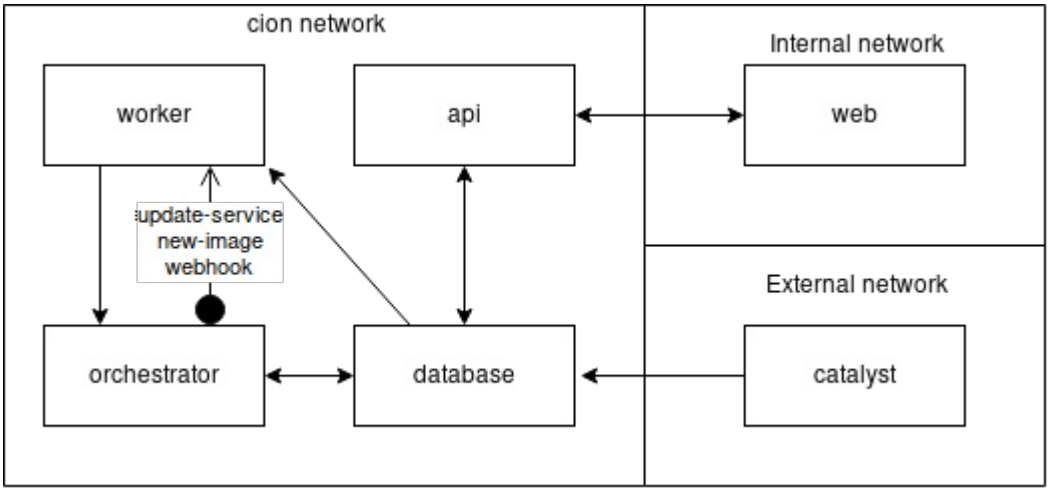
Cion

The cion solution is composed of 5 main microservices communicating with a database. Each service runs in one of the 3 following networks.

Network	Specification
External	Must be accessible by the image hosts(dockerhub/docker registry)
Internal	Must be accessible by the users of the cion web interface
cion	Internally used network.



The black arrows shows direction of data flow.

web

read the docs (http://docs.cionkubes.com/projects/web)
dockerhub (https://hub.docker.com/r/cion/web)
github (https://github.com/cionkubes/cion-web)

api

read the docs (http://docs.cionkubes.com/projects/api)
dockerhub (https://hub.docker.com/r/cion/api)
github (https://github.com/cionkubes/cion-api)

worker

read the docs (http://docs.cionkubes.com/projects/worker)
dockerhub (https://hub.docker.com/r/cion/worker)
github (https://github.com/cionkubes/cion-worker)

orchestrator

read the docs (http://docs.cionkubes.com/projects/orchestrator)
dockerhub (https://hub.docker.com/r/cion/orchestrator)
github (https://github.com/cionkubes/cion-orchestrator)

catalyst

read the docs (http://docs.cionkubes.com/projects/catalyst)
dockerhub (https://hub.docker.com/r/cion/catalyst)
github (https://github.com/cionkubes/cion-catalyst)# Configuring cion

The cion web interface primarily exists to aid in configuring the automatic deployment of images to services in either docker swarm or kubernetes. Some experimental features may be configurable in a online text editor while a UI is developed for the feature.

External webhooks

In order for cion to know about new images uploaded to docker image hosts, external webhooks must be configured in the image hosts.

Dockerhub

Firsly you will need admin-access to the repository you are adding the webhook to. If you are the owner you will already have this.

Go to the repository page and click `Webhooks`. This will bring you to a new page. Then click the `+` icon next to `WEB HOOKS`.

Enter a descriptive name for your webhook, e.g. 'cion', and the url for the cion-catalyst service. This is probably something like `yourdoman.org/cion-catalyst/dockerhub/<URL_TOKEN>` or `cion.yourdomain.org/dockerhub/<URL_TOKEN>`. This varies according to your setup, as in how you have exposed the catalyst service. `URL_TOKEN` will the token you created in the step where you generated your secrets (secrets.md#token).

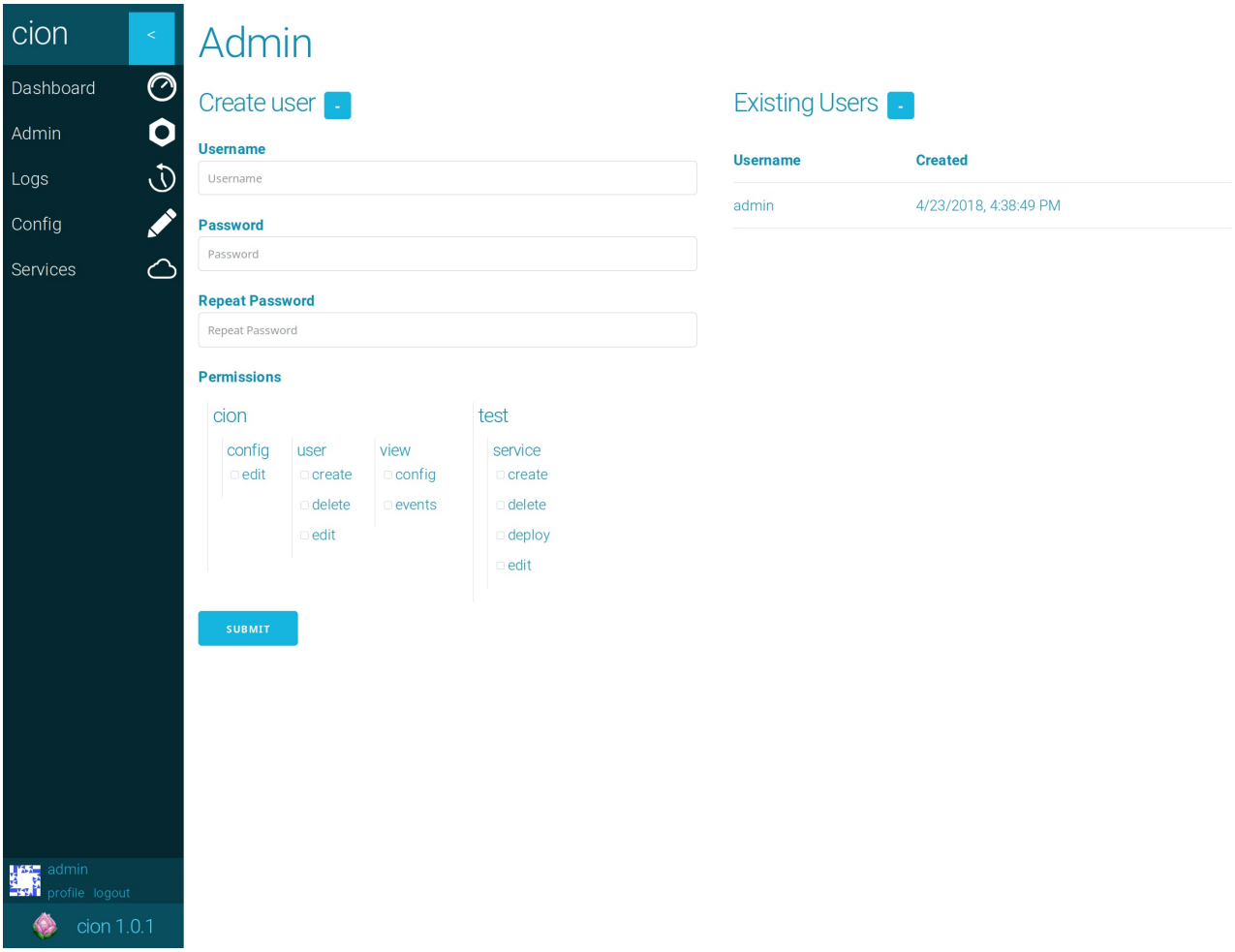
Docker registry

@ Kenan

User management

When cion starts for the first time a admin user will be created with the username and password **admin**, it's necessary to change this password *immediately*.

Navigate to the admin page from the menu on the left. You should see a list of existing users. Click on a user open the user settings for that user.



Changing password

To change a users password navigate to the users settings, here you should see a form where you can input the new password.

Changing permissions

To change a users permissions navigate to the users settings, here you should see a series of checkboxes for each available permission.

Deleteing a user

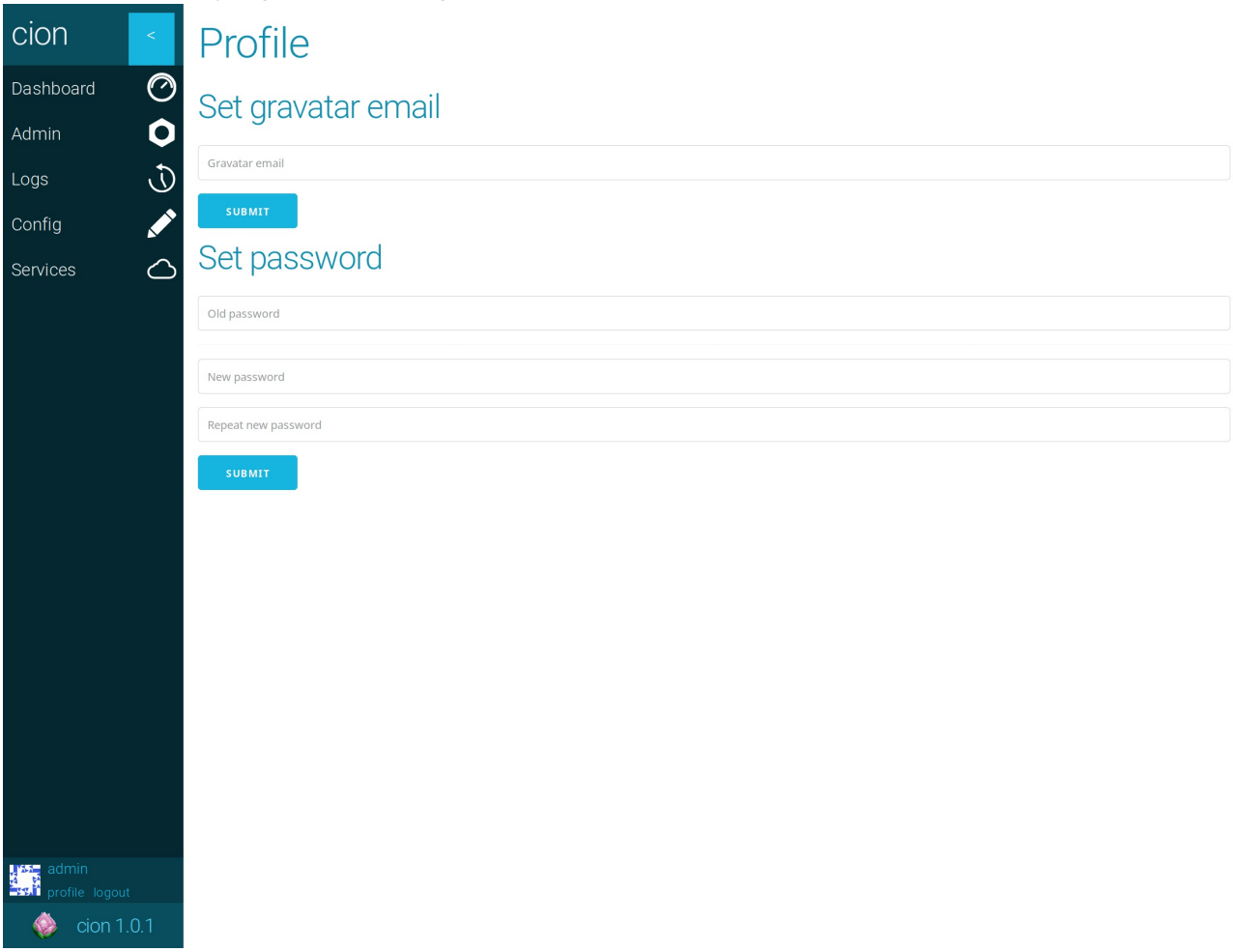
To delete a user navigate to the users settings, here you should see a delete button at the top right of the page.

Creating a user

To create a user navigate to the admin page, here you should see a page with a form for creating a new user.

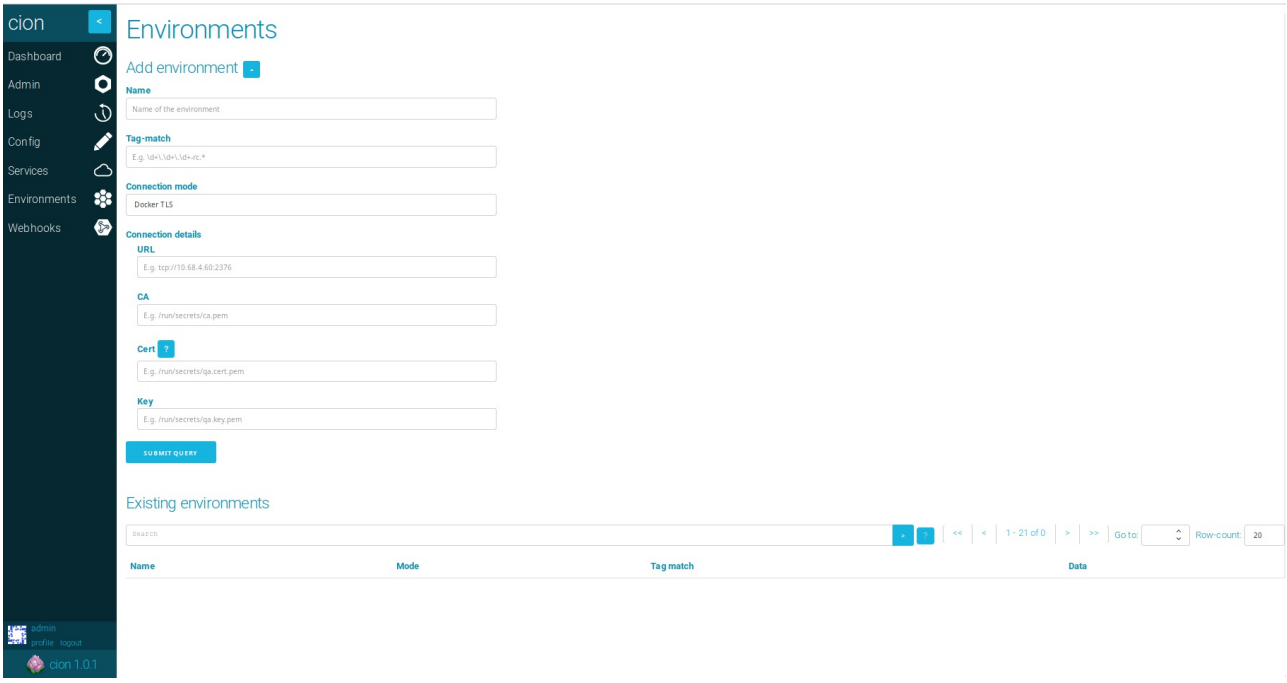
Profile

You can access your own profile by clicking on the *profile* link at the bottom left. Here you can change your own password and set a email account used for your gravatar profile image.



Environment

Navigate to the environments page from the menu on the left. You should see ....



Creating a new environment

To create a new environment navigate to the environments page, here you should see a form for creating a environment. The form consists of the following fields.

Field	Description
Name	The name of the service, e.g. **qa**
Tag-Match	A regex, when new images are pushed the new image's tags must match in order for the images to be deployed to services running in this environment.
Connection mode	Choose one of the connection modes detailed below.

Docker TLS

This connection mode allows you to securely connect to a remote docker environment over TLS. Click here for help (secrets.md#docker) with setting up the secrets. You will need to fill out the following extra fields.

Field	Description
URL	The remote environment's URL, e.g. **tcp://10.68.4.60:2376**
CA	The file with the certificate authority, e.g. **/run/secrets/qa.ca.pem**
Certificate	Client's certificate, e.g. **/run/secrets/qa.cert.pem**
Key	Client's key, e.g. **/run/secrets/qa.key.pem**

Kubernetes service account

This connection mode allows you to securely connect to a remote kubernetes environment over with a service account. Click here for help (secrets.md#kubernetes) with setting up the secrets. You will need to fill out the following extra fields.

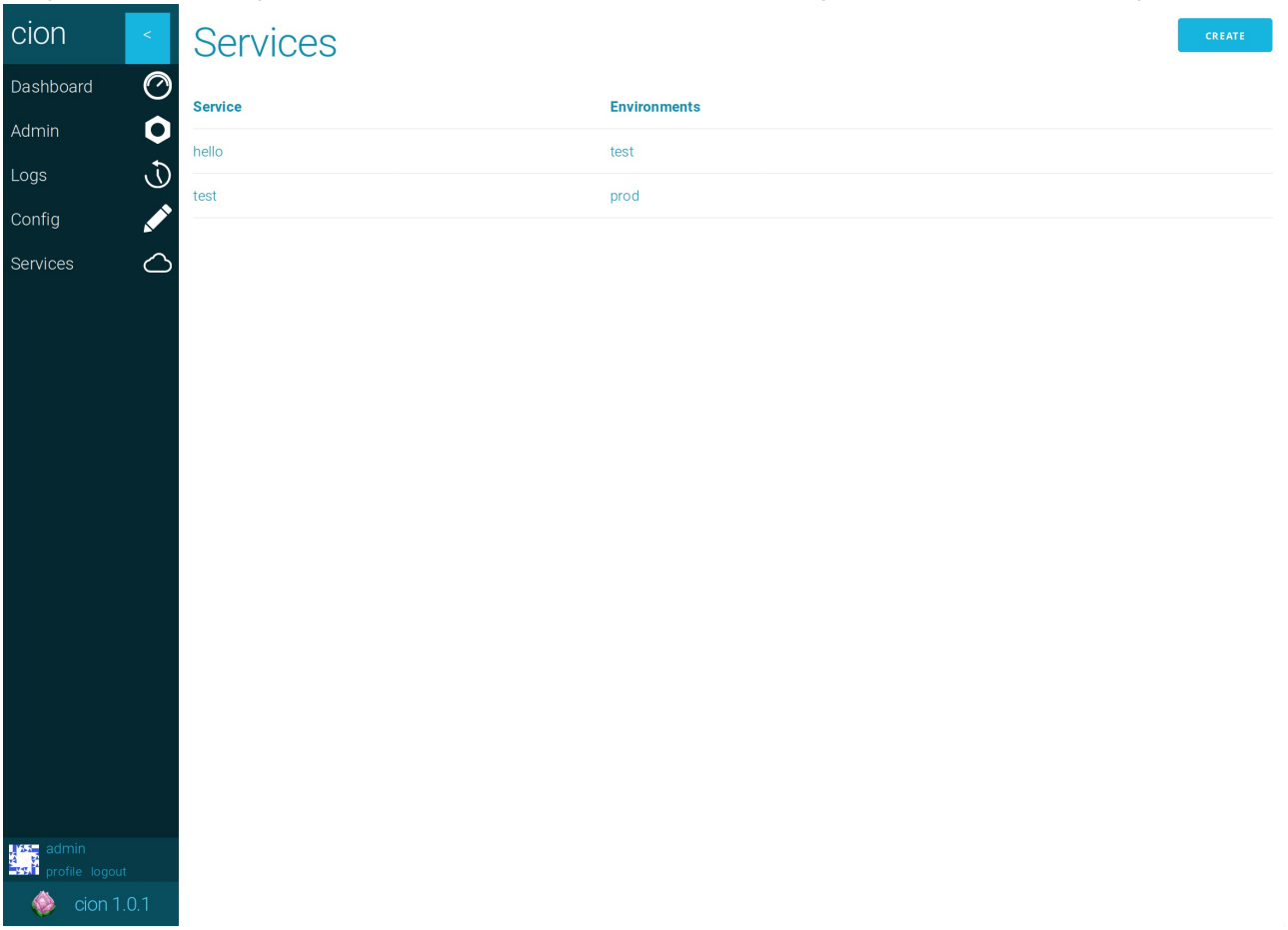
Field	Description
URL	The remote environment's URL, e.g. ****
CA	The file with the certificate authority, e.g. **/run/secrets/qa.ca.pem**
Certificate	Client's certificate, e.g. **/run/secrets/qa.cert.pem**
Key	Client's key, e.g. **/run/secrets/qa.key.pem**

Docker socket

This connection mode allows you to connect to a local docker environment through the docker socket. This mode will connect through */var/run/docker.sock*

Service

Navigate to the services page from the menu on the left. You should see a list of existing services and a button for creating new services.



Creating a new service

To create a service navigate to the services page, here you should see the `create` button in the top right. The form has the three following fields.

Field	Description
Service Name	This field corresponds with the name of the service running in your environments as show by docker service ls, if a service is running in a kubernetes environment the name should correspond with the deployment name.
Environments	Select the list of environments where cion should look for services matching the name field.
Image name	This field should identify the image you wish to update the found services with, the format is: `[repo]/[image]` e.g. `cion/web` (NB tags should not be specified here, it is up the the individual environments to accept or reject tags)

Repositories

Navigate to the config page from the menu on the left. You can configure repositories in the textbox titled `repos`.

The config is a json structure consisting of a list of docker image host `users`. A `user` consists of the following properties.

User

Field	Optional	Description
user	No	The image host username
repos	Yes	A list of repo objects described below
default_login	Yes	If a repository that is not in the `repos` object is accesses this default value will be used for all repositories under this user
default_glob	Yes	If a repository that is not in the `repos` object is accesses this default value will be used for all repositories under this user

Repo

Field	Optional	Description
repo	No	The name of the repository
login	Yes	A string pointing to a json file with credentials for logging in to `[user]/[repo]`

glob	Yes, default is `(.*)/(.*):(.*)`	A regex used for parsing incoming images, there should be three groups in the regex. The first group matches the user, the second group matches the repo, and the third group matches the tag. These matches are used i.e. when a environment accepts or rejects a tag.

Internal webhooks

Cion can trigger webhooks to other services when certain events occurr within cion. The currently supported event-types are:

- service-update: this event triggers when cion updates the image of a service
- new-image: triggers when cion recieves a notification that a new image exists

The following fields configure your webhook. Longer descriptinos are below this list.

- URL: the target URL of the webhook
- Headers: HTTP-headers to include in the request
- Event: what event to trigger on (service-update/new-image)
- Filters/triggers: regex-patterns to match on the event data. If one of these fail the webhook-request is not sent
- Body: the body of the webhook request

Filters

This is a list of name/value pairs. The name is the name of the field in the event data. What fields are contained in the event data varies per event-type.

The `new-image`-event contains these fields:

- image-name: name of the image received
- event: what type of event. `new-image`
- status: what status this task has. Possible values are: ready, processing, done and erroneous
- time: epoch time of when the event was recieved or last updated

The `service-update`-event contains these fields:

- service: name of the service to update
- image-name: name of the image to update the specified services with
- environment: what environment to update the service with the specified service-name
- status: what status this task has. Possible values are: ready, processing, done and erroneous
- event: what type of event. `service-update`
- time: epoch time of when the event was recieved or last updated

What cion does when a new event occurrs is go through all the filters of every webhooks that is configured to fire on that specific event. It will go through each webhook and run a `match` with the regex pattern contained in the `value` on the `name` matching a field in the event data.

An example with the event data:

```
{
  "service": "cion_api",
  "image-name": "cion/api:1.0.0"
}
```

and the filters:

```
{
  "image-name": "^cion\/api:\d.\d.\d$"
}
```

In this case the filter would pass because the regex pattern defined for the field `image-name` is `^cion\/api:\d.\d.\d$`, and that matches the input string `cion/api:1.0.0`.

But if the event data was:

```
{
  "service": "cion_api",
  "image-name": "cion/api:2.0.0-rc"
}
```

-the webhook would not be fired, because the defined pattern does not match the field `image-name` from the event data (`cion/api:2.0.0-rc`).

Body

The \textbf{body} is the body of the HTTP-request to send to the configured URL. It supports the python format-function. So the user can insert fields from the event data into the body by using curly brackets around field-names in the text. For example:

```
{{
  "service-name": "{service}"
}}
```

The above example generates a JSON-body containing exactly one key-value-pair, containing the service-name extracted from the event data.
The user has to escape curly brackets, as shown above, due to how python's format-function tries to interpret them as variable-names.
The above body-string would result in the following when the field "service" in the event data is "cion_api":

```
{
  "service-name": "cion_api"
}
```

This example used JSON as the content type, but any string and formatting is supported.

Prerequisites

- Docker Swarm (https://docs.docker.com/get-started/part4/)
- pip (https://pip.pypa.io/en/stable/installing/) (optional)
- git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)

Setup

Clone the cion repository

```
$ git clone https://github.com/cionkubes/cion
$ cd cion
```

Create secrets that you will need

- Token for webhook url (secrets.md#token)
- Login details for docker repositories that are private. (secrets.md#dockerhub)
- TLS certificates for any docker swarms with services you want to update (secrets.md#docker)
- Kubernetes service user for any kubernetes cluster with deployments you want to update (secrets.md#kubernetes)

Configuring the stack

It is recomended you have some experience with docker compose or docker stack, you can read about it here. (https://docs.docker.com/compose/compose-file/#service-configuration-reference)
You may create your own compose file(example (https://github.com/cionkubes/cion/blob/master/docker/cion-compose.yml)), or you may follow these instructions.

Install the docker stack deploy helper

```
$ pip install git+https://github.com/cionkubes/dsd
```

Create a compose file.

You can choose zero or more profiles. For production it is recommended to run without any profiles, for development the `local` and `live` profiles are recommended.

- local: This profile exposes all the ports necessary for accessing the cion web interface, the rethinkdb web interface and the catalyst. It also mounts the docker daemon socket so that cion can update services in the swarm in which it runs.
- live: Mounts all source code from the host to the containers, to avoid rebuilding the container images on every change. This profile requires that the environment variable CION_ROOT is pointing to a directory containing the github repositories workq (https://github.com/cionkubes/workq), interface (https://github.com/cionkubes/cion-interface), rethink-wrapper (https://github.com/cionkubes/rethink-wrapper), worker (https://github.com/cionkubes/cion-worker), orchestrator (https://github.com/cionkubes/cion-orchestrator), web (https://github.com/cionkubes/cion-web), api (https://github.com/cionkubes/cion-api) and catalyst (https://github.com/cionkubes/cion-catalyst)
- expose-rdb: Exposes the rethink api port 28015 to the host.
- expose-orchestrator: Exposes the orchestrator port 8890 to the host so that external workers can connect to it.

```
$ dsd docker/cion-compose.yml --out [profiles...] > my-stack.yml
```

Modify the compose file

Open *my-stack.yml* in a text editor, and edit the file to suit your needs. At a bare minimum you need to expose the catalyst and web interface services (the local profile exposes the ports to the host for you). You also need to add all of your docker secrets, except the url token, to the worker container.

You can expose the services through a proxy like docker flow proxy (http://proxy.dockerflow.com/swarm-mode-stack/), or, like shown below, you can map the ports to the host. The catalyst needs to be accessible by the image hosting solution, e.g. dockerhub (hub.docker.com) or your docker registry (https://docs.docker.com/registry/), while the web service needs to be accessible by the end users. Both services use port 80 internally.

```
services:
  web:
    ports:
      - 80:80
  catalyst:
    ports:
      - 8080:80
```

Adding the secrets simply requires you to add them to the `services.worker.secrets` list like so.

```
services:
  worker:
    secrets:
      - secret1
      - secret2

secrets:
  secret1:
    external: true
  secret2:
    external: true
```

Starting the stack

Now that we have configured our compose file we can start it up using docker.

```
$ docker stack --compose-file my-stack.yml [stack-name]
```

Eventually you should see the services starting.

```
$ docker service ls
> ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
> hio6xfarzscf        cion_api            replicated          1/1                 cion/api:latest
> y1sqt617s51b        cion_catalyst       replicated          1/1                 cion/catalyst:latest
*:8080->80/tcp
> el0hdw1bbk6z        cion_orchestrator   replicated          1/1
cion/orchestrator:latest
> kri1jv6mnwzm        cion_rethink        replicated          1/1                 rethinkdb:latest
> v5rzpi9v8fn0        cion_rethink-shard  replicated          1/1                 rethinkdb:latest
> iu3292szxf6u        cion_web            replicated          1/1                 cion/web:latest
*:80->80/tcp
> a2v2ea9uzp27        cion_worker         replicated          3/3                 cion/worker:latest
```

Next steps

You should now be able to access the cion web interface and configure (configure.md) cion

cion

A self-hosted solution to automatically update running services in docker swarm or kubernetes

Table of Contents

- Repositories (configure.md#repositories)
- Code Documentation (code.md)

Token

Because dockerhub does not implement authorization in their webhooks, we will need to generate a random string that is used in the catalyst webhook url.

First create a url-safe random string like so

```
$ dd if=/dev/urandom bs=1 count=64 2> /dev/null | base64 --wrap=0 | sed -e 's/+/-/g' -e 's/\//_/g' -e
's/=/~/g'
```

Now store that string in a secure way. You will need this when configuring webhooks in dockerhub or docker registry.

Add the string to docker secrets like this.
NB: if you don't use the default name `url.token`, you have to edit the compose file to reflect this.

```
$ docker secret create url.token [secure-string]
```

Dockerhub

If any repositories in use requires a docker login to pull images from it, you must create a secret containing the username and password of a user with read access to the repository. `$repo` refers to the repository in question.

```
cat << EOF | docker secret create $repo.login.json -
{
    "username": "cion",
    "password": "123456"
}
EOF
```

Docker

In order to add an external docker swarm we need to generate tls certificates. It is important to understand what is going on and to treat the generated files correctly. *They are equivalent to root access to the machine running the external swarm.* You can find the offical guide on securing the docker daemon here (https://docs.docker.com/engine/security/https/).

Generate artifacts

```
$ mkdir tls-certs && cd tls-certs
```

First we need to generate a Certificate Authority (https://en.wikipedia.org/wiki/Certificate_authority) which we will use to sign our certificates. Make sure the common name is the DNS name of the external docker swarm (referred to as `$HOST` from now on).

```
$ openssl genrsa -aes256 -out ca-key.pem 4096
> Generating RSA private key, 4096 bit long modulus
> e is 65537 (0x10001)
> Enter pass phrase for ca-key.pem:
> Verifying - Enter pass phrase for ca-key.pem:

$ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem
> Enter pass phrase for ca-key.pem:
> You are about to be asked to enter information that will be incorporated
> into your certificate request.
> What you are about to enter is what is called a Distinguished Name or a DN.
> There are quite a few fields but you can leave some blank
> For some fields there will be a default value,
> If you enter '.', the field will be left blank.
> -----
> Country Name (2 letter code) [AU]:
> State or Province Name (full name) [Some-State]:Queensland
> Locality Name (eg, city) []:Brisbane
> Organization Name (eg, company) [Internet Widgits Pty Ltd]:Docker Inc
> Organizational Unit Name (eg, section) []:Sales
> Common Name (e.g. server FQDN or YOUR name) []:$HOST
> Email Address []:Sven@home.org.au
```

Generate a server certificate and sign it with our CA.

```
$ openssl genrsa -out server-key.pem 4096
> Generating RSA private key, 4096 bit long modulus
> e is 65537 (0x10001)

$ openssl req -subj "/CN=$HOST" -sha256 -new -key server-key.pem -out server.csr

$ echo subjectAltName = DNS:$HOST,IP:10.10.10.20,IP:127.0.0.1 > extfile.cnf
$ echo extendedKeyUsage = serverAuth >> extfile.cnf

$ openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out
server-cert.pem -extfile extfile.cnf
> Signature ok
> subject=/CN=your.host.com
> Getting CA Private Key
> Enter pass phrase for ca-key.pem:
```

Create the signed client certificates.

```
$ openssl genrsa -out key.pem 4096
> Generating RSA private key, 4096 bit long modulus
> e is 65537 (0x10001)

$ openssl req -subj '/CN=client' -new -key key.pem -out client.csr
$ echo extendedKeyUsage = clientAuth > extfile.cnf

$ openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out
cert.pem -extfile extfile.cnf
> Signature ok
> subject=/CN=client
> Getting CA Private Key
> Enter pass phrase for ca-key.pem:
```

Delete unnecessary files and secure the artifacts.

```
$ rm -v client.csr server.csr
$ chmod -v 0400 ca-key.pem key.pem server-key.pem
$ chmod -v 0444 ca.pem server-cert.pem cert.pem
```

Artifact	Use by cion	Use by external docker swarm
ca.pem	Verify that servers are signed with this CA	Verify that clients are signed with this CA
server-cert.pem	None	Public key signed by CA
server-key.pem	None	Private key used to verify own identity to clients
cert.pem	Public key signed by CA	None
key.pem	Private key used to verify own identity to servers	None

Add the secrets to the docker swarm cion is running in

Cion needs access to the ca.pem, key.pem and cert.pem files from the previous section. `$env` refers to the name of the external swarm e.g. *qa*.

```
$ docker secret create $env.ca.pem /path/to/ca.pem
$ docker secret create $env.key.pem /path/to/key.pem
$ docker secret create $env.cert.pem /path/to/cert.pem
```

Configure external swarm to accept tls connections over https

The external machine needs access to the ca.pem, server-key.pem and server-cert.pem files from the previous section. We need to edit the docker daemon configuration. This is best done by editing the daemon.json (https://docs.docker.com/engine/reference/commandline/dockerd//#daemon-configuration-file) file usually found in `/etc/docker/daemon.json`. If `/etc/docker/daemon.json` does not exist, create it.

Open `/etc/docker/daemon.json` in a text editor. After setting up tls verification the configuration file should resemble this.

```
{
    "tlsverify": true,
    "tlscacert": "/path/to/ca.pem",
    "tlscert": "/path/to/server-cert.pem",
    "tlskey": "/path/to/server-key.pem",
    "hosts": [
        "fd://",
        "0.0.0.0:2376"
    ]
}
```

Unfortunately if you use systemd to start docker, the *hosts* option is already specified in the startup script's command line arguments. Because docker does not support a conflict between command line arguments and `daemon.json`, you need to resolve the conflict. If you are not running docker through systemd, you can skip this step and restart the docker daemon.

Open `/lib/systemd/system/docker.service` in a text editor. We need to modify `daemon.json` and ExecStart such that the hosts are only configured in one of them. In the following example they are kept in `daemon.json` and removed from ExecStart.

```
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network-online.target docker.socket firewalld.service
Wants=network-online.target
Requires=docker.socket
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
- ExecStart=/usr/bin/dockerd -H fd://
+ ExecStart=/usr/bin/dockerd
ExecReload=/bin/kill -s HUP $MAINPID
LimitNOFILE=1048576
# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitNPROC=infinity
LimitCORE=infinity
# Uncomment TasksMax if your systemd version supports it.
# Only systemd 226 and above support this version.
#TasksMax=infinity
TimeoutStartSec=0
# set delegate yes so that systemd does not reset the cgroups of docker containers
Delegate=yes
# kill only the docker process, not all processes in the cgroup
KillMode=process
# restart the docker process if it exits prematurely
Restart=on-failure
StartLimitBurst=3
StartLimitInterval=60s
[Install]
WantedBy=multi-user.target
```

Now restart docker

```
$ systemctl daemon-reload
$ systemctl restart docker
```

Kubernetes

In order for the cluster to authorize cion, we need to create a service account.

```
$ cat > /tmp/serviceaccount.yaml << EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cion
EOF
$ kubectl create -f /tmp/serviceaccount.yaml
> serviceaccount "cion" created
```

Kubernetes should automatically have created a secret token with the name `[serviceaccount]-token-[hash]` for the service account. This secret contains all the information needed to connect to the cluster.

```
$ kubectl get secrets
> NAME                  TYPE                                  DATA      AGE
> cion-token-79hrs      kubernetes.io/service-account-token   3         32d
> default-token-qxcdl   kubernetes.io/service-account-token   3         32d
```

First we will need the decoded CA file so our client can verify the tls connection. We also need the encoded token.

```
$ kubectl get secret cion-token-79hrs --output=json | jq '.data["ca.crt"]' --raw-output | base64 -d |
docker secret create $env.ca.crt -
$ kubectl get secret cion-token-79hrs --output=json | jq '.data.token' --raw-output | docker secret create
$env.token -
```