Expertise
and insight
for the future

Mikko Dittmer

GitLab Project management

Metropolia University of Applied Sciences

Batchelor of Engineering

Information Technology

Thesis

29 May 2019

Author(s) Title Number of Pages Date	Mikko Dittmer GitLab Project management 22 pages + 7 appendices 29 May 2019
Degree	Batchelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Janne Salonen, Head of Department

This thesis will demostrate how GitLab and Continuous integration can be used to develop a software project. Automating the creation of Docker images and exploring the mechanisms with which these can be managed.

Keywords	GitLab, Docker, DevOps, Git

Contents

1 Introduction

While programming can be seen as a puzzle needed to be solved or even an art form, working on a larger software project requires a structured process. The underlying technologies on which software is written for is constantly changing and evolving. Keeping up with deprecated dependencies, security issues, new languages, frameworks and deployment methods can become overwhelming to maintain. While scaling applications to ever increasing workloads comes with it’s own set of problems.

Automation and virtualization is set to help solve these issues. Automating as much as possible can reduce workloads for personnel, keep the codebase and underlying system uptodate and enables more streamlined monitoring and managing toolset to be used. Having a good overview of how the system is functioning, monitoring performance, cost and development cycles can be crucial for the success of a project and the organization. Many organizations have come to realize the value of a proper development and relase infrastructure, while some still struggle to release software in a structured, tested and mindful process. Thinking of the cloud as a solution to many development and deployment problems might not be a good idea, without the proper tools to manage such projects.

To find out how automation could help in software development, an example project was created, built, tested and deployed.

The idea was to create two very different deployment targets. An embedded device which requires a distinct toolchain and whose deployment might require some hardware access, and an web service, with a widely used deployment target and a lot of support to create applications that might scale to very large deployments.

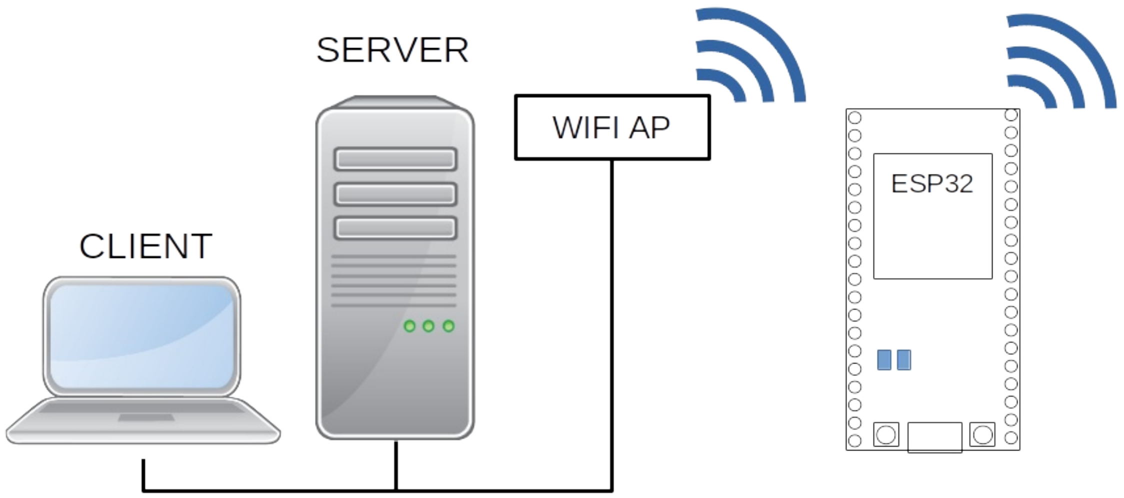


Figure 1. The service which will be created.

As shown in Figure 1. the final system consists of a Server and an embedded device. The system will automatically build and deploy software to both these targets. A client will then be able to connect to the service using a web browser.

2 Deployment targets

2.1 ESP32

The ESP32 WROOM is a SOC designed for low power, network connected applications i.e. IOT devices. Featuring a dual core processor, low-power co-processor, WI-FI, Bluetooth, on chip RAM, built in Flash memory and a variety of i/o options. [1]

Compiling programs for the ESP32 on the x86 architecture will require cross compilation, as the program will be compiled from on a different architecture from that of the target device. Here the chip manufacturer provided the necessary toolchain, which is open source and was generated with Crosstool-NG. [2]

In addition to the toolchain, the Espressif IoT Development Framework (IDF) was used. It includes a modified for symmetric multiprocessing (SMP) version of the real-time operating system FreeRTOS [3]. It also includes many useful libraries to accelerate development and enabling usage of the hardware features the device has to offer. There are also example programs with varying open source licences.

The examples included in the framework include various implementations working with networking and peripheral interfaces. The test program will be based on an example, which C code file is licensed under a public domain license. Libraries and other support files are also under a open source license, with the Apache license being common.

GNU Make is used as the build automation tool. [4] It determines automatically which pieces of code need to be compiled and issues commands to compile them. The provided Makefile includes the instructions for make, which includes rules for building the application, then flashing the partition table, bootloader and application on to the chip.

There are many messaging protocols the esp32 device could use. A widely used standard for IOT projects is MQTT. MQTT is a publish/subscribe messaging transport protocol, where devices can subscribe/publish to a messaging feed. MQTT works on top of TCP/IP. The messages go through a broker. [5] Many public brokers exist, or one could setup a self hosted one, with many open source brokers existing such as Mosquitto. [6] There are mqtt examples available in the espressif IDF.

The firmware coded in the C-language for this project uses http. Based on the IDF http example, the example was trimmed and modified and now implements http GET and POST protocols. The program will initialize a WIFI connection and start listening on the standard http port 80. A message parser was added so we can receive and send commands to the device. There is an onboard LED which state can be read or written using POST requests, from either three states, ON, OFF or BLINK. ON and OFF raise or lower the GPIO (General Purpose Input Output) ports, while BLINK creates a Task in FreeRTOS which has a loop turning the port input high and low with a delay. The Task is then destroyed if the light should no longer blink.

Full source code is listed in appendix 2.

2.2 Linux Servers

2.2.1 Disk Images

A disk image is a file which contains a bit by bit representation of the data which would be on a disk drive. This image can then be used and manipulated with the appropriate utilities as if it was an disk block device.

Creating an operating system can be made with a simple shell script. It is possible to create an disk image file, partition and format it. The image can then be mounted to a mountpoint.

For Debian based operating systems we can use debootstrap to copy a base system on to the image. Debootstrap also allows the installation of any additional required packages, such as the kernel and a bootloader. [7] It is also possible to chroot into the image filesystem. Chroot will change the processes and it’s children filesystem root directory. [8] This makes it possible to run commands as if it were the image machine and configure the system. This is not to be confused with containerization or virtualization as the only thing that is changed is the filesystem apparent root directory. The chrooted environment cannot trivially change files outside of the designated directory tree. Using these techniques should have generated an disk image which can be run on either Bare Metal or in an Virtual Machine. See Appendix 1 for an example.

2.2.2 Openstack

Openstack is a solution to deploy software on self hosted hardware. It is a collection of projects which can control compute, storage and networking resources in a datacenter. Applications and services could be better deployed and managed with openstack. Containers, Virtual Machines and even Bare Metal can be provisioned. Glance could be used to store System Images, Nova and Ironic could be used to provision virtual machines and bare metal.

Openstack can be used to manage and deploy system images in a more controlled manner as the Disk Image method above. This however wont be implemented for this project, due to resource limitations.

2.3 Virtual Machines

Virtual machines are a very good solution to save costs by sharing a single physical server. Many different types of virtualization solutions exist. XEN and KVM are two of the better known open source hypervisors. For this project KVM was chosen.

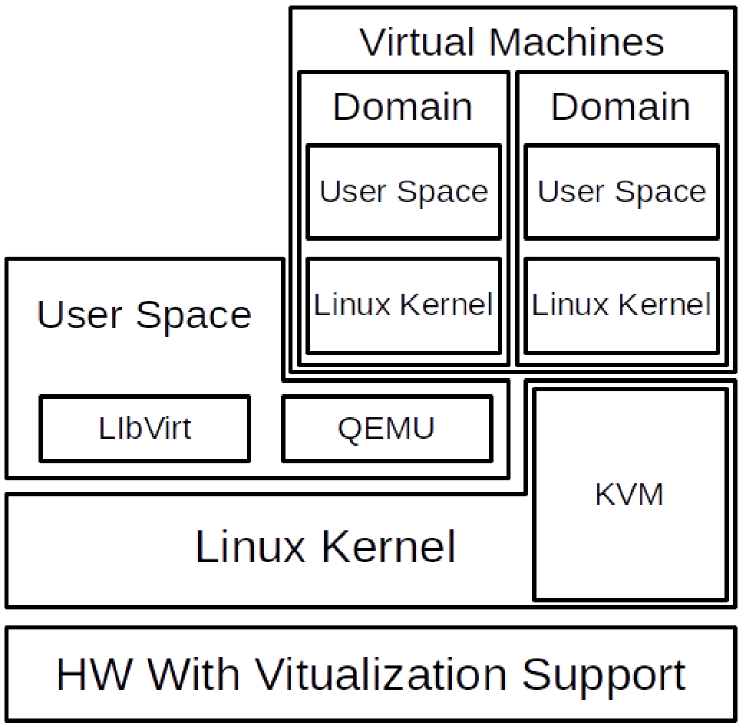


Figure 2. Virtual machine overview

Virtual machines run on top of a hypervisor, as shown in Figure 2. These Virtual Machines (VM) will have a slice of the servers resources to use. This is controlled by the hypervisor and it’s userspace counterparts. KVM is one such hypervisor. [9]

2.3.1 KVM

KVM provides hardware assisted virtualization and requires a supported CPU. For x86 this means that the processor should support either VT-x for intel CPUs and AMD-V for AMD CPUs. The kernel component of kvm is included in mainline Linux. The userspace component is included in qemu. [10]

KVM can give a VM access to the servers resources using a kernel component as seen in Figure 2. Each VM runs its own kernel. User space is where all applications are run, in other words everything outside of the kernel. This includes any application that will be run, starting with the init system.

2.3.2 QEMU

QEMU is a machine emulator and virtualizer. If used with KVM as a virtualizer, virtual hosts can execute code directly on the host cpu increasing performance greatly compared to full virtualization.

When used as an machine emulator, QEMU can be used to emulate an entirely different architecture. This could be very useful if some hardware is not readily available or, if some peace of software is needed to be compiled for an architecture where compilation on the target device would otherwise be problematic. [11]

2.3.3 LibVirt

Libvirt is a toolkit to manage virtualization platforms. While QEMU could be used to manage the virtual machines, libvirt has some tools which makes the process somewhat simpler. This also enables the usage of virt-manager which is a GUI for small scale deployments of virtual machines, and makes monitoring and debugging easier.

2.4 Docker

Docker enables applications to be packaged in discrete images which can be deployed and executed in a consistent manner as a container. This is possible as each image can be built to contain the necessary dependencies required. As the image is consistent, it can be developed and tested without each developer or development step having a different environment, which would eventually cause errors.

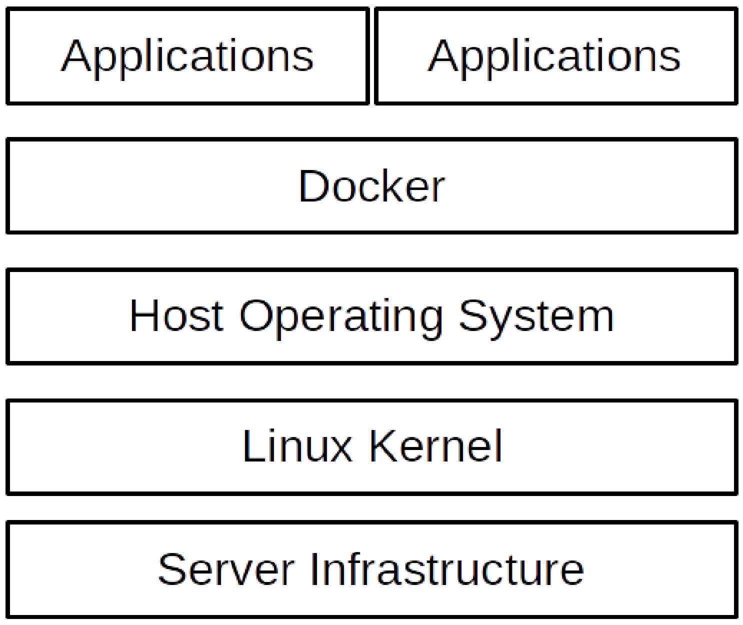


Figure 3. Docker overview

The docker system will run on top of the host operating system as seen in Figure 3. The kernel is shared with the host operating system and other containers. Resources can be shared more freely than the more complete isolation of KVM, and a full operating system is not necessarily required. This can be seen in comparison of KVM depicted in Figure 2 and Docker depicted in Figure 3. Docker uses Linux namespaces to provide isolated environments. Control groups (cgroups) are usd to provision system resources, such as maximum memory consumption and cpu usage.

2.4.1 Dockerfile

The Dockerfile describes how the docker image should be built. It is a text file which is read, in order.

A Base image is defined with an FROM statement, which is the basis on which the rest of the image is built. These images could be built from scratch, but many images are available prebuilt. In addition to distribution images, projects such as php, sql or Apache have their own official base images to build upon.

On this base image layer it is possible to COPY files to, or RUN commands and scripts, use ENV to set environment variables etc. This is how a lot of time can be saved, as an image can be built on top of a base image already having base dependencies met.

2.4.2 Images

Docker images are built from a series of layers. As seen in figure 4. Each command in a Dockerfile creates a new layer. Layers only contain a set of differences from the layer before it. [12]

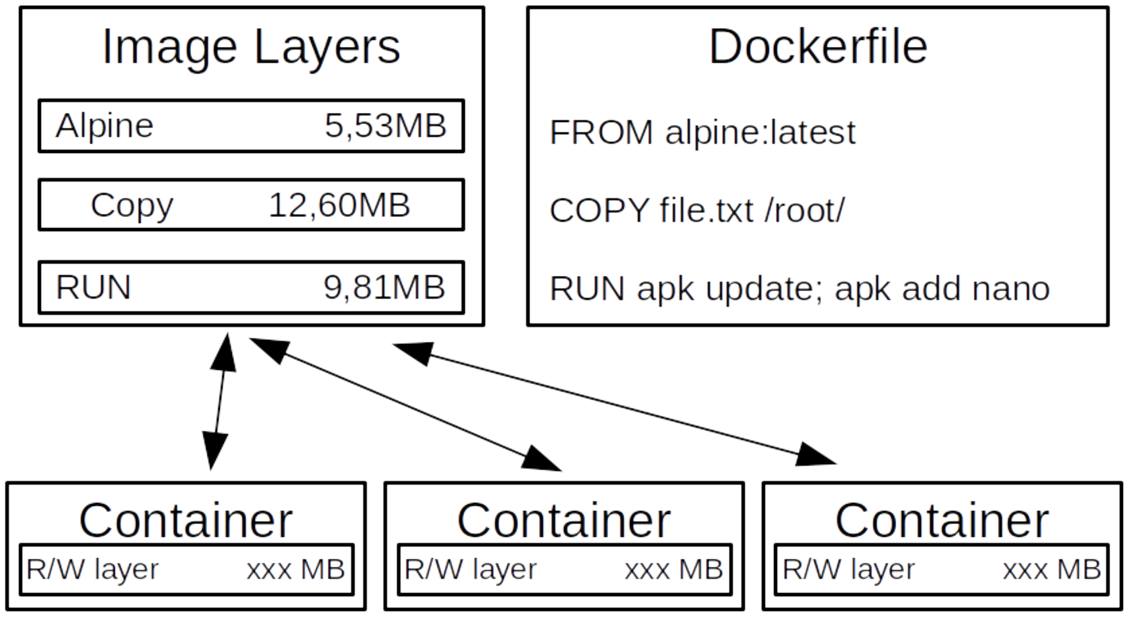


Figure 4. Docker layers

A small base image without unnecessary “bloat” is therefore a good idea as each image can add any necessary packages or files on a separate layer. Alpine linux is one such image which is widely used for docker and it’s base image is a little over 5MB in size.

2.4.3 Container

When a container is created from an image, a writable layer is added to the layer. Any changes made to the container are stored there. If the container is removed this data is lost. However It is possible to use volumes, which can hold persistent data for a container or sets of containers. These are managed by docker. Another method is using bind mounts which uses the underlying filesystem. It is then also be possible to store any of the persistent data on a network filesystem, such as NFS.

The container also contains all necessary information on how to run the system. This information is created from the Dockerfile and the build process. This may include in, addition to used images and volumes, settings for Networking, environment parameters, entrypoints, CPU, Memory limits etc.

2.4.4 Registry

A Registry is a space to store Docker images. There is a public Docker Hub registry where many project will publish prebuilt images of their projects. [13] This is the default registry when using docker. If for instance the first line of a Dockerfile is “FROM php:apache” and it isn’t yet stored locally, Docker will attempt to fetch this image from the Docker Hub registry. Now the official php image with Apache can be used.

Security should be a concern when using any software and pulling images from unknowns sources should be avoided. There have been incidents of malicious Images being distributed so having an extra entity in the middle of the chain might not turn out to be a good thing. [14]

GitLab also includes it’s own registry service where any generated images for the projects can be stored and retrieved. These images consist of the layers described earlier, so only changes from the lower layers need to be stored instead of full images every time a project is rebuilt. [15]

3 DevOps

Development Operations (DevOps) is a vague term [16] describing an operational model which includes software development, quality assurance and technology operations. [17] It might also include building the underlying infrastructure and procedures for the workflow of the development process.

As the problem of a disconnect between between IT functions was recognized the DevOps model aims to decrease the time required for the developent, testing and deployment cycle. While traditionally poor communication and conflicts between teams meant infrequent releases, DevOps might have releases several times a day. [18]

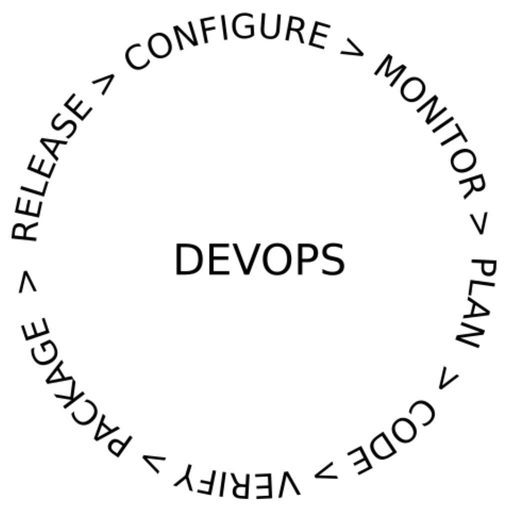

DEVOPS
CONFIGURE > MONITOR > PLAN > CODE > VERIFY > PACKAGE > RELEASE >

Figure 5. DevOps Cycle

Figure 5. shows the DevOps release cycle. [19] Much of the cycle will be automated, but will haveto be setup for the project. It is a continuous process lasting the lifetime of the project.

3.1 Continuous Integration and Deployment

Continuous Integration and Deployment (CI/CD) is creating software in a automated environment. Changes to the code are run through a build and testing pipeline, and with Continuous Deployment releases are also automated. [20]

Testing code often for errors makes fixing issues faster, as pinpointing the right commit which change caused the issue can be determined quickly. Having the CI system also build the underlying platform, ensures that the software behaves consistently with all the necessary dependencies when deployed. Builds and tests can be run on separate servers which can speed up build and testing times.

Several CI/CD systems exist such as GItlab-CI, Hudson, Jenkins, Buildbot, Drone and many hosted services. Hosted services can be a good option if data can be processed and stored in third party servers. Open source projects benefit greatly from hosted services for which some providers will give access at no cost.

Self hosted systems are required if data needs to be kept within the organization, or tight integration with a company existing infrastructure is required. It can in some cases also be more cost effective. For this project a self hosted GitLab-CI system will be used.

CI/CD systems have a file which tells what the system should do with the project. This description creates the pipeline. Drone has a .drone.yml file, Jenkins has a Jenkinsfile and GitLab-CI uses an .gitlab-ci.yml file. While the file syntax differs somewhat, they all describe the same thing: what should be build, what should be tested and what should be deployed. These pipelines can have many stages, some running in parallel and doing different tasks with different code branches.

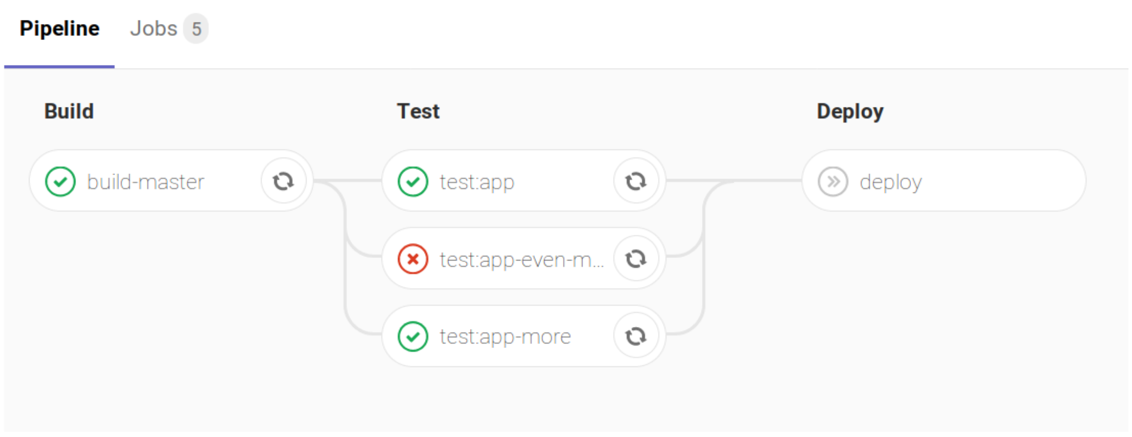


Figure 6. CI/CD pipeline

As can be seen in Figure 6. GitLab can run many build and test environments simultaneously, these are called “runners”. There are also different runner types. A shell runner runs all the tasks as shell scripts that can be pre installed with all the necessary dependencies necessary to build and/or test the software. Another type is running it with docker. Docker can be used in multiple ways, having docker start containers in parallel or running docker containers inside docker containers. GitLab also supports managing runners using kubernetes, which may result in an docker image or some other kubernetes supported virtualization/container setup. [21]

Using Kubernetes or Docker runners is a good way to ensure that a consistent, clean, environment is created each time. Different caching methods can be used to speed up the build process. However using a shell runner makes it easy to use the same underlying system quickly, without the need to reload or rebuild docker images. It also makes it easier to interact with the hardware directly if it becomes necessary.

4 Software Management tools

4.1 GIT

Git started as the Linux kernel developers needed a replacement for a proprietary source control management system, which didn’t meet the developers needs. [22] Git is a open source distributed version control system, and has since become widely used for many types of projects.

It was designed to make developing software in a highly distributed environment more manageable, making merging of development branches routine. Git projects usually contain multiple branches, which can then be merged into a single branch, aiding in parallel development. [23]

A master branch could for instance have a development branch, from which in turn developers could create their own branches to work on. This makes cooperation even on large projects possible. Different branches can be worked on separately. When merging changes, possible conflict handling between changes can be analyzed using GIT.

4.2 GitLab

GitLab is an application for the software development lifecycle. It is built to manage the entirety of a software project, including source code management, project management, Quality Assurance and Release management. GitLab and it’s toolchain can support the entire “DevOps” cycle. [19]

There are multiple variants available. There is a hosted commercial service, which offers free accounts to open source projects. There are also self hosted variants. The Enterprise Edition with commercial support and additional features and a Community Edition which is free and Licensed under the open source MIT license. The variant described in this project is the open source, self hosted one.

The toolchain has many individual services which can be enabled or disabled depending on the requirements. Some of the tools are replaceable with other tools or integrated with existing infrastructure.

Source code management, using GIT, is a cornerstone of the system. Developers can work with any GIT client they're used to, or more likely use their preferred Integrated Development Environment (IDE) which is bound to have some sort of GIT integration. Authentication is handled using ssh keys. Every user will use the git user in the ssh client. Authentication and authorization is then determined by the ssh key.

Continuous Integration/Deployment (CI/CD), creates a pipeline and runs builds and tests on runners. Continuous Integration reduces errors as code is tested often, potentially on every push of new code. With this system the process can be highly automated saving time and effort compared to manual systems. Code can then be released with little to no intervention if all the tests pass. This will require well written tests to be useful. GitLab CI is not the only option as integrations to other CI systems, such as Jenkins are available.

Managing projects is central to GitLab. Issue Tracking is integrated with many of the other tools gitlab has to offer. Referencing to commits, making merge requests or using issues to analyze progress is possible. External issue trackers can also be used, such as Redmine, Bugzilla and many others.

There are many ways to communicate, share ideas and code. Documentation can be created using the builtin wiki, if it is not desireable to store it in the code repository. Snippets is a service which can store bits of code or text with other users. It is also possible to host static web pages using GitLab pages, which can be a convenient way to host a dedicated project website.

Prometheus a time-series monitoring service can be used for monitoring. And mattermost can be used as a messaging platform. A web IDE can be used to program directly using the web interface, or to do quick fixes. There are tools for analytics and time tracking.

In other words GitLab contains many tools to create, build, monitor and manage a project. And this is done in a way where operations team are not forced to use the built in tools.

This project utilizes the GIT source code management and the CI/CD features.

5 Building the system

The development system was built using commodity hardware consisting of of two physical servers, a firewall, wireless access point and a network switch (Fig. 7). Most of the services are run on virtual machines.

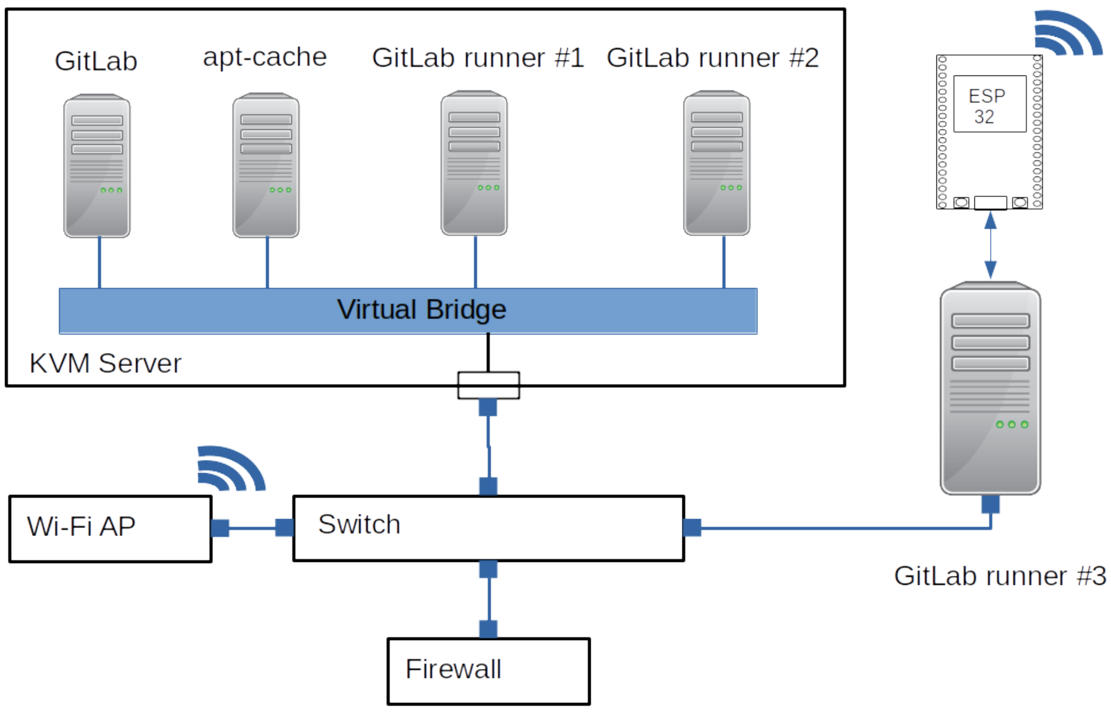


Figure 7. The complete network

The main KVM server hosting the Virtual Machines is using Alpine Linux for the base system. Alpine Linux was chosen for its efficient size, ease of configuration and stability. A virtual bridge was configured to attach to the physical Network Interface Controller (NIC). When the Virtual Machines start up they will attach themselves to this bridge. This makes network configuration and routing very simple. The required packages for running the virtual machines were installed using the apk package manager.

5.1 GitLab Installation

GitLab can be installed in a multitude of ways. While Docker images are available as an official installation method, the omnibus packages are recommended. An KVM virtual server was created for GitLab and Ubuntu 18.04 was installed on the VM. Official GitLab “Omnibus” package was then installed.

As it was desirable to use use the encrypted https protocol, a letsencrypt ssl certificate was needed. GitLab can be configured to fetch and install the certificate automatically using the free letsencrypt service. However this would need the GitLab instance to be connected directly to the internet, but access is only available through an VPN server. Therefore the certificate was created on a separate server and copied over to the GitLab instance. The GitLab server was also configured to be able to send mail, trough an pre-existing mail server. This enables GitLab to send notifications.

The setup is done mostly in a gitlab.rb configuration file. In the gitlab.rb configuration file the servers url had to be set, as well as the path to the ssl certificates for the nginx http server. Also a docker registry was configured to the same GitLab instance, which only needed to have it’s url and port set.

5.2 GitLab runners

Multiple GitLab runners were installed on virtual servers as well as a shell runner on an dedicated server. Different operating systems, deployment methods and runner types were tested.

5.2.1 GitLab Runner 1

A Virtual machine was created on the KVM server and Alpine Linux was installed as the operating system. Docker service was then installed, from the Alpine Linux community repository.

The GitLab Runner image was then pulled from Docker Hub, and a bind mount was configured to store the configuration data. The docker image was registered to the GitLab server using the gitlab-runner command included in the Docker image. The runner was then ready do accept tasks.

5.2.2 GitLab Runner 2

As with the other Runners a Virtual machine was created, however Debian GNU/Linux 9 stretch, was installed along with the Docker package from the official docker repository. GitLab runner was installed manually as a shell runner.

A new docker network was created using the macvlan driver. This allows a Docker container using this network to directly connect to the development network through the virtual bridge on the KVM server.

Using this setup shell scripts can be written to start Docker containers.

5.2.3 GitLab Runner 3

Yet another runner was created on a single stand alone server. This time using Debian as the operating system. This system was created using a shell script, which installed the GitLab runner environment and the ESP32 toolchain and development framework. The image was then copied to a flash drive which was used as the system disk. This runner was also registered as a shell runner.

The ESP32 is connected to the runner and is accessible to the build system using the ”/dev/ttyUSB0” device.

5.2.4 Apt-cache

A apt-cache server was created as a Virtual Machine. When installing software packages automatically, scripts may have to fetch the same package each time a build is run. This creates a lot of traffic for the software providers and can be quite slow. Therefore is makes sense to have a cache where software repositories can be downloaded once and then served locally. Debian based systems use the apt (Advanced Package Tool) and has software available which will cache packages being pulled from the repository. The package apt-cacher-ng was installed which can not only cache packages for debian based systems, but others too. [24]

6 Creating a project

Each GitLab project belongs to a group or a subgroup. Each project has it's own URL. Projects can be Private, Internal or Public. Private projects are only accessible to the users in the projects group. This group might consist of a single user, group or subgroup. Internal projects are accessible to any logged in user. Public projects are accessible to be read by anyone.

Projects can be created from scratch or populated using templates. Projects can also be imported from other services, such as other GitLab deployments, Google Code, GitHub or any git repository with an URL.

A project can then be accessed from an url. For instance, if we have a group named testers, and a project named testproject a url might have an address such as https://gitlab.example.com/testers/testproject and the project can be cloned using the address https://gitlab.example.com/testers/testproject.git

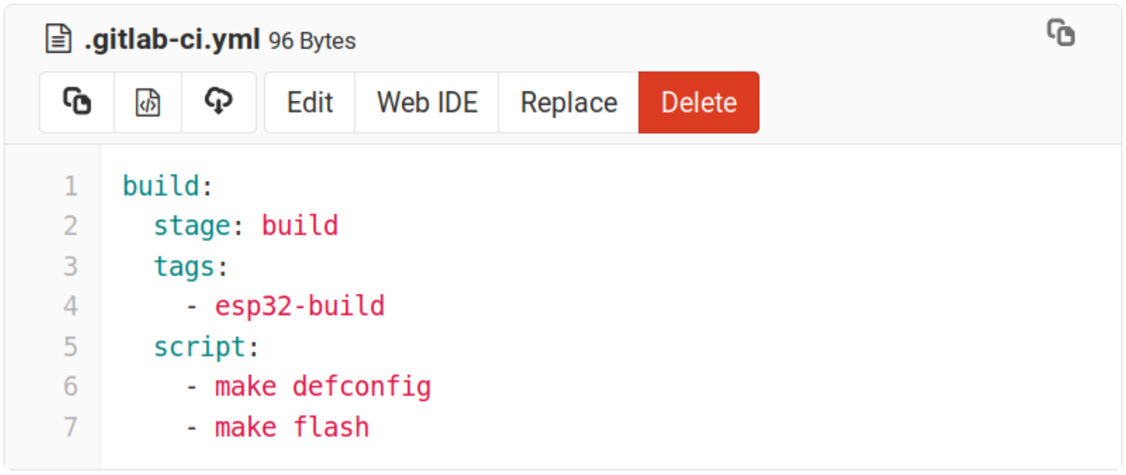
6.1 Programming the ESP32

A blank project was created in GitLab.

The esp32 C program was programmed using the example program from the espressif IDF examples as a template. The program consist of a single C file and the IDF framework. The program is included in Appendix 2.

Configuration files for the make build system were created. These include the project name and the instructions to use the makefile of the IDF framework. As the makefile is preconfigured there is no need to worry about compile options or memory positions for the firmware write process.

Multiple build commands can be given using the provided Makefile. Running the command ”make menuconfig” would run a interactive configuration tool, where various elements for the project using the IDF framework can be configured. Settings for wireless networking, which serial port to use when flashing etc. can be configured here. However as we want to run the configuration non-interacively the ”make defconfig” command will create a configuration file using default values. Settings such as wireless networking can be set at buildtime or overwritten in the C code. Running the command ”make flash” will build the software and flash the software onto the chip.



```yaml
build:
  stage: build
  tags:
    - esp32-build
  script:
    - make defconfig
    - make flash
```

Figure 8. gitlab-ci.yml file contents

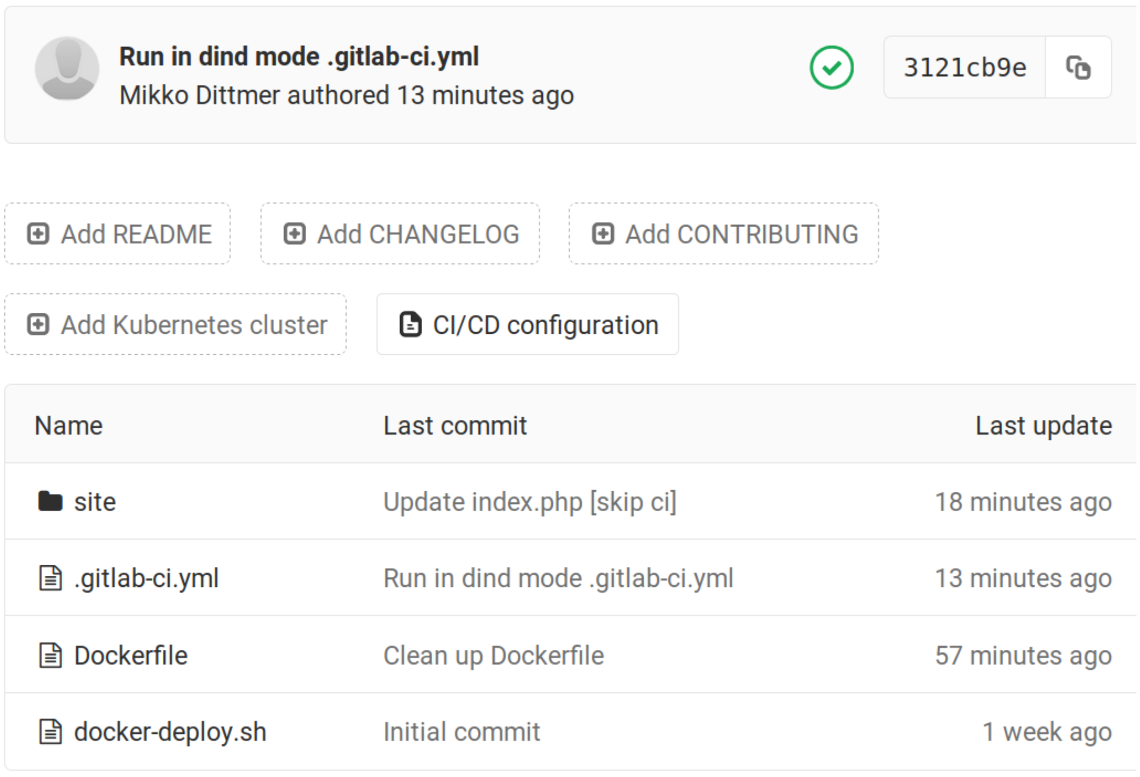
Finally a GitLab CI file was then created instructing what should be done when changes are made to the git master branch. As can be seen in Figure 8. it only has a build stage. The tags value instructs only to run the build with a runner with esp32-buid tag attached to it. Then it creates a configuration from the Makefile using ”make defconfig” and finally builds and flashes the crated binary to the chip.

What can also be done is to set a ”artifacts” path. This will store files or directories set in this path to an artifacts package. This would then store whatever binaries or build data that is required.

6.2 PHP Project in GitLab

A http site was written with an html and php files. The php uses curl to send values to a hardcoded address using the http POST method. The ESP32 will then listen to the request and process the data and send a status update back to the php server. The web content portion is stored under ”site” directory in the project.

The php programs project file listing can be seen in Figure 9. This is includes everything that is needed to build a small php project.


Run in dind mode .gitlab-ci.yml
Mikko Dittmer authored 13 minutes ago

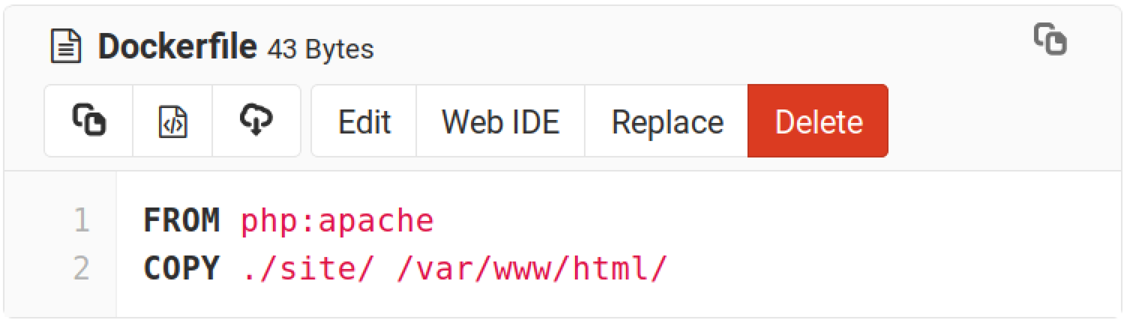
3121cb9e

Add README | Add CHANGELOG | Add CONTRIBUTING
Add Kubernetes cluster | CI/CD configuration

Name	Last commit	Last update
site	Update index.php [skip ci]	18 minutes ago
.gitlab-ci.yml	Run in dind mode .gitlab-ci.yml	13 minutes ago
Dockerfile	Clean up Dockerfile	57 minutes ago
docker-deploy.sh	Initial commit	1 week ago

Figure 9. File listing of the php project

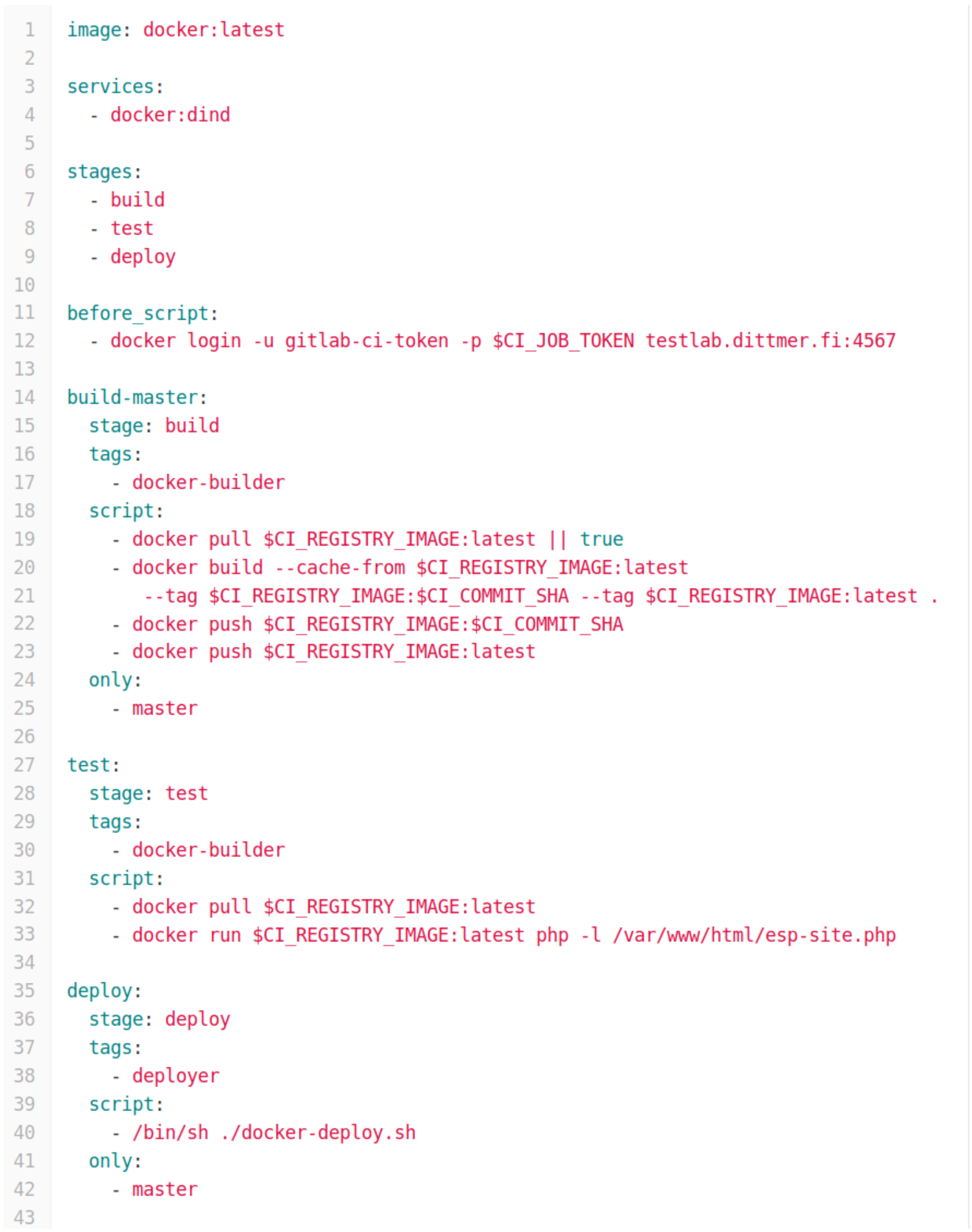
This would then be packaged into a Docker image. For this a very simple Dockerfile was needed. The Docker file seen in Figure 10, has only two lines. It fetches a ”php:apache” docker image, which already includes everything needed to run the php application. The Apache HTTP daemon, PHP, the PHP module for Apache webserver and curl are all included.



```
FROM php:apache
COPY ./site/ /var/www/html/
```

Figure 10. Dockerfile

All that was needed to do was to copy the content that needed to be served to a predefined directory. This directory was documented in the Docker Hub page where the image was pulled from.



```
 1  image: docker:latest
 2
 3  services:
 4    - docker:dind
 5
 6  stages:
 7    - build
 8    - test
 9    - deploy
10
11  before_script:
12    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN testlab.dittmer.fi:4567
13
14  build-master:
15    stage: build
16    tags:
17      - docker-builder
18    script:
19      - docker pull $CI_REGISTRY_IMAGE:latest || true
20      - docker build --cache-from $CI_REGISTRY_IMAGE:latest
21         --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA --tag $CI_REGISTRY_IMAGE:latest .
22      - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
23      - docker push $CI_REGISTRY_IMAGE:latest
24    only:
25      - master
26
27  test:
28    stage: test
29    tags:
30      - docker-builder
31    script:
32      - docker pull $CI_REGISTRY_IMAGE:latest
33      - docker run $CI_REGISTRY_IMAGE:latest php -l /var/www/html/esp-site.php
34
35  deploy:
36    stage: deploy
37    tags:
38      - deployer
39    script:
40      - /bin/sh ./docker-deploy.sh
41    only:
42      - master
43
```

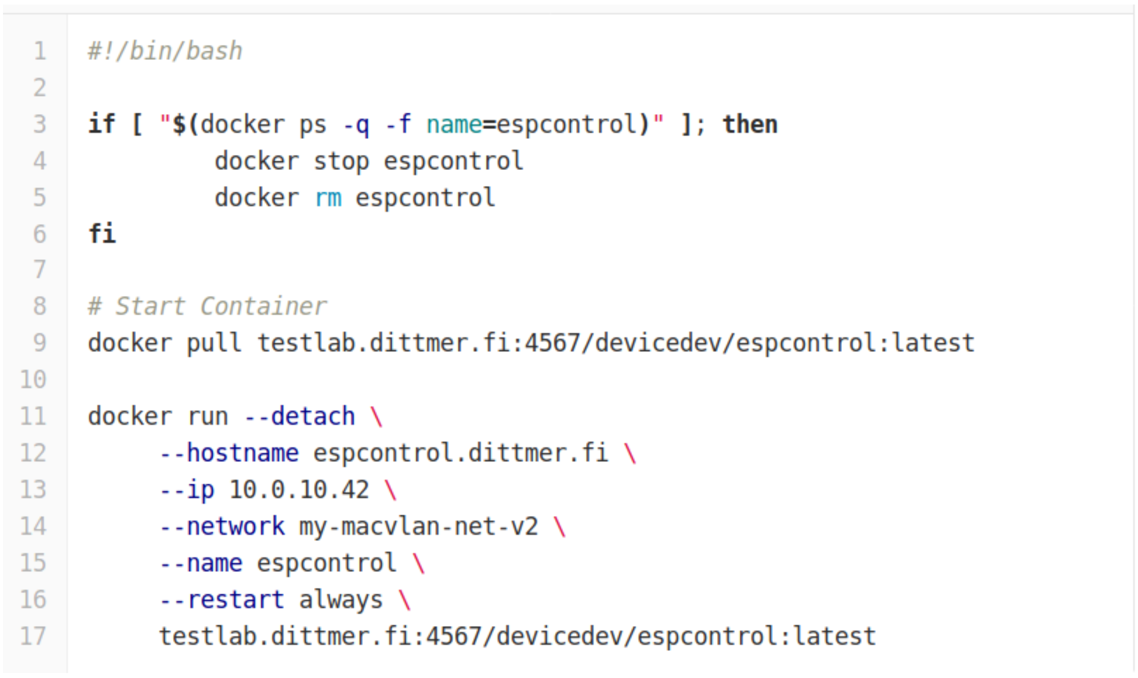
Figure 11. gitlab-ci.yml for a php project

The GitLab CI file as seen in Figure 11, is run in an docker in docker environment. Meaning the Docker image that we'd like to create is run inside another docker container. The before script on lines 11-12 gives access to the image registry for the CI runner using access tokens. Authentication and authorization to the runner is handled seamlessly by GitLab.

The image is then built using the Dockerfile stored in our repository.

A simple test is run on line 33, to see that php is working, our application file exists in the right directory and has the correct php syntax.

The image is then deployed using a GitLab runner with the tag ”deployer”. This is a shell runner with docker installed.



```
#!/bin/bash

if [ "$(docker ps -q -f name=espcontrol)" ]; then
         docker stop espcontrol
         docker rm espcontrol
fi

# Start Container
docker pull testlab.dittmer.fi:4567/devicedev/espcontrol:latest

docker run --detach \
     --hostname espcontrol.dittmer.fi \
     --ip 10.0.10.42 \
     --network my-macvlan-net-v2 \
     --name espcontrol \
     --restart always \
     testlab.dittmer.fi:4567/devicedev/espcontrol:latest
```

Figure 12. docker-deploy.sh

This is a quick shell script that starts the docker container on a docker host. It checks if the container exist, and if so destroys it. Then it will fetch the latest version of the container and runs it on the server.

7 Final Conclusion

Building an automated build system was successful, although many rough edges were left for future development. The system is working well for its intended purpose and is being used to build new projects.

A good development environment should be flexible and ready for change. Relying on any single technology or being locked to a vendor will eventually lead into problems.

The way applications are run on any system has always changed in the short history of computing, and will continue to change. Unikernel applications might change the way services are programmed and programming languages are constantly changing as well. Artificial Intelligence might change the way we look at programming on another abstraction layer. Whatever the case may be, managing these applications will continue to be a challenge for the foreseeable future.

Many improvements can be made to this system. Starting from an centralized authentication and authorization service, not only managing people but also computer systems. Storage systems could be hosted in a SAN network, improving storage reliability, extensibility and management. Openstack could be used to deploy such a storage system, and much more.

These solutions however would require larger financial investment to do properly. But there are lot of improvements that could and should be implemented. The first of which is kubernetes.

Deploying to a kubernetes cluster would enable the usage of its API which is used by many deployment systems such as GitLab. It should be possible to run a small self hosted cluster or use many of the cloud providers which support some form of Kubernetes deployment, such as Amazon EKS or Google GKE. Some kind of hybrid system could also be made, deploying depending on the needs of the application, privacy requirements, scaling and cost.

The planned goals were met, and the system will continue to be used and evolve depending on future needs.

References

[1] Esp32-WROOM-32 Datasheet [cited 2019 April 10] <https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf>

[2] Setup Linux Toolchain From Scratch [Cited 2019 April 10] <https://docs.espressif.-com/projects/esp-idf/en/latest/get-started/linux-setup-scratch.html>

[3] ESP-IDF FreeRTOS SMP Changes [Cited 2019 April 11 <[https://docs.espressif.-com/projects/esp-idf/en/latest/api-guides/freertos-smp.html>

[4] GNU Make Project site [Cited 2019 May 10] <https://www.gnu.org/software/make/>

[5] MQTT Documentation [Cited 2019 May 15] <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>

[6] Mosquitto Project Page [Cited 2019 May 15] <https://mosquitto.org/>

[7] Debootstrap manual [Cited 2019 February 10] <https://manpages.debian.org/stretch/debootstrap/debootstrap.8.en.html>

[8] Chroot manual [Cited 2019 Febuary 10] <http://man7.org/linux/man-pages/man2/ch-root.2.html>

[9] Introduction to KVM Virtualization [Cited 2019 May 16] <https://doc.opensuse.org/documentation/leap/virtualization/html/book.virt/cha.kvm.intro.html>

[10] Kernel Virtual Machine [Cited 2019 May 16] <https://www.linux-kvm.org/page/Main_Page>

[11] Qemu wiki [Cited 2019 May 16] <https://wiki.qemu.org/Main_Page>

[12] About images, containers, and storage drivers [Cited 2019 May 18] <https://docs.-docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/>

[13] Docker Hub – Container Image Library [Cited 2019 May 18] <https://www.docker.-com/products/docker-hub>

[14] Yet Another Crypto Mining Botnet? [Cited 2019 May 18] <https://www.fortinet.-com/blog/threat-research/yet-another-crypto-mining-botnet.html>

[15] GitLab Container Registry administration [Cited 2019 May 19] <https://docs.gitlab.-com/ee/administration/container_registry.html>

[16] What is DevOps? [Cited 2019 April 20] <http://radar.oreilly.com/2012/06/what-is-devops.html>

[17] Revisiting “What is DevOps” [Cited 2019 April 20] <http://radar.oreilly.com/2014/06/revisiting-what-is-devops.html>

[18] CISSP Official Study Guide, Seventh edition, ISBN 978-1-119-04271-6

[19] The DevOps Lifecycle with GitLab [Cited 2019 May 5] <https://about.gitlab.com/stages-devops-lifecycle/>

[20] What is CI/CD [Cited 2019 April 20] <https://www.redhat.com/en/topics/devops/what-is-ci-cd>

[21] GitLab Runner [2019 May 10] <https://docs.gitlab.com/runner/>

[22] Linux Kernel Mailing List [Cited 2019 April 8] <http://lkml.iu.edu/hypermail/linux/kernel/0504.0/1540.html>

[23] About Git [Cited 2019 April 8] <https://git-scm.com/about>

[24] Apt-Cache NG [Cited 2019 April 8] <https://www.unix-ag.uni-kl.de/~bloch/acng/>

Create-disk-image.sh

```
#/bin/bash
# Create the mountpoint where the disk image will be mounted
mkdir mntpnt

# Writing a bunch of zeroes (Null bytes) into a file, which is our disk image.
dd if=/dev/zero of=devuan-686.img bs=1G count=3

# Parted requires the disk to be labelled before partitioned.
parted -s devuan-686.img mklabel msdos

# Partitioning the drisk image with a single partition and setting the boot flag as
# we want to boot from the device.
parted -s -a opt devuan-686.img mkpart primary 0% 100% set 1 boot

# We want access to the disk image as a block device. This can be achieved using a
# loop device. losetup is found in the util-linux package
losetup -P -f ./devuan-686.img /dev/loop0

# Formatting the disk image with an ext4 filesystem.
# The disk image is attached to the loop device loop0
# We can access the first (and single) partition as a block device at /dev/loop0p1
mkfs.ext4 /dev/loop0p1

# Mounting the disk image at our mountpoint directory. This will give us access to the filesystem.
mount /dev/loop0p1 mntpnt/

# Install the basic OS filesystem. We will also install some additional packages such as the kernel,
# grub bootloader and networking tools.
debootstrap --arch=i386 --variant=minbase \
--include=acpid,netbase,net-tools,ifupdown,isc-dhcp-client,linux-image-686-pae,grub-pc ascii mntpnt/ \
http://pkgmaster.devuan.org/merged

# mount proc and sys pseudo-file systems. These are directories from which there is access to
# proccess information or kernel data structures. Mounting the same sys and dev as the host to
# the disk image fs during installation. dev includes access to devices such as disks.
/bin/mount -t proc proc mntpnt/proc
/bin/mount -o bind /dev mntpnt/dev
/bin/mount -o bind /sys mntpnt/sys

# Running these commands in the chroot environment, from bin/bash until the EOF line
cat << EOF | chroot mntpnt/ /bin/bash

# Setting hostname
echo myimage > /etc/hostname
echo 'Acquire::http { Proxy "http://apt.example.com:3142"; };' >> /etc/apt/apt.conf.d/01proxy
echo deb http://pkgmaster.devuan.org/merged ascii-backports main >> /etc/apt/sources.list

# Install the Grub boot loader.
grub-install /dev/loop0
update-grub2

# Configure network configuration for eth0 nic. Will use dhcp here.
printf "auto eth0\niface eth0 inet dhcp" >> /etc/network/interfaces

# Set a default password
echo "root:secret_password_here" | chpasswd

# Exit the chroot environment
EOF
```

```
# Cleanup. Unmount devices. Disconnect loop device and remove mountpoint directory.
/bin/umount mntpnt/proc
/bin/umount mntpnt/dev
/bin/umount mntpnt/sys
/bin/umount mntpnt/
losetup -d /dev/loop0
rmdir mntpnt

# Image should now be usable, file devuan-686.img
```

main.c

```
/* HTTP Server */
/* Based on the simple http_server example, included in the espressif IDF */
/* which is in the Public Domain (or optionally CC0 licenced), */

#include <esp_wifi.h>
#include <esp_event_loop.h>
#include <esp_log.h>
#include <esp_system.h>
#include <nvs_flash.h>
#include <sys/param.h>
#include <string.h>
#include "driver/gpio.h"
#include <esp_http_server.h>

#define BLINK_GPIO 2
#define EXAMPLE_WIFI_SSID "WiFISSID"
#define EXAMPLE_WIFI_PASS "WiFiPassword"

static const char *TAG="APP";
char led_status[10] = "OFF";

void set_led_state();

// Blink the LED in a seperate task
void blink_task(void *pvParameter) {
 while (strcmp("BLINK", led_status) == 0) {
        gpio_set_level(BLINK_GPIO, 0);
        vTaskDelay(500 / portTICK_PERIOD_MS);
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
 set_led_state();
 vTaskDelete( NULL );
}

// Set LED in correct state
void set_led_state() {
 if(strcmp("OFF", led_status) == 0) { gpio_set_level(BLINK_GPIO, 0); }
 if(strcmp("ON", led_status) == 0) { gpio_set_level(BLINK_GPIO, 1); }
  if(strcmp("BLINK",  led_status)  ==  0)  {xTaskCreate(&blink_task,  "blink_task",  configMINIMAL_STACK_SIZE,  NULL,  5,
NULL); }
}


/* An HTTP GET handler */
esp_err_t hello_get_handler(httpd_req_t *req)
{
    char*  buf;
    size_t buf_len;

    /* Get header value string length and allocate memory for length + 1,
     * extra byte for null termination */
    buf_len = httpd_req_get_hdr_value_len(req, "Host") + 1;
    if (buf_len > 1) {
        buf = malloc(buf_len);
        /* Copy null terminated value string into buffer */
        if (httpd_req_get_hdr_value_str(req, "Host", buf, buf_len) == ESP_OK) {
            ESP_LOGI(TAG, "Found header => Host: %s", buf);
        }
        free(buf);
    }
    /* Send response with custom headers and body set as the
     * string passed in user context*/
    const char* resp_str = (const char*) req->user_ctx;
    httpd_resp_send(req, resp_str, strlen(resp_str));

    /* After sending the HTTP response the old HTTP request
     * headers are lost. Check if HTTP request headers can be read now. */
    if (httpd_req_get_hdr_value_len(req, "Host") == 0) {
        ESP_LOGI(TAG, "Request headers lost");
    }
    return ESP_OK;
}

httpd_uri_t hello = {
    .uri       = "/",
    .method    = HTTP_GET,
    .handler   = hello_get_handler,
    .user_ctx  = "<html><body><h1>Hello there!</h1>This is the esp32</body></html>"
};

char *parse_post(char message_str[100]){
  // Parse status request
  if (strncmp("STATUS", message_str, 6) == 0){
    ESP_LOGI(TAG, "==================STATUS==============");
    if (strcmp("LEDS", message_str+7) == 0){
      ESP_LOGI(TAG, "==================LEDS==============");
      if (strcmp("OFF", led_status) == 0) {
        return "OFF";
      }
      if (strcmp("ON", led_status) == 0) {
        return "ON";
      }
      if (strcmp("BLINK", led_status) == 0) {
        return "BLINK";
      }

    }
    if (strcmp("PING", message_str+7) == 0){
        return "PONG";
    }
  }

 // Parse SETLED request
 if (strncmp("SETLED", message_str, 6) == 0){
    ESP_LOGI(TAG, "==================Setting LED==============");

    if (strcmp("ON", message_str+7) == 0){
      ESP_LOGI(TAG, "==================LED ON==============");
      strcpy(led_status, "ON");
      return led_status;
     }
    if (strcmp("OFF", message_str+7) == 0){
      ESP_LOGI(TAG, "==================LED OFF==============");
      strcpy(led_status, "OFF");
      return led_status;
     }
    if (strcmp("BLINK", message_str+7) == 0){
      ESP_LOGI(TAG, "==================LED BLINK==============");
      strcpy(led_status, "BLINK");
      return led_status;
    }
 }
 return "Can't Parse Message";
}
```

```
/* An HTTP POST handler */
esp_err_t led_post_handler(httpd_req_t *req)
{
    char buf[100];

    int ret, remaining = req->content_len;

    while (remaining > 0) {
        /* Read the data for the request */
        if ((ret = httpd_req_recv(req, buf,
                        MIN(remaining, sizeof(buf)))) <= 0) {
            if (ret == HTTPD_SOCK_ERR_TIMEOUT) {
                /* Retry receiving if timeout occurred */
                continue;
            }
            return ESP_FAIL;
        }

        /* Log data received */
        ESP_LOGI(TAG, "=========== RECEIVED DATA ==========");
        ESP_LOGI(TAG, "%.*s", ret, buf);
        ESP_LOGI(TAG, "====================================");


 char *return_message = parse_post(buf);
 set_led_state(led_status);
        ESP_LOGI(TAG, "Sending now: %s : %i", return_message, strlen(return_message));

        httpd_resp_send_chunk(req, return_message, strlen(return_message));
        remaining -= ret;
}

    // End response
    httpd_resp_send_chunk(req, NULL, 0);
    return ESP_OK;
}

httpd_uri_t led = {
    .uri       = "/led",
    .method    = HTTP_POST,
    .handler   = led_post_handler,
    .user_ctx  = NULL
};


httpd_handle_t start_webserver(void)
{
    httpd_handle_t server = NULL;
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();

    // Start the httpd server
    ESP_LOGI(TAG, "Starting server on port: '%d'", config.server_port);
    if (httpd_start(&server, &config) == ESP_OK) {
        // Set URI handlers
        ESP_LOGI(TAG, "Registering URI handlers");
        httpd_register_uri_handler(server, &hello);
        httpd_register_uri_handler(server, &led);
        return server;
    }

    ESP_LOGI(TAG, "Error starting server!");
    return NULL;
}
```

```
void stop_webserver(httpd_handle_t server)
{
    // Stop the httpd server
    httpd_stop(server);
}

static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    httpd_handle_t *server = (httpd_handle_t *) ctx;

    switch(event->event_id) {
    case SYSTEM_EVENT_STA_START:
        ESP_LOGI(TAG, "SYSTEM_EVENT_STA_START");
        ESP_ERROR_CHECK(esp_wifi_connect());
        break;
    case SYSTEM_EVENT_STA_GOT_IP:
        ESP_LOGI(TAG, "SYSTEM_EVENT_STA_GOT_IP");
        ESP_LOGI(TAG, "Got IP: '%s'",
                ip4addr_ntoa(&event->event_info.got_ip.ip_info.ip));

        /* Start the web server */
        if (*server == NULL) {
            *server = start_webserver();
        }
        break;
    case SYSTEM_EVENT_STA_DISCONNECTED:
        ESP_LOGI(TAG, "SYSTEM_EVENT_STA_DISCONNECTED");
        ESP_ERROR_CHECK(esp_wifi_connect());

        /* Stop the web server */
        if (*server) {
            stop_webserver(*server);
            *server = NULL;
        }
        break;
    default:
        break;
    }
    return ESP_OK;
}

static void initialise_wifi(void *arg)
{
    tcpip_adapter_init();
    ESP_ERROR_CHECK(esp_event_loop_init(event_handler, arg));
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = EXAMPLE_WIFI_SSID,
            .password = EXAMPLE_WIFI_PASS,
        },
    };
    ESP_LOGI(TAG, "Setting WiFi configuration SSID %s...", wifi_config.sta.ssid);
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config));
    ESP_ERROR_CHECK(esp_wifi_start());
}
```

```
void app_main()
{
    gpio_pad_select_gpio(BLINK_GPIO);
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
    gpio_set_level(BLINK_GPIO, 0);

    static httpd_handle_t server = NULL;
    ESP_ERROR_CHECK(nvs_flash_init());
    initialise_wifi(&server);
}
```