18. January 2018

cion

Gruppe 4

Harald Floor Wilhelmsen & Erlend Tobiassen

Table of contents

Preface

This assignment was solved by two students at the ITHINGDA study program at NTNU in Trondheim, Norway. The study program is an engineer's Bachelor degree in IT, focusing on practical experience in development and working together in small teams on software development projects.

The goal of the project is to learn about cloud services on a deeper level. In this case the software Docker Swarm. To learn how to achieve asynchronous code in python, to learn how to implement secure REST-endpoints and to learn how to reliably use SSL certificates to secure communication between two services. And use that knowledge to create a continuous deployment tool that updates running services in docker swarm.

During the development the executers have had to figure out docker at a deep level, not only because cion is a large multi-service stack running in docker swarm itself, but also because we had to interface with docker's low level api in order to manage running services.

The project executers would like to thank Trondheim municipality for offering the assignment, providing workstations and an environment where they have direct access to the intended end-users. They would also like to thank Elena Falkenberg Nordmark for the design of the cion logo.

Assignment description

The project is a development-assignment where the developers are to create a product that eases operations by automating the process of delivering updates to running services. This in turn will make the team more agile[1].

The assignment is only one part in an preexisting continuous deployment system, namely it deals with the deployment to docker swarm environments when a external system tells it there is a new build.

The contractor is the IT-services at Trondheim municipality. IT-services at Trondheim municipality is an organization that provides software solutions and IT support to all employees of Trondheim municipality. Since the organization has over 10.000 employees this is a large collection of tasks.

How was the assignment made/solved/executed?

Methods and Standards

The developers started the development process using the scrum[2], [3] framework, but they realised during the process that using scrum in its full form was going to be impractical. This was due to multiple reasons, the most important one was that the developers' available time was unevenly distributed across the semester, so following the planned 3-week sprint was not going to work.

So what they did was use some components of scrum, like the task board with swimlanes, specifically they used JIRA[4] to track tasks and issues and distribute them across the team.

Our python code is formatted according to the PEP8[2] style guide.

Use of literature and internet resources

The team has only used internet resources to trouble-check and search for errors in their software, and they have used a lot of documentation for individual components that are prominent in the software. This includes:

- Docker[5]
- The python asyncio library[6]
- The Docker python API[7]
- Docker Engine SDK[8]

Hardware used in the process

When it comes to hosting and testing of the solution, mainly the Quality Assurance environment from Trondheim municipality was used, this due to it being as similar to a production environment as possible, but if something broke it would not be a problem.

The team's computers that were used for development are lent from the municipality. They also supplied a place to work in their offices on the 4th floor of *Trondheim Torg*.

Software used in the process

When it comes to development tools our IDE of choice is PyCharm, and we have used multiple different ssh-clients to connect to our hosts. The hosts running Docker on Trondheim

municipality's servers are running Ubuntu Server and we have used Oracle VM VirtualBox with CentOS to do manual testing of our solution.

The municipality provided access to JIRA for task/issue tracking, confluence for writing documentation and BitBucket for source control.

How tasks were divided across the team

JIRA was used to manage tasks. And the team claimed tasks as they worked. In general, the work was divided into two categories, frontend/web and raw backend. Frontend/web includes the components catalyst and web frontend/backend. Backend includes the orchestrator and workers. The work required on the database component was evenly divided as the work was needed. So as it stands at the time of writing one team member has a good understanding of the web parts and another has a good understanding of the backend.

Created Documentation

Together with this document you should have also received these documents:

- User guide
- Vision document
- Project plan
- Hours list spreadsheet
- Meeting requests and reports.

Project execution

What went well

The development process and use of JIRA went well. The developers on the team are dedicated programmers, and enjoy writing code. So when given a large task like this they enjoyed doing the programming work, and because of that they ended up with a well structured code base. Leveraging Docker to host the solution also worked out well, due to how Docker is is used. cion is designed to run as docker containers, which made the team design the solution to leverage that structure. This is why the cion is designed as *components*.

The architecture design outlined in the vision document was very well thought out and ended up in the end product with next to no changes.

What could have gone better

The developers project plan defined 5 epics, with each having gotten a percentage of the 500 hours at disposition. As is evident in our time management spreadsheet we underestimated the time allotted for Documentation and overestimated the time needed for Security. The developers prioritised making a good end product over working on the planned user documentation. The reason for this was mostly that the original plan was not to continue working on it in a bachelor's thesis, and that they wanted to create a product that they were proud of writing in the alloted time for the project.

They wanted something to link their name to, mention on CVs and job applications, and to have a real response to the classic question that people ask when they hear you are a developer: "*Have you made anything cool?*".

When the developers decided they were continuing the project into their bachelor's they realised that documentation would quickly become deprecated as they continued development, which means that they postponed it until features started being complete. This had the unfortunate outcome of current documentation being poor and incomplete, due to planned features not being implemented yet, due to them being planned for the bachelor's. This was decided due to some of the current features being planned for deprecation as the bachelor's thesis starts going.

What could have been done differently

The team should have started working on this document earlier, and worked on it continuously through the development process. This would have resulted in a more well documented end product. But this could also have had the unfortunate outcome of the product likely ending up in an unfinished state.

The service deployment components got too complicated with little gain for Trondheim municipality's needs. Although creating a service that scales is an exciting task, it became more work than anticipated. The service should have been a single instance that combines the orchestrator and the worker.

Project requirements and execution

A requirements document does not exist. But the team agreed with the contractor at the start of the work what the product was to do. The main task of cion is to update services running in docker swarm, with a configurable level of automation, but there were some other requirements for the solution:

- The solution **is** to run in a docker swarm

- The solution **is** to update services running in a docker swarm where cion itself does not run
- The solution **is** to trigger when pushing docker images to dockerhub
- The solution **may** have a web UI where the user can configure the solution and view logs/status
- There **may** also be a configuration option, when adding a new service to cion to manage, that cion is to only deploy to a specific swarm with interaction from a human/admin, for the municipality this swarm is their production environment. This feature was known as “signing deployments”

Those were the originally discussed goals. All the requirements were filled, but a lot of features not discussed at the start of the project were implemented, and some were changed. These include:

- A user can trigger updates of services manually through the web UI
- The web UI is protected with username and password through a user system
- In the web UI, configuration of the solution is done through menus and forms, not just a raw JSON text editor(which was the original plan)
- The “signing deployments” feature was replaced by the manual deployments feature, due to manual deployment being approximately 5 clicks away from the main web UI page

Further work

The project is being continued in a bachelor’s assignment.

At the time of writing, the plan is to expand support to beyond Docker Swarm and to let the end-user not only to update services, but to also set up completely new services through the web interface. There also a planned feature to add support for triggering functions on completion of specific tasks. These functions can be webhooks, running of python-code or similar.

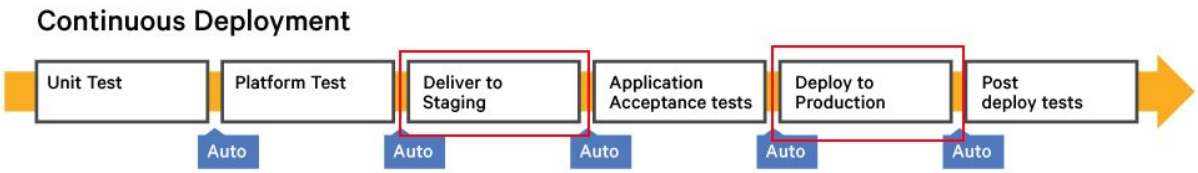
At the time of writing, the solution is running on Trondheim municipality’s servers. Since this is mainly a tool for deploying to environments, it would be correct to say that cion is in production. Though due to updates in Trondheim municipality’s production servers, we were not able to install the necessary certificate files for cion to manage that docker swarm as well.

But this is planned by the first month of the bachelor’s. This is also when we are going to set up cion so that it can update itself on Trondheim municipality’s servers. So that when the developers pushes a new update to one of cion’s components, the solution will update itself on Trondheim municipality’s servers

What is cion

What it does

cion is designed to be a part of a Continuous Deployment[9] environment. Specifically the part that deploys updates to services.



What it does **not** do

cion is at the time of writing not designed to run any tests or similar when deployment is finished.

These kinds of tests include:

- Unit tests
- Platform tests
- Application acceptance tests
- Post deploy tests

What it **does** do

- Any kind of service update deployment to any environment:
    - Test
    - Staging (QA)
    - Production
    - And so on…

Why?

The project was started because of a need in Trondheim Municipality's development team. The need was automatic deployment to the Docker swarm environments that the development-team has. At the time of writing these environments are *production*, *Quality Assurance* (QA) and *test*.

The team already has automatic testing and building of Docker images environment through Bitbucket Pipelines. And countless other unit test and build-services already exist, this is why cion is focused on the actual deployment phase of the Continuous Deployment pipeline.

How it works

cion is designed to run in a Docker swarm, and utilizes the container structure that is provided by swarm.

It consists of multiple components:

- Database
- Catalyst
- Web frontend
- Web backend
- Service deployment orchestrator
- Service deployment worker

Database

The database that is used in cion is a json document storage maintained by the Linux Foundation named RethinkDB[10].

The database provides 3 different storage areas, for the users, for configuration, and for events. The events can be external events like “new build for image x” or internal events like “deploy image x to environment y”. This enables all of the loosely coupled components to agree on a common state.

Catalyst

This component receives notifications when a new docker image has been built. The notifications are webhooks that can be configured in your Dockerhub repository[11]. The catalyst parses the webhook request body and creates a “new build for image x” event in the database.

Web interface

This component serves a webpage and leverages the web backend letting the user:

- manually create deployment tasks, selecting any version of any recorded docker image to update any running service
- add docker environments that cion can deploy to

- configure new services for cion to update
- view logs of past deployments
- view status of deployments that are in progress

Service deployment orchestrator and worker

The service deployment is split into two distinct parts, this is done in order to take advantage of docker's ability to scale applications. Because one can create as many workers as one wishes the service deployment should scale horizontally[12]

The orchestrator listens to changes in the events in the database and when a new unprocessed event is added it distributes a task to a single worker, when the worker is done the orchestrator changes the status of the event to "processed" assuming the task was completed without error.

The worker is some python code that implements a task interface, as of now the main method that are implemented by workers is: given a image and a target service, update that service with the image.

Concerns and decisions

Choice of technologies

Language

Python is used in cion for several reasons:

- Docker provides an API to python
- It integrates well with the chosen database
- cion's developers have previous experience with the language, which:
    - makes the development process faster
    - makes the solution inherently more stable and secure, due to the developers knowing about holes and concerns that have to be addressed
- it provides an easy way to leverage other people's work
- python's interpreter starts up fast compared to e.g. Java[13], which means that cion's docker images start quickly, and the solution feels responsive.

Database

Cion uses a RethinkDB database to store its state. This is because it offers an easy integration with python. It is not a relational database, which fits cion's usage of it, reason being there are very few instances where a relational database would be positive instead of negative.

One of the few situations where a relational database would be a good thing is binding users to manual deployments, but this is a small part of the solution and the shortcomings of RethinkDB are easily worked around here.

The cion web component

The web interface consists of two docker containers. One for the backend API, and one for serving the web-page in itself and a proxy to talk to the API.

The frontend container, the one that acts as a proxy and serves the web page, uses javascript and the library *mithril[14], [15]* to generate the web-page and Caddy[14]–[16] for the proxy. Mithril allows us to code the webpage in pure javascript. Caddy allows us to protect the web endpoints in HTTPS without requiring the user to wrap cion in an HTTPS-proxy on their own. To set up HTTPS in Caddy all the user has to do is supply the SSL certificate files.

The backend API container, the container that serves the web-endpoints that fetches data from the database, uses python with *aiohttp[14]–[17]* to serve the web endpoints and the rethinkdb python driver[10], [18] to fetch data from the database.

The cion catalyst component

The catalyst is built with python using a library called Flask[14]. Flask is a small web-framework for python. The catalyst uses the obscurity[19][20] principle to protect its endpoint, by hiding its URL with a token configured by the user. This security measure was taken because Dockerhub simply does not support sending any kind of authorization data with its requests.

Platform and use of Docker

The cion solution is a Continuous Deployment environment to deploy updates to services run in Docker, which means that presuming that the user is using a Docker environment is a safe bet. This is why cion is designed to be run in a Docker environment.

Security

cion is a sensitive solution, because it allows you to deploy services to running docker instances. This is why security has been one of our main concerns during development. Securing the communication between the components running on different environments has to be encrypted or obfuscated, to secure the solution from man-in-the-middle (MITM) attacks, and to prevent leaks of sensitive data such as login credentials and keys.

Webhook

The solution has a webhook which receives requests from a docker registry. This is a vulnerable component, seeing as if you can trick the webhook-receiver to accept your requests you can deploy updates to services in production environments.

cion solves this issue by using the security by obscurity principle[20]. It does this by keeping the URL for where it can receive webhooks a secret.

Docker

Swarm-mode

cion is intended to run in a docker swarm. The advantage to this is the ability to easily run as multiple containers, and have multiple services, and it leverages container replication to optimize performance. Since cion is a management tool for docker swarms, it was also seen as a given that the end user would have one available for cion to run in.

Techniques for managing docker swarms

Managing a docker instance can be done in two ways[21]: through the non-networked docker unix socket, or through an HTTP socket. To communicate with the unix socket the process has to have direct access to the unix socket file. To manage a docker swarm you have to communicate with the socket of a docker swarm leader.

To communicate with the HTTP socket it has to enabled in the docker daemon configuration. Enabling this feature will open up your docker instance to attacks, so securing it with TLS is all but mandatory. Using a TLS-protected HTTP socket is the recommended way of letting cion manage the services of a swarm, but cion supports both ways of communication. This is because, if cion had to use the unix socket, you would have to give the cion worker access to the unix socket file for docker, and it would have to run on the same host and in the same docker instance as the services cion is to manage, which means that cion would only be able to manage a single swarm, the one it is running inside.

Terms and explanations

Actors

End-user

Since cion is a development/deployment tool, the end-users are intended to be developers or operations managers.

Owners

These are the owners of the hosts and services where cion is to manage service updates. These hosts may or may not be where cion in itself runs.

Developers

These are the developers, designers and managers of cion as a software product. They are in no way inherently the managers, owners or developers of the systems/hosts/docker swarms where cion is deployed and running.

cion

Deployment task/task

This refers to a task where cion is to update a service. Tasks can be created by either the webhook-receiver *catalyst* or by manually them through the web-interface.

Environment

An environment is where your software is running. The term is used in this document to refer to the distinction between having your software be accessible only by developers from the same company, developers from different companies and everyone else. The IT-services of Trondheim municipality has three environments. Production, Quality Assurance (QA) and test.

Test

This environment developers can deploy without much regard. It is where they test that their software runs. No units outside the development team has access to the services running here and no guarantee is given for having services running here.

Quality Assurance (QA)

Outside units can be granted access to this environment to test their own code. For example if they are developing an integration to the production environment.

Production

This is the environment where the production code is running. As in the services and application that are in use by the end-users of the company.

Docker

Docker [22] is a container platform, allowing developers to isolate and manage running services.

Docker image

A docker image is not an instance of the application, but merely describes how docker can instantiate the application. So it contains the application's runnable code, and a script to start it. Images are hosted and stored in what is known as a *registry*.

An image has a name, which consists of the name of the repository and a tag, separated by a colon (app-name:tag). A repository is intended for images for a single application, so the repository name should be the name of the application. The tag is usually the version of that specific application.

So if you have an application called *web-page*, your image names would be something like *web-page:latest*, *web-page:1.0.0* and *web-page:2.0.0*.

Docker registry/repository

A docker registry stores and serves docker images. It is also referred to as a *repository*.

Dockerhub

Dockerhub [23] is a cloud-based docker image hosting service which developers use to serve images of their application. A user can create an account and create unlimited free public

repositories, and pay a fee to have private repositories. So in other words, it is the github of docker images.

Docker container

A docker container is the a running instance of an application and the sandbox the application runs in. It can be seen as a virtual machine, with its own file-system and users, where the only running process is the application itself.

Docker swarm

Swarm [24] is a docker-feature that binds multiple physical or virtual machines together into the same docker environment.

Webhook

A webhook is an HTTP POST-request that is triggered on some event to a receiver.

In the context of this document, a webhook is most often an HTTP POST request that is triggered when a new docker image has been uploaded to a docker image repository/registry. The webhook receiver in this case is the running instance of cion.

Bitbucket

Bitbucket is a git-server with additional features by Atlassian. It can be self-hosted or accessed as a cloud service hosted by Atlassian.

The features beyond being a git-server include:

- integration with the rest of Atlassian's software, like:
  - JIRA
  - Confluence
- Bitbucket pipelines, a CI/CD environment
- A web UI

Bitbucket pipelines

Bitbucket pipelines is a CI/CD environment integrated with the Bitbucket git-server, which lets the user create scripts that run on specific triggers, like a git push to a specific branch.

The feature is very customisable. For example in Trondheim municipality's development team, the flow of a typical pipeline looks like this:

- Trigger on push to master branch
    - Run tests using maven
    - Build the application to a *war*-file with maven
    - Build a docker image from the *war*-file created by maven
    - Push the docker image to our dockerhub repository, tagging the image with the version from the last git commit to the master branch
- Trigger on push to the develop branch
    - Run tests using maven
    - Build the application to a *war*-file with maven
    - Build a docker image from the *war*-file created by maven
    - Push the docker image to our dockerhub repository, tagging the image with *latest*

Citations

[1] “What is Agile Software Development?,” *Agile Alliance*, 29-Jun-2015. [Online]. Available: https://www.agilealliance.org/agile101/. [Accessed: 11-Jan-2018]

[2] “PEP 8 -- Style Guide for Python Code,” *Python.org*. [Online]. Available: https://www.python.org/dev/peps/pep-0008/. [Accessed: 11-Jan-2018]

[3] “What is Scrum?,” *Scrum.org*. [Online]. Available: http://www.scrum.org/resources/what-is-scrum. [Accessed: 11-Jan-2018]

[4] Atlassian, “Jira | Issue & Project Tracking Software | Atlassian,” *Atlassian*. [Online]. Available: https://www.atlassian.com/software/jira. [Accessed: 11-Jan-2018]

[5] “Docker Documentation,” *Docker Documentation*, 09-Jan-2018. [Online]. Available: https://docs.docker.com/. [Accessed: 11-Jan-2018]

[6] “18.5. asyncio — Asynchronous I/O, event loop, coroutines and tasks — Python 3.6.4 documentation.” [Online]. Available: https://docs.python.org/3/library/asyncio.html. [Accessed: 11-Jan-2018]

[7] “Docker SDK for Python — Docker SDK for Python 2.0 documentation.” [Online]. Available: https://docker-py.readthedocs.io/en/stable/. [Accessed: 18-Jan-2018]

[8] “Develop with Docker Engine SDKs and API,” *Docker Documentation*, 09-Jan-2018. [Online]. Available: https://docs.docker.com/develop/sdk/. [Accessed: 11-Jan-2018]

[9] C. Caum, “Continuous Delivery Vs. Continuous Deployment: What’s the Diff?,” *Puppet*. [Online]. Available: https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff. [Accessed: 10-Jan-2018]

[10] “RethinkDB: the open-source database for the realtime web.” [Online]. Available: https://www.rethinkdb.com/. [Accessed: 17-Jan-2018]

[11] “Webhooks for automated builds,” *Docker Documentation*, 17-Jan-2018. [Online]. Available: https://docs.docker.com/docker-hub/webhooks/. [Accessed: 18-Jan-2018]

[12] D. Beaumont, “How to explain vertical and horizontal scaling in the cloud - Cloud computing news,” *Cloud computing news*, 09-Apr-2014. [Online]. Available: https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/. [Accessed: 18-Jan-2018]

[13] “AWS Lambda – compare coldstart time with different languages, memory and code sizes,” *theburningmonk.com*, 13-Jun-2017. [Online]. Available: http://theburningmonk.com/2017/06/aws-lambda-compare-coldstart-time-with-different-languages-memory-and-code-sizes/. [Accessed: 18-Jan-2018]

[14] “Welcome | Flask (A Python Microframework).” [Online]. Available: http://flask.pocoo.org/. [Accessed: 17-Jan-2018]

[15] “Introduction - Mithril.js.” [Online]. Available: https://mithril.js.org/. [Accessed: 17-Jan-2018]

[16] “Caddy - The HTTP/2 Web Server with Automatic HTTPS.” [Online]. Available: https://caddyserver.com/. [Accessed: 17-Jan-2018]

[17] “aiohttp: Asynchronous HTTP Client/Server for Python and asyncio — aiohttp 2.3.9-documentation.” [Online]. Available: https://aiohttp.readthedocs.io/en/stable/. [Accessed: 17-Jan-2018]

[18] “ReQL command reference - RethinkDB.” [Online]. Available: https://www.rethinkdb.com/api/python/. [Accessed: 17-Jan-2018]

[19] “Learn Cryptography - Security By Obscurity.” [Online]. Available:

https://learncryptography.com/cryptanalysis/security-by-obscurity. [Accessed: 18-Jan-2018]
[20] “Obscurity is a Valid Security Layer,” *Daniel Miessler*. [Online]. Available: https://danielmiessler.com/study/security-by-obscurity/. [Accessed: 10-Jan-2018]
[21] “Protect the Docker daemon socket,” *Docker Documentation*, 17-Jan-2018. [Online]. Available: https://docs.docker.com/engine/security/https/. [Accessed: 18-Jan-2018]
[22] “What is Docker,” *Docker*, 14-May-2015. [Online]. Available: https://www.docker.com/what-docker. [Accessed: 21-Sep-2017]
[23] “dockerhub,” *dockerhub*. [Online]. Available: https://hub.docker.com/. [Accessed: 21-Sep-2017]
[24] “Swarm mode overview,” *Docker Documentation*, 21-Sep-2017. [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed: 21-Sep-2017]