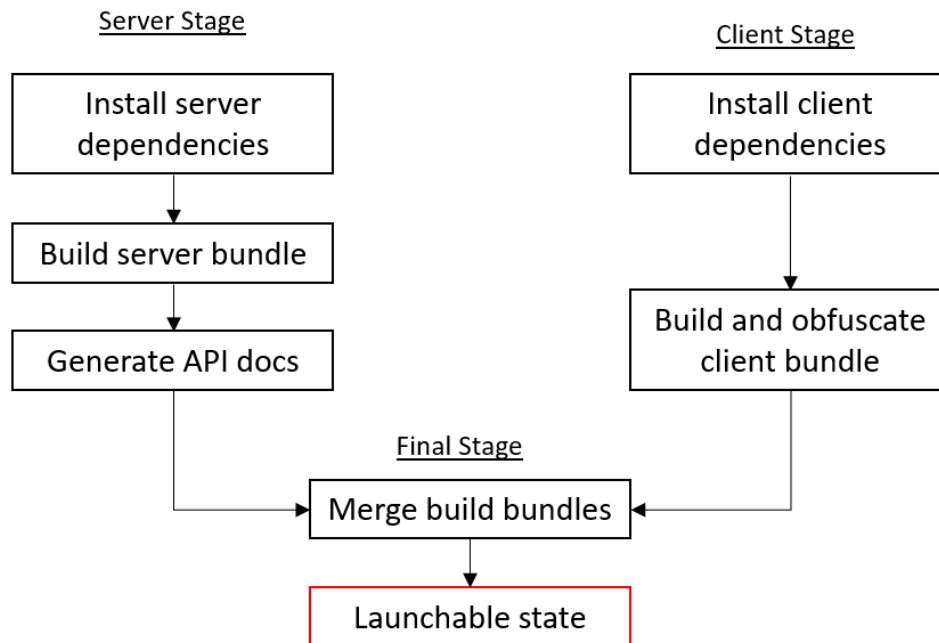


Figure 7. Diagram of intended build steps accommodating multi-stage builds.

The final-stage base image will be a node-slim as we will benefit from have some compilers pre-installed in the environment compared to Alpine where we would need to install everything ourselves in a heavy already time consumption build process. The final image will copy over the compiled server and client files in order to be able to run the stack appropriately. As per the executor container we will have to mount the shared volume here as well to be able to access the logs. An additional volume will have to be mounted so that we can save user files which includes their robot test files.

In this container we will have to utilize an ENTRYPOINT script to create the configuration files for communication with the MongoDB container.

4.3.1 Problems occurred

In the testing of the webserver container, we have found that there is still a Python function call made to retrieve the version of the attached executor. This has been written at the start of project and assumes the executor is within the same environment. This will have to be rewritten to make an XML-RPC request to the local executor container instead as the executor is within its own isolated container now. This will mean that an environmental variable must be introduced for the public Ip-address of the executor to where this request needs to be sent.

Another problem found during testing was in regards with the licensing of the webserver. Upon inspection of the installer scripts, it was clear that a universal unique identifier is generated and saved to the environment as the installation identification. This identification string is needed in order to be able to generate a valid license.

As our solution is already in use by end-users for mainly testing purposes, we have established a licensing server centrally within our intranet. Here we verify a license request key generated by the project's webserver. This request consists of an installation identification number which is the unique identifier we need to generate, a valid email address and purpose of utilizing the project. Upon copying over a valid license request key, the license server will automatically generate a license key which can be applied to the webserver and enables the usage of the testing tool.

The file which contains the installation id will have to be preserved accordingly as the installation id string is checked upon test executions against the license to make sure the license is valid. Thus, we will have to make the final stage generate this installation identification file and preserve it through an attached volume to its saved directory as we do not want to have to generate a new license every time a container is replaced. As we cannot generate this identification file during the image build time, we will have to add this functionality to the ENTRYPOINT script to be executed at container runtime if not installation identity file was found. If it was implemented during the image build, then every container derived from this image would have the same installation identity.

An oversight in the planning of the containerization was that the test editor which is embedded in the client utilizes XML files to describe the Robot Framework libraries which are installed upon the executor. However, this XML files are generated by the executor itself and the webserver accesses this directory. To resolve this, we could use our previous build executor image. That being the introduction of third stage which will run parallel with the client and server build stages. Here we can utilize our executor image that we have built as the base image and overwrite the CMD instruction with the python script that generates the needed XML files as the Robot Framework libraries are already installed within the image. These XML files can then be copied over to the final stage.

4.3.2 Implementation results

The resulting image is thus split into 4 stages. The first 3 stages will be running parallel which are the server and client build stages and XML generation stage, these client and server build stage utilizing the node-stretch base image which has all the necessary

compilers present. The server build stage will install all the server dependencies, these being the node packages. Upon its completion we will run a build of the server which utilizes the webpack library to build, our webpack config does not include the packaging of the node modules. This is mainly due to the splitting of our React and Node with separate builds and bundling them. After the build has been completed, we generate the API documentation which is included in our end solution. Finally, within this stage we bundle our solution and run another node install in production mode. Here all the development and build dependencies will be removed so that everything can be copied over without bloating our solution. The second build stage or the client build stage follows the same pattern except that here we will run an obfuscation of our client build. This is done to prevent the logic of the application to be exposed to end users and to avoid tampering with the solution (Appendix 4).

Our final build stage utilizes a node-slim base image. The choice for node-slim being optimal in our situation is because it's considerably smaller than the node-stretch image and only one compiler is missing which can be easily installed. Within this stage we will need to install Python as our we provide Robot Framework test editing service within our webserver directly. This means that both Python and Robot Framework will have to be installed within the container in order to provide the correct syntax and support of this test editing feature. Once this is done, we copy over the build bundles from the client and server build stage, the node modules from the server, and merge everything into one directory ready to run. To allow communication between the webserver and the database we will have to provide a configuration file which is editable by our end-user if they host the database remotely or locally. This is achieved by an environmental variable which can be overwritten to specify the database Ip-address. From this environmental variable we will create the proper configuration for communication with the database in an entry point script before starting our built webserver. The entry point script will also create an installation id if none has been generated yet by checking our persistent volume attached if the file exists.

As stated in the previous section we would technically need 3 volumes to accommodate both the user data, the installation id and our log storage shared volume. However, we want to lower this to a single volume to better accommodate our solution and leave on single point of data needed to persist. To do this we utilized symbolic links. Symbolic links are files that are put within a directory that will point towards another file or directory, if a process is looking for a specific file which has a symbolic link, then it will be redirected towards to actual location of the file. This makes it so we can create a data volume and use 2 symbolic links. 1 link for the installation id file for our license to work and 1 for our

user data directory. Now instead of 3 volumes attached to the webserver container we have 2 (see appendix 3).

4.4 Pipelines construction process

4.4.1 GitLab Runner creation

Our solution is now successfully containerized, the next step is to construct the pipelines to automate the building process of our Docker images. You can let the GitLab server handle your pipeline jobs by letting jobs run locally. However, the configuration of a single CI server is quite difficult if it would handle a variety of different jobs. (Levan 2020) This is where a GitLab runner comes to leverage self to setup hosts for specific pipeline jobs. A runner is thus a host on which the GitLab software is installed upon that can be registered to a GitLab server. Here we can define one or more runner executors upon which we run our jobs. We can define different type of executors upon which our jobs will run, these being shell, VirtualBox, SSH, Docker, Kubernetes (Despa 2021).

Shell based executors are generally the easiest to configure into a ready state, but they do not offer the best performance possible. In our use-case we would benefit the most from utilize Docker as our executor. Here we can further use Docker-in-Docker service. This allows us to run Docker inside a Docker container to build our images. This makes it so the environment used to build our Docker images is also cleaned up after each job and to build somewhat faster (Wooster 2017).

Setting up the physical runner for our usage case is a quite easy process which requires you to install Docker before installing the runner software. After installing the runner software, we will have to configure our own runner using the register command. Docker runners recently started to utilize TLS certificates to protect the host from malicious processes breaking out of Docker containers, this would take some extra configuration. However, as all our hosts and servers involved with the GitLab CI are within the same protected intranet we can opt to expose our Docker socket which is the Docker runtime directly to the Docker container which will build our images as a volume. Even with TLS certificates enabled we would have to expose this socket to be able to execute Docker commands within the container. Registering the runner with the intended Git repositories is done during the setup as well. Here we need a GitLab registration token to identify the runner to the specific repositories for which it will run. As we have 2 repositories which house our total project, we can opt to house them into a GitLab group. In this manner we can register our runners to work for both repositories as a group runner instead of a single repository directly. The final step for the configuration is to configure Docker to utilize our

http proxies which act as gateways for retrieving outside intranet resources. This can be done by simply modifying the Docker configs to state intended proxies. Now our GitLab Runner is successfully configured and is ready to execute upon.

<pre>gitlab-runner register -n \ --url https://gitlab.com/ \ --registration-token <Group_TOKEN_From GitLab> \ --executor docker \ --description "Docker CI Runner" \ --docker-image "docker:19.03.12" \ --docker-volumes /var/run/docker.sock:/var/run/docker.sock</pre>	<pre>~/docker/config.json { "proxies": { "Default": { "httpProxy": "<ip-address>" "httpsProxy": "<ip-address>" "noProxy": "<ip-address>" } } }</pre>
--	--

Figure 8: Simple GitLab Runner configuration with Docker as executor.

4.4.2 Pipelines implementation

With our runner all set up and ready to execute upon we can continue and start creating our pipelines. We have two repositories on which we will have to construct the pipelines upon, the webserver and executor repositories to be specific. The pipeline definitions will be mostly the same for both repositories as we are utilizing Docker in Docker and our merely building the images and publishing them to Artifactory. GitLab uses a single file within the project repository to define the jobs to be executed when changes are made to their respective repository. Within this file we will define our 2 jobs and define the order as stages (Appendix 4). This being the build and publish jobs. Technically you could combine these 2 jobs into a single job.

Before we get started with the implementation, we must note that our project utilizes Git-LFS. Git-LFS which stands for "Large File Storage" is a Git extension. It is used to manage large files and binary files in an alternative repository designated for large files effectively storing them outside of the normal project repository to avoid bloating the size of the repository. Having these large files present in the normal repository would make our GitLab run significantly slower as GitLab keeps a history of files and pulling these files down from the GitLab on every pull request would make the process significantly slower (Rotsaert 2018). This impacts our choice for the Git Strategy within the pipeline definition. A Git strategy definition can be clone, fetch or none. Here we can choose how we will provide the project files from the repository to the GitLab Runner before building. Clone would automatically pull down the project files when starting execution within our Docker container before it will build the image. Utilizing fetch would mean that we would have to clone the repository on the GitLab runner host, and the runner will automatically pull down the latest project changes before executing jobs. The none setting will disable the feature

to automatically provide the project files to the runner and will make it so you will have to manage the accessibility to the project yourself (Sevat 2020). In our use case we cannot use fetch as we have a Docker container to build our images in. Using fetch would mean that we would have to expose the project files from our Runner host to the container through a volume and Git-LFS would have to be managed both inside and outside the container. With clone we would have to still install Git-LFS within the container and pull down the large files. Subsequently, the optimal choice for us would be to handle the cloning of the project files ourselves.

When it comes to constructing the pipeline, we must choose a base image which the pipeline will run a container from in order to build our project images. As our chosen method is Docker in Docker building, the official Docker image comes with a version which has Git preloaded. Choosing this as our base image makes so we only have to install Git-LFS inside the container. As the Docker image is alpine based we'll have to update in order to be able to install Git-LFS and initialize it. After this we will configure our SSH keys inside the container so that we can pull our project from GitLab and the needed Docker images from our Artifactory mirror before we pull down the project files. With Git-LFS initialized, the large files will be automatically retrieved as well upon a git clone. These steps are considered to be pre-configuration of the container before the actual job takes place and thus are marked as the "before script".

GitLab pipelines come with a variety of variables you can use and define your own. We use a variable to dynamically name our Docker images. When it comes to Docker images, we will have to name them and tag them appropriately. The name is specific to the image we are building but to display the versioning Docker utilizes what is called tags. Tags can be anything from a number to a word to define what version the image exactly is (Subheksha 2018). In our case we are naming our images according to the component they are and tagging them respectively to the commit sha which has triggered our pipeline job to start, so the developers can easily identify their built image against the commit to the repository they have made. A commit sha is a 40 character long hashed identifier based on the meta data and contents from a commit (Burgdorf 2014).

To enable Docker in Docker builds from the pipeline's side we will have to state the "DIND" service so that the needed supporting software and configuration of the container is automatically provided. In our actual build job, we export the BuildKit variable in the container to ensure that the Docker BuildKit feature is enabled before we build our actual image. Here we want to save our image to a compressed file so that we can artifact our image for a limited time in the GitLab server. An artifact is a file or directory stored within

the GitLab server after a pipeline job has been successfully completed. Here all the future jobs within the same pipeline can download the artifact from the GitLab server to be available in the context of the next job (Yakutovich 2021). What's also useful is that developers can retrieve the built image directly from GitLab for a limited time, they do not immediately have to retrieve it from the Artifactory.

In the publish job we automatically download the artifact from the build job thus we do not have a need to utilize or re-run the before scripts to retrieve the project files as we do not need them. Here we login to our Artifactory service and re-tag our image with the same name but include the Artifactory destination repository before we upload the image to the Artifactory. After the upload has been completed, we remove the built Docker image to clean up the runner. If we would not remove the image the GitLab Runner host would run out of space quite fast.

For the webserver repository we take on the same template as the executor pipeline (Appendix 5). The only difference here being that we are combining the database and webserver build within 1 build job and the upload to Artifactory in 1 publish job.

Technically we could split the database and webserver build in 2 build jobs and 2 publish jobs. However, as the database image will rarely be rebuilt due to it being a component that almost never changes. Having to rebuild the database image is only necessary when database scripts change. Thus, we would lose time starting an individual job for the database as compared to building the webserver and database image at the same time due to a small overhead taking place when a job is starting. This overhead being starting, stopping the container and possibly downloading a previous job artifact.

4.5 Storing container images

Like indicated before we upload our images to Artifactory. Within our Artifactory we already have a Docker registry. A registry being the Artifactory which hosts the Docker repositories. The Docker repository being the host where we will store our Docker images (Atzmony & Aleksandrowicz 2022). These repositories are local as opposed to remote repositories such as Docker hub. A local repository allows us to privately store our images as to where Docker hub also offers publication of Docker images. To ensure our own proprietary software stays private we host our own Artifactory. Other registries do provide private features however, to ensure security we chose to utilize our own solution.

Our Docker registry and repository were already created before the project commenced for proof of concepting of the solution. The process to create these can be done in a

guided fashion through the Artifactory user interface. A useful feature of Artifactory which can be integrated in our project is the promotion of Docker images. Promotion is a feature that allows us to move Docker images from one repository to another without having to download the image and reuploading it to another repository (Atzmony & Aleksandrowicz 2022). In our case this is useful to promote a Docker image from our own internal Artifactory repository to a release repository for end-users. Here in the next containerization process to make the containerized solution available for the end-users we can utilize the promote feature to move tested images that make up the developed work of the last sprint a release repository.

4.6 Hosting the containerized project

The hosting of the containers is not included within the project due to the variety of possibilities when it comes to hosting the resulting containers, end-users have the freedom to host the containers how they see fit outside of our own recommendation. However, the topic is in need for discussion as orchestration is required for running our entire container stack in a feasible manner. Container orchestration is the management of all processes related the containers. These being the lifecycle, environment, automation of containers through various software tools. The lifecycle of a container specifies the acquisition, deployment, scaling, redundancy, monitoring and replacing of a container (Eldrige 2018).

The simplest tool to orchestrate containers would be to manually bring up the containers through Docker Compose. Compose is mainly used for its simplicity within the development and testing cycles of containers. However, it is not that uncommon to utilize it for orchestration containers within production when it comes to smaller projects. With Compose you create YAML file with the definitions of all the images you want to run into a container. Within this file you can define images, virtualized networks configurations, variables, volumes and much more. This allows us to bring up a whole stack of containers and network configurations with a single command (Avishek 2020). But, Compose can only take care of the running of containers from Docker images. It does not however manage the containers once they are running. When it comes to configuring redundancy, scaling and the monitoring of the containers the end-users would have to either script these services manually or utilize other tools (Wallen 2021). This makes Compose rather undesirable and not our official recommendation to end-users.

What we recommend and use internally within our team is Kubernetes. This solution is an entire platform which can manage entire workloads and services when it comes to

containers. This includes everything related to the lifecycle of containers compared to merely starting the containers with Compose. Kubernetes is an excellent solution for deploying, scaling and managing containers on a single host or multiple ones. With Kubernetes you can create a cluster which consists of one or more nodes. Nodes are individual hosts upon which the containers can run, and pods are the manageable unit that can contain one or more containers. Pods come alongside a YAML file which describes the pod and its contents. Here the specific containers are described, and configurations can be maintained. Outside of deployment Kubernetes can replicate pods to accommodate for high traffic with load balancing and scaling. What makes Kubernetes especially desirable is the monitoring of pods. Automation within Kubernetes allows for the monitoring of application and container health. In the case of an ungraceful shutdown the manager can be configured to try and restart the specific container or to bring another one up instead. (Hwang 2019). Within our company we have opted to utilize MicroK8s. MicroK8s is a more lightweight Kubernetes solution. What makes it special is that it's the fastest solution available in the sense that it can take just a few minutes to go from installing MicroK8s to having your pods up and running. It provides most of the normal Kubernetes features while also being simple to manage (Anaqvi 2019). However, end-users can utilize any Kubernetes solution they seem fit.

Managing a large number of containers can be painful as each pod needs to be described with a specific YAML file. Kubernetes utilizes these files for the creation of its resources, deployment and services as Kubernetes is managed created in a declarative way. Forcing the containerized implementation of our project could leave the end-users with the need for coming up with their own YAML file definitions which for example, includes the allocation of physical resources to the containers. They might allocate a too low number of resources which might make our project's application run slow or cause problems. Some form of YAML template will have to be provided to end-users so that they could properly run our containers. This is where Helm Charts prove to be beneficial. Helm aims to solve this specific form by allowing to make templates. As opposed to the standard YAML files, we can create Helm charts to describe a pod and child charts to describe the individual containers as a template to the end-users (Santos 2021). This will allow them to run our containers according to what we find the correct configuration.

Helm Charts are YAML manifests that are combined within a single package that allows easier management of your Kubernetes resources. When traditionally using the standard YAML files a developer would have to state the same information multiple times at different levels to be able to deploy a container. Helm makes this process easier by allowing you to create a package that can be promoted to your Kubernetes cluster directly.

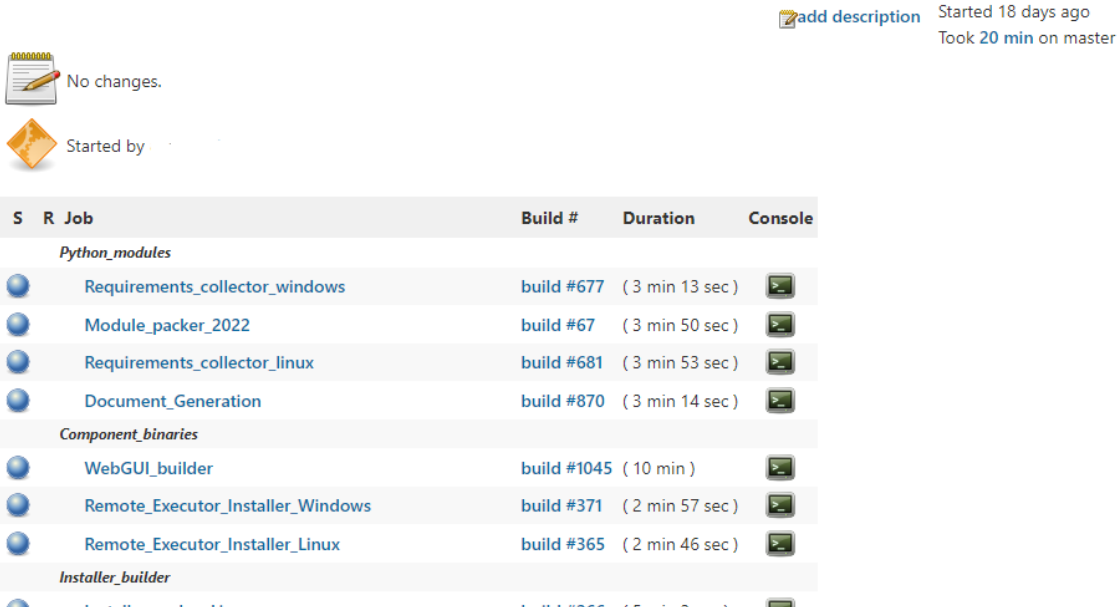
Helm charts can be thought of as a package manager in the sense that you define the resources, versioning, and instructions within the charts. You can then package these charts and deploy these directly to the Kubernetes cluster instead of providing standard YAML files for each resource, pod, or pipeline (Idowu & Merron 2020).

When providing the images together with the templated Helm charts the end-users can run the solution in a way that we recommend, we'd have to make a general baseline on the amount of physical resources that would be needed. This means that a next step to the hosting solution would be to benchmark, and stress test our containers under different circumstances so that we can create a correct template.

5 Evaluation

When it comes to evaluating the Jenkins state to the to be introduced state, we are mainly interested in the comparison of the times. The evaluation doesn't directly critique the Jenkins software, but it brings in the bloat of build time as the main stressor. This bloat being caused by the growth of the software project and the interlinked dependencies of the components when building with Jenkins that causes long sequential build phases. It is possible to optimize the Jenkins build jobs. However, this would just like the Docker process require to split the components to be more independent. The size of an installer cannot compare well with that of a containerized project. We are looking to provide multiple software environments which will run our software project on top off compared to just providing the software.

5.1 State as is with Jenkins



add description Started 18 days ago
Took 20 min on master

No changes.

Started by

S	R	Job	Build #	Duration	Console
<i>Python_modules</i>					
		Requirements_collector_windows	build #677	(3 min 13 sec)	
		Module_packer_2022	build #67	(3 min 50 sec)	
		Requirements_collector_linux	build #681	(3 min 53 sec)	
		Document_Generation	build #870	(3 min 14 sec)	
<i>Component_binaries</i>					
		WebGUI_builder	build #1045	(10 min)	
		Remote_Executor_Installer_Windows	build #371	(2 min 57 sec)	
		Remote_Executor_Installer_Linux	build #365	(2 min 46 sec)	
<i>Installer_builder</i>					
		Installer_maker_Linux	build #266	(5 min 2 sec)	

Figure 9: Latest completed project installer builds with Jenkins.

Jenkins currently handles our installer builds which is a process that is manually triggered by our developers when they want to test the project in production mode. Here we can notice the overall length of a complete build takes 20 minutes according to the latest build and recent trend (Appendix 6). This time is consistent overall according to the latest couple of builds. Most notably is that our Jenkins can run various individual jobs parallel. However, the speed of the builds is blocked due to that many jobs are dependent on the build processes of previous builds.

The first phase of the entire build is the handling of the python requirements. This being the collection of the libraries, dependencies and modules required to make the executor run. The module packer handles that packing of what the collection job produces so that the needed python externals are present within the installer bundle. The document generation job is the generation of the needed XML files for the test editor documentation within the webserver as explained in chapter 4.3.1. In this phase we have some parallel work going however, the document generation and the module packer jobs are dependent on the completion of the requirements collector jobs.

The second phase of the Jenkins build is the building of webserver and the remote executor installers. The remote executor installer builds are not relevant to the performance as they are independent installers build which run parallel within the same job to save time instead of duplicating the first stage of the build. These builds run parallel with the webserver builder job. The webserver build takes up the bulk of the time due to its sequential building process of building the server, then the client and finally bundling the total build as displayed in chapter 4.3. The webserver builder job is dependent on the documentation generation job.

The last phase is where Jenkins builds the installer binary file. This job creates an installer from the previous jobs and adds the external software dependencies to the binary file. Here configurations are made to the installer in order to be able to install the components and necessary directories on the to be installed host. These being MongoDB, Python and their software dependencies. This job is dependent on all the previous jobs as it creates the total installer. Upon completion of the job, one of our developers would manually retrieve the installer and deploy it on a virtual machine for testing. Upon the end of the testing phase if the testing is successful the installer will be manually uploaded to a webserver where end-users can download it from. If the testing was unsuccessful then the solution would have to be rebuilt, re-downloaded and re-installed in the same manner after the errors have been ironed out.

5.2 Full automatization with GitLab and Docker

The screenshot shows a GitLab pipeline with two jobs. The first job is 'Build' (ID #9336480) with a 'passed' status, using 'docker' as the provider, and a duration of 00:04:29. The second job is 'Publish' (ID #9336481) with a 'passed' status, using 'docker' as the provider, and a duration of 00:01:37. Both jobs were executed 6 days ago.

Status	Job ID	Name	Coverage
Build			
passed	#9336480	build_executor_image	00:04:29 6 days ago
Publish			
passed	#9336481	publish_executor_image	00:01:37 6 days ago

Figure 10: Executor build and publish GitLab job.

When it comes to the Docker process of building the solution, we would have to look at the execution times of the individual jobs. When it comes to the executor image, we can notice that we have a total time of about 6 minutes to both produce and upload the image to the Artifactory. This however is with a minimal amount of caching taking place. Due to the nature of building the executor we can only leverage a minimal amount of layer caching. When making changes to the executor code base both the stages will have to be rebuilt. The only cached layer that could be used is the installation of the python requirements as the accompanied file with the needed libraries rarely changes and are contained within its own "RUN" instruction. About 1 to 1 minute and a half can be saved with this cache being utilized. The multi-stage build has been mainly introduced in this image to reduce the size of the final image.

The screenshot shows a single GitLab pipeline job named 'Build Images' (ID #9351660) with a 'passed' status, using 'docker' as the provider, and a duration of 00:07:35. It was executed 5 days ago.

Status	Job ID	Name	Coverage
Build Images			
passed	#9351660	build_images	00:07:35 5 days ago

Figure 11: Database and Webserver GitLab build job.

When it comes to the building of the webserver and database image, we can notice that this process is quite a bit shorter than webserver build. Here we are building both the webserver and database image within the same job. The database image taking about 1 minute and the webserver 6 and a half minutes with no caching. However, here the caching will play a big role in the build times. This is dependent on which component of the webserver will have changes committed to the codebase. When only changes are committed to 1 component of the webserver repository then the other component will not have to be rebuilt and the cached version will be utilized. Due to multi-stage building

being optimized for time in this process we are cutting out the time of the shorter stages in fully non-cached builds. The database image will almost always be a fully cached image that will not have changes due to the infrequent need to modify database scripts, this cuts out that 1 minute in most pipeline executions. We witness an average publish time of 1 minute and a half and image builds ranging from 1 and a half to 7 and a half minutes depending on what kind of changes we commit to the repository.

Build Images			
passed	#9351779	build_images	00:01:24 5 days ago
Publish Images			
passed	#9351780	publish_images	00:01:10 5 days ago

Figure 12: Fully cached Database and Webserver GitLab build job with publish stage.

5.3 Build time comparison

The Jenkins job merely handles the building of the installer, the storing of the installer is a manual process and is thus difficult to measure accurately as there are many variables that could influence a developer's process in uploading the installer to the webserver for distribution. Meanwhile our containerized solution does handle this process in an automated manner. But when it comes to strictly comparing the build times when using a strict no-cache we can witness a total build time of about 12 minutes in the GitLab pipelines when utilizing one GitLab Runner. However, we have multiple Runners that can be utilized and both pipelines can run on any Runner that is available. Making this pipeline execution process parallel. In this case when doing a full no-cache build on 2 Runners, we would only be subject to wait on the longest build job out of the two. This being the webserver, we would have a maximum wait time of 7.5 minutes.

When building with Jenkins we are always be subject to an entire rebuild of every component making the installer build a consistently long process with very little to no variations in time. When measuring the build time difference, we have theoretical maximum time of 7.5 minutes with our new solution with 2 GitLab Runners compared to a consistent 20 minutes with Jenkins. This makes for a difference of minimum 12.5 minutes or a reduction of a minimum of 62.5 % solely for the build times.

Adding the publish time to the calculation will give us a theoretical maximum time of 9 minutes with an average 1.5 minute publish job included. This difference making for a minimum of a 55 % decrease in time with the new solution. When the Docker cache comes in to play, the time saved can be significantly larger. However, this comparison is not as accurate due to the manual labour required for the installer publishing.

5.4 Containers brings complexity

It's often stated that containerization processes bring unnecessary complications and difficulties to software projects. Especially in the case of large complicated applications as managing various components, dependencies and images can bring difficulty (Bellishree & Deepamala 2020, 4). The thesis displays that this is not always the case, and our project is quite substantial with multiple interlinked components. The time frame of completing the thesis product has displayed that containerization is still feasible within the late stages of a large project. However, the implementation of the containerization could have been a smoother and a continuous on-going process when handled earlier in the project's lifecycle. Here the initial Dockerfiles would be more simplistic and evolve more gradually to its current state. The earlier implementation of containerization would have also led to developers making more conscious decisions when it comes to implementing new features as they would have to keep the isolated nature of the individual components in mind. This would have prevented some of the problems encountered within the thesis. The statement in question is to a degree more correct when it comes to the hosting of the containers. More specifically this would complicate the hosting of the project within the end-user's technology stack as the official support opts to utilize the containers instead of the installer version. When dropping support for the installer version our end-users will need to switch to hosting the containers if they want to be able to utilize future developed features of the software project. In some cases, this could provide somewhat of a challenge if they have little to no experience setting up a Kubernetes environment. Internally we do provide a snap installation of MicroK8s. The end-user could be in need of assistance when it comes to this providing the platform to host the containers. Hosting containerized applications is thus more so difficult than just running an installer. This is the trade-off to be made to be able to leverage more consistency.

5.5 Containers require extra security

A common critique towards the container technologies is the overall security of the solution. Docker specifically has suffered from major problems in relation to security over it's past before companies were willing to give the technology a trail (Muñoz 2019). Often the security of the container images and how they are composed are questioned. A

container image vulnerability is when there is a security risk embedded within a container image. Here a vulnerability can be caused by an installed package or dependency. This is not directly a threat until a container is created from the vulnerable image that will introduce this security risk in a live environment (Bruner 2020).

When solely providing the software as an installer the security process is only applicable to the software itself. Our developers only need to ensure the security of the project when we are providing the software in this manner. Now that we are switching over to containers, we will have to increase our efforts towards the security hardening process. This is because when providing container solutions there are extra factors that incorporate the total security of the application. Developers need to be tentative towards not only the security of the application but also to that of the build pipeline, deployment environment and the container host as Bennet describes (2020). This will bring an increase in research towards the possible vulnerabilities of the security of containers and if they are applicable to our solution. The security hardening process of our application that is the next step towards the productization of the project will have to be expanded to incorporate the scanning of the containers. The scanning of the containers is a much-needed process to reveal possible security issues with the packages installed within containers. If any are found, they will need to be patched as soon as possible.

Thus, containers will bring an increased work need towards security compared to solely providing the software as an installer. This work increase mostly being the checking for security issues and the patching of them of the container environment and hosting solutions.

6 Conclusion

The project has achieved a success on all its goals. The full-stack project has been containerized with image size in mind while optimizing build performance where possible. The pipelines have been put into place which now handle the building and publishing of the Docker images. This effectively removes the publishing responsibility from the developers. The thesis set out to reduce the build time by a minimum of 40 % which has been surpassed by a 62.5 % reduction minimum. The containerization process has endured more problems than anticipated but managed to be completed in a timely fashion. The project has been in development for an already long duration which makes the transition to a containerized solution more difficult.

The end-result of our project is a directly usable product which will be taken into use immediately for both the testing and running of the project. The Jenkins jobs will be taken out of commission. However, this will not be done immediately in-order to offer our end-users a grace period where they can make the necessary accommodations to take the containerized version of the project into use. During this grace period both the installer and containerized version will be provided to the end-users. The decision has already been made to cut support for the installer version of the project after this grace period and to only support the containerized version.

When support for the installer version of the project has been dropped, we will notice a reduction in errors caused by the environment. End-users will only be able to run the project within the containers which includes the environment we created and provided to them. This will reduce environmental errors to only that isolated environment if they occur as opposed to all the possible environments they would deploy on with installers. This will create more time for our developers to focus on the enhancement of the project instead of fixing environmental errors. For our overall time management there could possibly be an increase in time for our development cycle of our sprint and a reduction of the testing cycle if the total time saved over a week consistently gives us testing time to spare.

Thus, the containerization process has and will provide many indirect and direct benefits in our development processes, testing and distribution processes. The thesis well indicates the possibility to introduce container technology within large applications and that this technology is an applicable alternative for traditional methods of hosting and delivery.

6.1 Next steps

The completion of the thesis does not mark the completion of all containerization related topic for the project. The image building process will have to evolve alongside the testing tool's development as significant changes might require processes within the building of the Docker images to be changed as well. The next step for the productization of the testing tool will be the security hardening which can now take fully place with initial containerization being completed. This process can also bring modification to the images as security vulnerabilities might be found within the project or the project's isolated container environment that need to be addressed.

The first future step will be to fully document our hosting solution and create a baseline benchmark of our total solution so that we can provide Helm charts as a template for end-users. In the near future we will see changes being made to our database solution as we will be switching away from MongoDB due its stated licensing changes. This means that the MongoDB image will be replaced completely for a new database solution. Outside of this we will be implementing the MINIO object storage which will introduce a 4th container within our solution to replace the current temporarily shared volume between the executor and the webserver containers.

References

Agesen, O. 2009. VMware software and Hardware Techniques for x86 Virtualization.

URL:

https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/software_hardware_tech_x86_virt.pdf. Accessed: 15 March 2022.

Alpine s.a., About Alpine. URL: <https://alpinelinux.org/about/>. Accessed: 6 April 2022.

Anaqvi, 2019. Introduction to MicroK8s. URL: <https://ubuntu.com/blog/introduction-to-microk8s-part-1-2>. Accessed: 11 May 2022.

Atzmony, A., Aleksandrowicz A. 2022. Docker Registry URL:

<https://www.jfrog.com/confluence/display/JFROG/Docker+Registry>. Accessed: 11 May 2022.

Avishek, R. 2020. Docker Compose. Medium URL:

<https://medium.com/teckdevops/Docker-compose-5bc79778ff8c>. Accessed: 11 May 2022.

Bellishree P. & Deepmala N. 2020. A Survey on Docker Container and its Use Cases.

URL: <https://www.irjet.net/archives/V7/i7/IRJET-V7I7481.pdf>. Accessed: 16 May 2022.

Bennet L. 2020. Container Security requires more than securing your images. URL:

<https://developer.ibm.com/blogs/container-security-requires-more-than-securing-your-images/>. Accessed: 16 May 2022.

Bitnami s.a., MongoDB packaged by Bitnami. URL:

<https://hub.docker.com/r/bitnami/mongodb>. Accessed: 3 April 2022.

Bruner K. 2020. Container Image Security: Beyond Vulnerability Scanning. URL:

<https://cloud.redhat.com/blog/container-image-security-beyond-vulnerability-scanning>. Accessed: 16 May 2022.

Burgdorf, C. 2014, The anatomy of a Git commit. URL:

<https://blog.thoughttram.io/git/2014/11/18/the-anatomy-of-a-git-commit.html>. Accessed: 11 May 2022.

Callaway, T. 2019, Server Side Public License (SSPL) V1. URL: <https://lists.fedoraproject.org/archives/list/devel@lists.fedoraproject.org/thread/IQIOBOGWJ247JGKX2WD6N27TZNZNM6C/>. Accessed: 3 April 2022.

Carey, S. 2022, Demand for software developers doubled in 2021. InfoWorld. URL: <https://www.infoworld.com/article/3654480/demand-for-software-developers-doubled-in-2021.html>. Accessed: 26 April 2022.

Carklin, N. 2021, What are the benefits of Virtual Machines. URL: <https://www.parallels.com/blogs/ras/benefits-virtual-machines/>. Accessed: 11 May 2022.

Dahlitz, F. 2021. Docker essentials: managing dependencies with ease. A gentle introduction to a popular container solution. URL: <https://florian-dahlitz.de/articles/Docker-essentials-managing-dependencies-with-ease>. Accessed: 11 May 2022.

Demchenko, M. 2021, Software Development Life Cycle: A Guide to Phases and Models. URL: <https://ncube.com/blog/software-development-life-cycle-guide>. Accessed: 26 April 2022.

Despa, V. 2021, A Brief Guide to GitLab CI Runners and Executors. Medium. URL: <https://medium.com/devops-with-valentine/a-brief-guide-to-gitlab-ci-runners-and-executors-a81b9b8bf24e>. Accessed: 10 May 2022.

Docker s.a., Docker Overview. URL: <https://docs.Docker.com/get-started/overview/#:~:text=Docker%20uses%20a%20client%2Dserver,to%20a%20remote%20Docker%20daemon>. Accessed: 17 March 2022.

Eder, M. 2015. Hypervisor- vs. Container-based Virtualization. Seminar Future Internet, Technical University Munchen. Germany. URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf. Accessed: 16 March 2022.

Eldrige, I. 2018. What is container orchestration? URL: <https://newrelic.com/blog/best-practices/container-orchestration-explained>. Accessed: 11 May 2022.

Eschweiler, S. 2019. Docker – Beginner’s guide – images and containers. Medium. URL: <https://medium.com/codingthesmartway-com-blog/docker-beginners-guide-part-1-images-containers-6f3507ffc98>. Accessed: 14 May 2022.

Ferrill, P. 2021. A beginner’s guide to a multistage Docker build. TechTarget. URL: <https://www.techtarget.com/searchitoperations/tip/A-beginners-guide-to-a-multistage-Docker-build>. Accessed: 14 May 2022.

Garcia, J. 2020, Alpine, Slim, Stretch, Buster, Jessie, Bullseye – What are the differences in Docker Images. Medium. URL: <https://medium.com/swlh/alpine-slim-stretch-buster-jessie-bullseye-bookworm-what-are-the-differences-in-Docker-62171ed4531d>. Accessed: 19 April 2020.

GNU 2007, GNU General Public License Version 3. URL: <https://www.gnu.org/licenses/gpl-3.0.html>. Accessed: 3 April 2022.

GNU s.a., The GNU C library (g lib). URL: <https://www.gnu.org/software/libc/started.html>. Accessed: 3 April 2022.

Guo, J. 2017 Demystifying containers vs VM-based security: Security in plaintext. URL: <https://cloud.google.com/blog/products/gcp/demystifying-container-vs-vm-based-security-security-in-plaintext>. Accessed: 11 May 2022.

Haakman, W. s.a. Windows containers in a Linux world. URL: <https://intercept.cloud/en/news/windows-containers-in-a-linux-world/#:~:text=The%20biggest%20difference%20is%20the,Docker%20image%20based%20on%20Linux>. Accessed: 19 April 2022.

Hwang, E. 2019. A beginner-friendly explanation of Kubernetes. URL: <https://faun.pub/a-beginner-friendly-explanation-of-kubernetes-b7e7784acdb0>. Accessed: 11 May 2022.

IBM Cloud 2017. A Brief History of Cloud Computing. URL: <https://www.ibm.com/cloud/blog/cloud-computing-history>. Accessed: 15 March 2022.

Idowu T. & Merron D. 2020. Introduction to Kubernetes Helm Charts. URL: <https://www.bmc.com/blogs/kubernetes-helm-charts/>. Accessed: 16 May 2022.

Jayaraman, A. & Rayapudi P. 2012. Master's thesis. Compative study of Virtual Machines Software Packages with Real Operating System. Blenkinge Institute of Technology, Degree in Electrical Engineering. URL: <https://www.diva-portal.org/smash/get/diva2:829210/FULLTEXT01.pdf>. Accessed: 17 March 2022.

Jayawardana, M. 2019. Understanding Dockerfile. Get an in-depth look at the internal make up of a Dockerfile and the commands with it. DZone. URL: <https://dzone.com/articles/understanding-dockerfile>. Accessed: 14 May 2022.

Jfrog s.a., Artifactory URL: <https://jfrog.com/Artifactory/>. Accessed: 19 April 2022.

Levan, M. 2020, 5 Advantages of GitLab CI/CD pipelines. Tech Target URL: <https://www.techtarget.com/searchsoftwarequality/video/5-advantages-of-GitLab-CI-CD-pipelines>. Accessed: 10 May 2022.

Microsoft 2021, Windows and Containers. URL: <https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>. Accessed: 19 April 2022.

MINIO s.a., Multi-Cloud Object Storage. URL: <https://min.io/>. Accessed: 19 April 2022.

MongoDB 2018. Server Side Public License (SSPL). URL: <https://www.mongodb.com/licensing/server-side-public-license>. Accessed: 3 April 2022.

Muñoz, S. 2019. The history of Docker's climb in the container management market. Tech Target URL: <https://www.techtarget.com/searchitoperations/feature/The-history-of-Dockers-climb-in-the-container-management-market>. Accessed: 11 May 2022.

Murillo, K. 2019. Containerization explained: what it is, benefits and applications. URL: <https://www.masterdc.com/blog/what-is-containerization-benefits-explained/>. Accessed: 11 May 2022.

Peltokorpi A. 2021. The benefits of virtualization across the software development pipeline. URL: <http://jultika.oulu.fi/files/nbnfioulu-202106258745.pdf>. Accessed: 26 April 2022.

PEP 656 2021. Platform Tag for Linux Distributions using Musl. URL: <https://peps.python.org/pep-0656/>. Accessed: 3 April 2022.

Powell R. 2021, Benefits of containerization. URL: <https://circleci.com/blog/benefits-of-containerization/>. Accessed: 11 May 2022.

Pulfer, J. 2018. Container Technology: Consistent Deployment and Execution. URL: <https://www.linkedin.com/pulse/container-technology-consistent-deployment-execution-pulfer>. Accessed: 12 May 2022.

Robocorp 2022, Basic Concepts of Robot Framework. URL: <https://robocorp.com/docs/languages-and-frameworks/robot-framework/basics>. Accessed: 20 April 2022.

Rotsaert, G. 2018. Why and how to use Git-LFS. DZone URL: <https://dzone.com/articles/git-lfs-why-and-how-to-use>. Accessed: 10 May 2022.

Santos, L. 2021. What is a Helm Chart? A tutorial for Kubernetes Beginners. URL: <https://www.freecodecamp.org/news/what-is-a-helm-chart-tutorial-for-kubernetes-beginners/>. Accessed: 11 May 2022.

Saurabh 2022. What is Jenkins? Jenkins for Continuous Integration. Edureka. URL: <https://www.edureka.co/blog/what-is-jenkins/#:~:text=Jenkins%20is%20an%20open%2Dsource,to%20obtain%20a%20fresh%20build>. Accessed: 12 May 2022.

Sevat, P. 2021, Optimize your git clone / fetch strategy for CI pipelines. URL: <https://dev.to/patricksevat/optimize-your-git-clone-fetch-strategy-for-ci-pipeline-3pka>. Accessed: 10 May 2022.

Shubheksha, J. 2018, A quick introduction to Docker tags. URL: <https://www.freecodecamp.org/news/an-introduction-to-Docker-tags-9b5395636c2a/>. Accessed: 11 May 2022.

Simic, S. 2019a. Containers vs Virtual Machines (VMs): What's the difference? URL: <https://phoenixnap.com/kb/containers-vs-vms>. Accessed: 10 May 2022.

Simic, S. 2019b, What is a Hypervisor? Types of Hypervisors 1 & 2. URL: <https://phoenixnap.com/kb/what-is-hypervisor-type-1-2#type-1-hypervisor>. Accessed: 16 March 2022.

Singh, B. & Singh, G. 2018. A study on virtualization and hypervisor in cloud computing. International Journal of Computer Science and Mobile applications. URL: <https://ijcsma.com/publications/january2018/V6I102.pdf>. Accessed: 16 March 2022

Turner-Trauring, I. 2020, Using Alpine can make Python Docker builds 50x slower. URL: <https://pythonspeed.com/articles/alpine-Docker-python/>. Accessed: 3 April 2022.

VMware, what is a hypervisor? URL: <https://www.vmware.com/topics/glossary/content/hypervisor.html>. Accessed: 16 March 2022.

Walker, J. 2021. What is Docker's BuildKit and why does it matter? How-To Geek. URL: <https://www.howtogeek.com/devops/what-is-dockers-buildkit-and-why-does-it-matter/>. Accessed: 14 May 2022.

Wallen, J. 2021, Simplifying the mystery: When to use Docker, Docker-Compose, Docker Swarm and Kubernetes. TechRepublic URL: <https://www.techrepublic.com/article/simplifying-the-mystery-when-to-use-Docker-Docker-compose-and-kubernetes/>. Accessed: 11 May 2022.

Wooster, T. 2017, Docker-in-Docker in GitLab runner. Medium. URL: <https://medium.com/@tonywooster/Docker-in-Docker-in-gitlab-runners-220caeb708ca>. Accessed: 10 May 2022.

Yakutovich, A. 2021, GitLab CI: Cache and Artifacts explained by example. URL: <https://dev.to/drakulavich/gitlab-ci-cache-and-artifacts-explained-by-example-2opi>. Accessed: 11 May 2020.

Appendices

Appendix 1. Resulting database Dockerfile

```
FROM bitnami/mongodb:3.6.8

EXPOSE 27017

ENV USER_NAME="mongo" \
     USER_ID=1010 \
     MONGODB_ROOT_PASSWORD="user"

RUN apt-get update && \
     apt-get install dos2unix && \
     groupadd -g ${USER_ID} ${USER_NAME} && \
     useradd --uid ${USER_ID} --gid ${USER_ID} -s \
     /usr/sbin/nologin --no-create-home \
     ${USER_NAME}

COPY *.js *.json import.sh /Docker-entrypoint-initdb.d/

RUN chmod +rx /Docker-entrypoint-initdb.d/*.sh && \
     dos2unix /Docker-entrypoint-initdb.d/import.sh && \
     chown -R ${USER_NAME}:${USER_NAME} /Docker-entrypoint- \
     initdb.d

USER ${USER_NAME}

VOLUME /bitnami/mongodb/
WORKDIR /bitnami/mongodb/
```

Appendix 2. Resulting executor 2-stage Dockerfile

```

FROM python:3.7 AS stage-compiler

ARG version=1

WORKDIR /build
RUN pip install pyminifier
COPY . .
RUN python autocompiler.py && \
    python ./installer/packager.py ${version}

#####

FROM python:3.7-slim AS stage-final

ARG version=1

WORKDIR /executor
ENV USER_NAME="user" \
    USER_ID=1010 \
    USER_HOME="/executor" \
    LOGSTORAGE_IP="10.10.10.10"

RUN apt-get update && \
    apt-get install -y gcc && \
    groupadd -g ${USER_ID} ${USER_NAME} && \
    useradd --uid ${USER_ID} --gid ${USER_ID} -s \
    /usr/sbin/nologin -d \
    ${USER_HOME} ${USER_NAME}

COPY --from=stage-compiler --chown=${USER_ID}:${USER_ID} \
    /build/installer/resources/core/commons/*.whl ./
COPY --chown=${USER_ID}:${USER_ID} requirements*.txt ./

RUN sed -i /*pypiwin32*/d ./requirements_release.txt && \
    pip3 install -r ./requirements_release.txt -r \
    ./requirements_ute.txt && \
    apt-get purge -y --auto-remove gcc && \
    pip3 install *.whl
USER ${USER_NAME}

VOLUME /var/logstorage

EXPOSE 44000-44299

CMD python3 -m executor.framework.remote.remoteserver \
    executor.framework.remote.masterservice 1 -l \
    ${LOGSTORAGE_IP} -d \
    /var/logstorage

```

Appendix 3. Resulting webserver 4-stage Dockerfile

```

FROM node:14.19-stretch AS stage-server-build

COPY custom_entrypoint.sh ./server ./server/
WORKDIR /server

RUN npm install --verbose && \
    npm run build && \
    npm run generate-apidocs && \
    mv -t ./build/ custom_entrypoint.sh ./ssl robot_parser.py
&& \
    mv ../build/apidocs ./build/apidocs && \
    npm install --production

#####

FROM node:14.19-stretch AS stage-client-build

COPY ./client /client
WORKDIR /client

RUN npm install concurrently cross-spawn fs-extra glob javascript-
obfuscator pkg && \
    npm install && \
    npm run build && \
    npm run obfuscate

#####

FROM executor:latest as stage-document-generator

RUN cd docs && \
    pip uninstall -y elasticsearch && \
    pip install elasticsearch==7.13.4 sphinx recommonmark && \
    python3 documentation.py && \
    python3 -m sphinx -c . -b html .. userguide

#####

FROM node:14.19-slim AS stage-final

WORKDIR /build
ENV USER_NAME="user" \
    USER_ID=1010 \
    USER_HOME="/build"

RUN apt-get update && \
    apt-get install -y python3 python3-pip uuid-runtime
libgssapi-krb5-2 && \
    pip3 install robotframework==3.2.2 && \
    groupadd -g ${USER_ID} ${USER_NAME} && \
    useradd --uid ${USER_ID} --gid ${USER_ID} -d ${USER_HOME}
${USER_NAME}

COPY --from=stage-server-build --chown=${USER_ID}:${USER_ID}
/server/build/ .

```

```
COPY --from=stage-client-build --chown=${USER_ID}:${USER_ID}
/build/ .
```

```
RUN chown -R ${USER_ID}:${USER_ID} ${USER_HOME} && \
    chmod +x custom_entrypoint.sh && \
    mkdir -p /etc/license && \
    mkdir -p /usr/local/core/webserver/userdata
```

```
COPY --from=stage-server-build --chown=${USER_ID}:${USER_ID}
/server/node_modules ./node_modules
```

```
EXPOSE 8443
VOLUME /var/logstorage
VOLUME /data
```

```
ENTRYPOINT ["/custom_entrypoint.sh"]
```

Appendix 4. Resulting executor pipeline job

```

image: Docker:19.03.12-git

variables:
  DOCKER_DRIVER: overlay2
  GIT_STRATEGY: none
  IMAGE_NAME: executor
  IMAGE_NAME_REF_TAG: ${IMAGE_NAME}:${CI_COMMIT_SHA}

stages:
  - build
  - publish

build_executor_image:
  before_script:
    - apk update
    - apk add --no-cache git-lfs
    - git-lfs install
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "${SSH_PRIVATE_KEY}" > ~/.ssh/id_rsa
    - chmod 400 ~/.ssh/id_rsa
    - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" >
~/ssh/config
    - chmod 400 ~/.ssh/config
    - ls -l
    - git clone ${CI_REPOSITORY_URL} ${CI_PROJECT_DIR}
    - cd ${CI_PROJECT_DIR}
    - git checkout ${CI_COMMIT_SHA}
  tags:
    - Docker
  image: Docker:19.03.12-git
  stage: build
  services:
    - Docker:19.03.12-dind
  variables:
    DOCKER_TLS_CERTDIR: ""

  script:
    - export DOCKER_BUILDKIT=1
    - Docker image build --build-arg version=${CI_PIPELINE_IID} .
-t ${IMAGE_NAME_REF_TAG}
    - mkdir executor
    - Docker save ${IMAGE_NAME_REF_TAG} >
executor/${IMAGE_NAME_REF_TAG}.tar
    - Docker history ${IMAGE_NAME_REF_TAG}
  artifacts:
    paths:
      - executor
    expire_in: 30 mins

publish_executor_image:
  tags:
    - Docker
  image: Docker:19.03.12-git
  services:

```

```
- Docker:19.03.12-dind
stage: publish
script:
  - Docker login -u ${ART_USER} -p ${ART_PASS}
local.Artifactory.com
  - Docker tag ${IMAGE_NAME_REF_TAG} local.Artifactory.com
/${IMAGE_NAME_REF_TAG}
  - Docker push local.Artifactory.com /${IMAGE_NAME_REF_TAG}
  - Docker image rm ${IMAGE_NAME_REF_TAG}
dependencies:
  - build_executor_image
```


Appendix 5. Resulting webserver and database pipeline job

```

image: docker:19.03.12-git

variables:
  DOCKER_DRIVER: overlay2
  GIT_STRATEGY: none
  WEB_IMAGE_NAME: web
  WEB_IMAGE_NAME_REF_TAG: ${WEB_IMAGE_NAME}:${CI_COMMIT_SHA}
  DB_IMAGE_NAME: mongo
  DB_IMAGE_NAME_REF_TAG: ${DB_IMAGE_NAME}:${CI_COMMIT_SHA}

stages:
  - build images
  - publish images

build_images:
  before_script:
    - apk update
    - apk add --no-cache git-lfs
    - git-lfs install
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "${SSH_PRIVATE_KEY}" > ~/.ssh/id_rsa
    - chmod 400 ~/.ssh/id_rsa
    - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" >
~/\.ssh/config
    - chmod 400 ~/.ssh/config
    - ls -l
    - git clone ${CI_REPOSITORY_URL} ${CI_PROJECT_DIR}
    - cd ${CI_PROJECT_DIR}
    - git checkout ${CI_COMMIT_SHA}
  image: docker:19.03.12-git
  stage: build images
  tags:
    - docker
  services:
    - docker:19.03.12-dind
  variables:
    DOCKER_TLS_CERTDIR: ""
  script:
    - export DOCKER_BUILDKIT=1
    - ls -l
    - cd mongo
    - docker login -u ${ART_USER} -p ${ART_PASS}
local.Artifactory.com
    - docker build -f Dockerfile . -t ${DB_IMAGE_NAME_REF_TAG}
    - cd ..
    - mkdir db
    - docker save ${DB_IMAGE_NAME_REF_TAG} >
db/${DB_IMAGE_NAME_REF_TAG}.tar
    - ls -l db
    - docker history ${DB_IMAGE_NAME_REF_TAG}
    - docker image build . -t ${WEB_IMAGE_NAME_REF_TAG}
    - mkdir web
    - docker save ${WEB_IMAGE_NAME_REF_TAG} >
web/${WEB_IMAGE_NAME_REF_TAG}.tar

```

```
- ls -l web
- docker history ${WEB_IMAGE_NAME_REF_TAG}
artifacts:
  paths:
    - db
    - web
  expire_in: 30 mins

publish_images:
  tags:
    - docker
  image: docker:19.03.12-git
  services:
    - docker:19.03.12-dind
  stage: publish images
  script:
    - docker login -u ${ART_USER} -p ${ART_PASS}
local.Artifactory.com
  - docker tag ${WEB_IMAGE_NAME_REF_TAG}
local.Artifactory.com/${WEB_IMAGE_NAME_REF_TAG}
  - docker push local.Artifactory.com/${WEB_IMAGE_NAME_REF_TAG}
  - docker tag ${DB_IMAGE_NAME_REF_TAG}
local.Artifactory.com/${DB_IMAGE_NAME_REF_TAG}
  - docker push local.Artifactory.com/${DB_IMAGE_NAME_REF_TAG}
  - docker image rm ${WEB_IMAGE_NAME_REF_TAG}
  - docker image rm ${DB_IMAGE_NAME_REF_TAG}

dependencies:
  - build_images
```

Appendix 6. Jenkins build trend

Build Time Trend

Build	Duration	Agent
#384	20 min	master
#383	20 min	master
#382	16 min	master
#381	7 min 56 sec	master
#380	15 min	master
#379	16 min	master
#378	15 min	master
#377	7 min 55 sec	master
#376	15 min	master
#375	14 min	master

