Docker Project Template Documentation

Release 2

Wolnosciowiec Team

Dec 07, 2020

Contents:

1 Getting started **3**
- 1.1 Read about project Structure . . . 3

2 Structure **5**
- 2.1 .rkd . . . 5
- 2.2 apps/conf/ . . . 5
- 2.3 apps/conf.dev/ . . . 5
- 2.4 apps/profile/ . . . 5
- 2.5 apps/healthchecks/ . . . 5
- 2.6 apps/repos-enabled/ . . . 6
- 2.7 apps/www-data/ . . . 6
- 2.8 containers/ . . . 6
- 2.9 data/ . . . 6
- 2.10 hooks.d/ . . . 6

3 Basic commands **7**
- 3.1 Installing applications from Snippet Cooperative . . . 7
- 3.2 Starting, upgrading and stopping services . . . 7
- 3.3 Controlling maintenance mode on production . . . 8
- 3.4 Diagnosing issues (advanced usage on production) . . . 8

4 Creating first service **9**
- 4.1 Generating a service from template using Cooperative . . . 9
- 4.2 Creating a service - the manual way . . . 9
- 4.3 Bringing up created service . . . 10
- 4.4 Read more about Creating practical services with volumes, subdomains, passing credentials . . . 10

5 Creating practical services with volumes, subdomains, passing credentials **11**
- 5.1 Connecting domains, subdomains and optionally SSL . . . 11
- 5.2 Passing credentials and configuration options . . . 12
- 5.3 Volumes . . . 12

6 Understanding architecture **15**

7 Profiles - matching services by query **17**
- 7.1 Example . . . 17
- 7.2 Syntax . . . 17

8 Deploying to production with Ansible **19**
8.1 Concept - Single key to all credentials 19
8.2 Getting started with Harbor deployments 20
8.3 Advanced usage 20

9 From authors **23**

docker-compose based framework for building production-like environments - developing and testing on your local computer, deploying to your server or cluster from shell or from CI. **Harbor is a pre-configured set of most popular technologies available to use with docker-compose, in addition of our exclusive features**

Features:

- Service discovery (pins containers into WWW domains by labelling)
- Deployment strategies: compose's standard, recreation, and **rolling-updates (zero-downtime updates)**
- Automatic Letsencrypt SSL
- Standardized directory structures and design patterns
- Ready to use snippets of code and solutions
- Ansible integration to prepare your production/testing server and deploy updates in extremely intuitive way

CHAPTER 1

Getting started

1. Install Harbor

```
pip install rkd-harbor
```

2. Create GIT project

```
mkdir my-project
cd my-project

git init
```

3. Create Harbor project

```
harbor :create:project
```

1.1 Read about project Structure

CHAPTER 2

Structure

Project consists of a standard structure which includes:

2.1 .rkd

RiotKit-Do directory, where you can define custom tasks, there are also temporary files and logs stored (ADVANCED)

2.2 apps/conf/

docker-compose YAML files with definitions of containers, networks and volumes

2.3 apps/conf.dev/

Same as apps/conf, but enabled only on development environment

2.4 apps/profile/

Defined service profiles that allows to select services on which you operate in given command (eg. wordpress profile = all instances of wordpress)

2.5 apps/healthchecks/

RiotKit's InfraCheck integration, here are placed all of the healthcheck definitions (see section about health and monitoring)

2.6 apps/repos-enabled/

GIT repositories definitions (see section about applications from external GIT repositories)

2.7 apps/www-data/

Cloned applications from GIT (see section about applications from external GIT repositories)

2.8 containers/

Configuration data mounted via bind-mount to inside containers (should be read-only and versioned by GIT)

2.9 data/

Bind-mounted volume storage for containers, only data that is generated dynamically by containers is stored there.

Example use cases:

- Database data eg. /var/lib/mysql
- Generated SSL certificates storage
- NGINX generated configurations

2.10 hooks.d/

Scripts that are executed at given time in the Harbor lifecycle (eg. pre-start, post-start, pre-upgrade, ...) See section about hooks.

```
hooks.d/
hooks.d/pre-upgrade
hooks.d/(...)
hooks.d/post-start
```

2.10.1 Keeping standards

KISS - keep it simple stupid

By keeping standards in your project you make sure, that any person that joins your project or a contributor could be satisfied with Harbor documentation. Any outstanding solutions would require you to create extra documentation in your project.

2.10.2 OK, got it, let's learn Basic commands

CHAPTER 3

Basic commands

Harbor defines a lot of RKD tasks, that automates preparing changes to project as well as operating on live organism.

At first we would like you to get familiar with our Snippet Cooperative, which is a place to share a code with others - for you now it means you can install an application with a single command.

3.1 Installing applications from Snippet Cooperative

Browse the catalogue of applications at: https://github.com/riotkit-org/harbor-snippet-cooperative Then, perform an installation within a single command.

```
# at first do a repository sync, later you don't need to do it all the time
harbor :cooperative:sync :cooperative:install harbor/redis
```

3.2 Starting, upgrading and stopping services

Basic tasks lets you control the services running in your environment, just like you were doing before with **docker ps** but with a difference that Harbor interface is more domain-focused interface.

```
# create and start containers
harbor :start

# pull new images, update git repositories, then start
harbor :upgrade

# start selected service
harbor :service:up hello

# remove selected service
harbor :service:rm hello
```

(continues on next page)

(continued from previous page)

```
# list all services, running and not running
harbor :service:list

# check a report of running service
harbor :service:report hello
```

3.3 Controlling maintenance mode on production

Sometimes, when bad things happens, or a scheduled repair is planned a maintenance mode is required. Harbor provides a simple maintenance mode in 3 ways: global, per-service, per-domain

```
# maintenance mode per single service
harbor :maintenance:on --service hello

# per single domain
harbor :maintenance:on --domain domain-name.org

# global maintenance mode - for all services
harbor :maintenance:on --global
```

3.4 Diagnosing issues (advanced usage on production)

```
# do the docker-compose ps, in case you need
harbor :diagnostic:compose:ps

# in case you need a full docker-compose arguments used by Harbor to execute some␣
↪commands manually
harbor :diagnostic:dump-compose-args

# dump all yamls to big one for analysis
harbor :diagnostic:compose:config

# force regenerate all Letsencrypt certificates (use with caution, there are limits␣
↪of hits on Letsencrypt)
harbor :gateway:ssl:regenerate

# reload gateway in case, when the the nginx.tmpl was modified
harbor :gateway:reload
```

CHAPTER 4

Creating first service

Service definitions are docker-compose.yml files, with addition of Harbor's patterns which allows to automate and standardize the way of environment preparation.

Few rules:

- YAML files are stored at `./apps/conf`
- The naming: `apps.MY-APP-NAME.yaml` for applications, and `infrastructure.MY-TECHNICAL-APP-NAME.yaml` for technical services (health checks, backups etc.)
- Volumes with configuration files eg. nginx.conf - should be in `./container/MY-APP-NAME` directory
- Volumes with external git repositories should be in `./apps/www-data/MY-APP-NAME` directory
- Volumes with dynamic data such as user uploads should be in `./data/MY-APP-NAME` directory

4.1 Generating a service from template using Cooperative

Best way to create a service is to use a generator - to avoid common mistakes.

Demo: https://asciinema.org/a/348867

```
harbor :cooperative:sync
harbor :cooperative:install harbor/webservice
```

The below example will sync coop repositories, then use `harbor/webservice` template to generate docker-compose yaml file, that will be placed in `./apps/conf` directory.

4.2 Creating a service - the manual way

Create a standard docker-compose format file in `./apps/conf` directory, name it properly eg. `apps.adminer.yml` and put following example contents:

```
version: 2.3
services:
    adminer:
        image: adminer
        restart: always
        environment:
            VIRTUAL_HOST: db.example.localhost
            VIRTUAL_PORT: "80"
            LETSENCRYPT_HOST: db.example.localhost
            LETSENCRYPT_EMAIL: example@example.org
        labels:
            org.riotkit.updateStrategy: "rolling"
```

4.3 Bringing up created service

Use `:service:up` task to bring up a recently created service.

```
harbor :service:list
harbor :service:up service-name
```

After checking that everything works correctly the service definition + configuration files placed in `./container` directory should be pushed to GIT.

4.4 Read more about Creating practical services with volumes, sub-domains, passing credentials

CHAPTER 5

Creating practical services with volumes, subdomains, passing credentials

Harbor is a complete framework for building flexible multi-container environments, to complete it's mission Harbor provides set of tools and patterns described in this documentation chapter.

5.1 Connecting domains, subdomains and optionally SSL

Domains and subdomains are automatically discovered by JWilder's Docker-Gen, when a container is started.

Docker-gen container, later called **"service discovery"** collects environment variables - including **VIRTUAL_HOST** and **VIRTUAL_PORT** for each running container, then generates NGINX configuration file and calls reload.

Similar mechanism is practiced by docker-letsencrypt-nginx-proxy-companion to automatically connect Let's Encrypt certificate - **LETSENCRYPT_HOST** and **LETSENCRYPT_EMAIL** environment variables are required to do so.

Example:

```
version: 2.4
services:
    app_web_mattermost:
    image: mattermost/mattermost-prod-web:5.23.2
    depends_on:
        - app_mattermost
    environment:
        APP_HOST: "app_mattermost"
        APP_PORT: "8000"

        # gateway configuration
        VIRTUAL_HOST: "mattermost.${MAIN_DOMAIN}${DOMAIN_SUFFIX}"
        VIRTUAL_PORT: "80"
        LETSENCRYPT_HOST: "mattermost.${MAIN_DOMAIN}${DOMAIN_SUFFIX}"
        LETSENCRYPT_EMAIL: "${LETSENCRYPT_EMAIL}"
```

```
MAIN_DOMAIN=riotkit.org
DOMAIN_SUFFIX=.localhost
LETSENCRYPT_EMAIL=noreply@riotkit.org
```

MAIN_DOMAIN, DOMAIN_SUFFIX and LETSENCRYPT_EMAIL convention

- Use MAIN_DOMAIN to specify a main domain if hosting services under multiple subdomains
- DOMAIN_SUFFIX, on development environment set to ".localhost" - in result on Linux you will be able to access services like on production but under localhost sudomain eg. my-subdomain.riotkit.org.localhost. Please note: When using `harbor :deployment:apply` the DOMAIN_SUFFIX is automatically erased when deploying to production server
- LETSENCRYPT_EMAIL allows to have a globally defined e-mail address for all services

5.2 Passing credentials and configuration options

Most universal way to configure services is to pass environment variables. Passwords, sensitive data and common values shared between services put in `.env` file, then encrypt it using Ansible Vault command `harbor :env:encrypt`. In result a `.env-prod` file will be produced. Don't commit `.env` to git - add it to ignore, commit `.env-prod` instead.

When deploying to production server with `harbor :deployment:apply` mechanism the `.env-prod` will be decrypted on-the-fly and placed as `.env` on the destination server.

```
version: 2.4
services:
    postgres:
        image: postgres:12.4
        environment:
            POSTGRES_USER: "postgres"
            POSTGRES_PASSWORD: "${DB_PASSWD}"
            POSTGRES_DATABASE: "mydb"
        expose:
            - 5432
        volumes:
            - ./data/pg:/var/lib/pgsql
```

```
DB_PASSWD=my-passwd
```

Note: `.env` is read by docker-compose and by RKD in makefile.yaml by default. It is a good place to put your configuration options

5.3 Volumes

In previous chapter we were talking about naming conventions, remember? There is a distinction for static and dynamic volumes.

- Static volumes are kept in GIT repository, those are usually versioned configuration files
- Dynamic volumes are application data (database binary files, user file uploads)

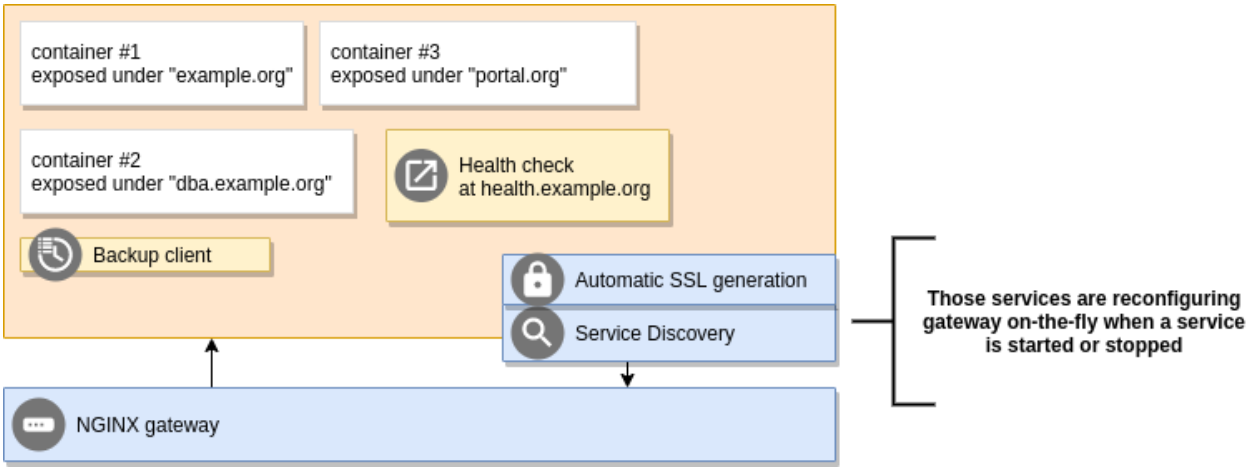
```
version: 2.4
services:
    my-website:
        image: nginx:1.19
        environment:
            VIRTUAL_HOST: "my-website.localhost"
            VIRTUAL_PORT: "80"
        volumes:
            # in www-data we keep other cloned git repositories managed by Harbor
            - ./apps/www-data/my-website:/var/www/html
            - ./container/my-website/nginx.conf:/etc/nginx/nginx.conf:ro

    postgres:
        image: postgres:12.4
        environment:
            POSTGRES_USER: "postgres"
            POSTGRES_PASSWORD: "${DB_PASSWD}"
            POSTGRES_DATABASE: "mydb"
        expose:
            - 5432
        volumes:
            - ./data/pg:/var/lib/pgsql
```

CHAPTER 6

Understanding architecture

Harbor imposes the architecture of a centralized gateway for web services. It is just like in a cloud, or in Kubernetes - the centralized Ingress/Webserver takes the traffic and routes it to other services.



Advantages:

- Easy to manage, and to maintain, one router to reload
- Integration with services such as LetsEncrypt without additional work to be done
- Most popular architecture for hosting multiple services
- SSL termination at the router edge makes SSL support almost transparent to applications

CHAPTER 7

Profiles - matching services by query

Harbor 2.0 introduced Service Profiles to make operating only on selected services possible. The profiles are selectors that picks services you want to operate on.

Benefits

- Secure. Services that should not be touched are not touched
- Handy. Can be used in `harbor :deployment:apply` task when deploying to production environment to update only part of services (eg. all instances of data collecting application)
- Flexible. The syntax of the filter is pure Python, you can create as much advanced queries as far as you would be able to understand them :)

7.1 Example

Given we have a "gateway" selector, that picks all services that name begins with "gateway_"

apps/profile/gateway.profile.py

```
name.startswith('gateway\_')
```

Now we can use it in all service management and environment stop/start tasks, for example `harbor :start --profile=gateway`

7.2 Syntax

Variable	Description
name	Name of the service (string)
service	Service attributes, docker-compose definition (dict)

Notes:

- Always check every node in dictionary for existence - Example: 1) labels, 2) labels.some-label

Advanced example:

```
"labels" in service and "org.riotkit.group" in service['labels'] and service['labels
↪']['org.riotkit.group'] == "database"
```

CHAPTER 8

Deploying to production with Ansible

8.1 Concept - Single key to all credentials

Harbor 2.0 standardizes the way of deploying itself to production servers, introducing a simplified deployment from single repository with one passphrase for all secrets.

Deployment mechanism is installing Harbor + dependencies from requirements.txt, cloning the repository, setting permissions, adding autostart with systemd and starting the project. Please note, that it requires **all changes to be committed to git repository** before starting `harbor :deployment:apply` command.

Encrypted deployment.yml file can contain ssh passwords, ssh private key. It's safe to store it in repository - Ansible Vault is using strong AES encryption

```
deploy_user: my-deployment-user
deploy_group: my-deployment-user

# Directory, where the project will be installed
remote_dir: /home/my-deployment-user/project

# Target repository to clone (in most cases it should be the same repository as␣
↪current one)
# leave commented for automatic detection
#git_url: git@github.com:your-org/your-repo.git

# Secret url is helpful, when you cannot setup working ssh-agent. Secret url is used␣
↪only at deployment time, later
# a regular URL (without credentials) is leaved on the machine
#git_secret_url: https://user:password@github.com/your-org/your-repo.git

# Will make a file in /etc/sudoers.d/ to allow ssh-agent passing into sudo session
configure_sudoers: true

nodes:
    production:
```

(continues on next page)

(continued from previous page)

```
        - host: remote-host.org
          port: 2222
          user: my-deployment-user
          sudo_password: my-sudo-password

          # select between password or key-based authentication
          password: my-password
          private_key: |
              -----BEGIN OPENSSH PRIVATE KEY-----
              (.............................)
              -----END OPENSSH PRIVATE KEY-----
```

8.2 Getting started with Harbor deployments

First time you need to download a required Ansible role and optionally generate an example deployment.yml file

```
harbor :deployment:files:update :deployment:create-example
```

Now fill up **deployment.yml** file, then perform a test deployment.

```
# tip: use --ask-vault-pass if you encrypt .env file
# tip: you need to have all changes (except deployment.yml - you can hold with this␣
↪file) committed to repository before running deployment
harbor :deployment:apply
```

When deployment ran smoothly and you are sure that's pretty all, then encrypt deployment.yml

```
# tip: Use same key as in .env file to make it simpler
harbor :vault:encrypt deployment.yml
```

8.3 Advanced usage

Use switches and environment variables to customize playbook name, inventory name, to pass Ansible Vault password, to ask for user ssh login or ssh password.

```
# ask interactively for sudo password
harbor :deployment:apply --ask-sudo-pass

# provide a vault password in alternative way
VAULT_PASSWORDS="oh-thats-secret" harbor :deployment:apply

# another way to provide vault password
echo 'VAULT_PASSWORDS="oh-thats-secret"' > /mnt/secret-encrypted-storage/.secret-env
source .secret-env && harbor :deployment:apply


# run witha custom playbook (place it in .rkd/deployment/
PLAYBOOK="my-playbook.yml"  harbor :deployment:apply

# deploying from a custom branch instead of "master"
harbor :deployment:apply --branch primary
```

(continues on next page)

(continued from previous page)

```
# providing a key for GIT clone used to setup project repository on target machine
harbor :deployment:apply --git-key="~/.ssh/id_rsa"
```

CHAPTER 9

From authors

Project was started as a part of RiotKit initiative, for the needs of grassroot organizations such as:

- Fighting for better working conditions syndicalist (International Workers Association for example)
- Tenants rights organizations
- Various grassroot organizations that are helping people to organize themselves without authority
- Grassroot groups fighting for democratic rights

RiotKit Collective