freight-forwarder Documentation

Release 1.0.0

Alex Banna

February 19, 2016

Freight Forwarder is a utility that uses Docker to organize the transportation and distribution of Docker images from the developer to their application consumers. It allows the developer to focus on features while relying on Freight Forwarder to be the expert in continuous integration and continuous delivery.

The project website can be reference for the following information. Please report any bugs or feature requests on Github Issues.

CHAPTER 1

Introduction

1.1 General Overview

Freight Forwarder focuses on continuous integration and continuous delivery. At first glance it looks and feels a lot like Fig/Compose. However, Fig/Compose are very focused on the developers workflow and easing the pain of multiple container environments. Freight Forwarder can be used to accomplish that same task and much more. Freight Forwarder focuses on how Docker images are built, tested, pushed, and then deployed. Freight Forwarder uses an image based CI/CD approach, which means that the images being pushed to the registry are the artifacts being deployed. Images should be 100% immutable, meaning that no additional changes should need to be made to the images after being exported. It is expected that containers will be able to start taking traffic or doing work on initialization. When deploying from one environment to the next, the image from the previous environment will be pulled from the registry and configuration changes will be made and committed to a new image. Testing will be run with the new configuration changes. After the image is verified, it will be pushed up to the registry and tagged accordingly. That image will then be used when deploying to that environment.

Freight Forwarder works on Docker version 1.8, API version 1.20.

Please review the *project integration* documentation to start integrating your project with Freight Forwarder.

1.1.1 Configuration File

The configuration file defines your CI/CD pipeline. The definition of the manifest is something developers will have to define to support their unique workflow. This empowers the developer to define the CI/CD pipeline without interaction with an operations team.

Configuration file documentation

Warning: The *configuration file* is required if your planning to use the *CLI*.

1.1.2 SDK

Freight forwarder is an SDK that interacts with the docker daemon api. The SDK provides and abstraction layer for CI/CD pipelines as well as the docker api itself. The SDK allows developers to use or extend its current functionality.

SDK documentation

1.1.3 CLI

Freight Forwarder CLI consumes the SDK and provides an easy to use interface for developers, system administrators, and CI services. The CLI provides all of the functionality required to support a complete CI/CD workflow both locally and remote. If a project has a manifest the cli provides an easy way to override values without having to modify the manifest itself.

CLI documentation

1.1.4 Injector

Freight Forwarder plays a roll in the injection process. It will pull an Injector Image from a registry then create and run the container. The Injector shares whatever files that need to be injected with freight forwarder with a shared volume. Freight Forwarder then copies, chowns, and chmods the files into the application image based on the metadata provided in the injectors response.

Injector documentation

1.2 Install Freight Forwarder

1.2.1 OSX Install

Requirements:

- Python 2.7
- pip, setuptools, and wheel Python packages.
- libyaml. This can be installed via brew.

Install via pip:

```
$ pip install freight-forwarder
```

1.2.2 Ubuntu Install

Ubuntu 14.10:

> wget https://bootstrap.pypa.io/get-pip.py sudo python get-pip.py aptitude update && sudo aptitude remove libyaml-dev pip install libyaml sudo pip install freight-forwarder freight-forwarder

1.2.3 Arch Linux Install

Arch 4.2.3-1-ARCH:

Because Arch installs python3 as the default python, it is strongly suggested installing pyenv and using that to manage the local python version.

> # Set the local version to a freight forwarder compatible version pyenv local 2.7.9 # Install setuptools wget https://bootstrap.pypa.io/ez_setup.py -O - | python # Install pip deps pip install wheel # Install freight forwarder pip install freight-forwarder freight-forwarder info

1.2.4 CentOS install

When we install this on CentOS we will need to update these docs.

1.3 Project Integration

1.3.1 Overview

Before being able to use Freight Forwarder there must be a Dockerfile in the root of your project. The Project Dockerfile is a standard Dockerfile definition that contains the instructions required to create a container running the application source code. The Project Dockerfile must container an entrypoint or cmd to start the application.

If the project has tests a second Dockerfile should be built. This test Dockerfile should reside in the root of the application tests directory and inherit from the Project Dockerfile. The test Dockerfile should contain instructions to install test dependencies and have an entrypoint and command that will run the entire applications test suite. The tests should return a non zero on failure.

If there are dependencies or base image Dockerfiles they can live anywhere in your projects and can be referenced in any service definition, via the *build: path*. This allows for more complex projects to be managed with one configuration file.

Example Project Dockerfile:

```
FROM  ubuntu:14.04
MAINTAINER John Doe "jdoe@nowhere.com"
ENV REFRESHED_AT 2015-5-5

RUN apt-get update
RUN apt-get -y install ruby rake

ADD ./ /path/to/code

ENTRYPOINT ["/usr/bin/rake"]
CMD ["start-app"]
```

Example Test Dockerfile:

```
FROM docker_registry/ruby-sanity:latest
MAINTAINER John Doe "jdoe@nowhere.com"
ENV REFRESHED_AT 2014-5-5

RUN gem install --no-rdoc --no-ri rspec ci_reporter_rspec
ADD ./spec /path/to/code/spec
WORKDIR /path/to/code
ENTRYPOINT ["/usr/bin/rake"]
CMD ["spec"]
```

1.3.2 Namespacing

Freight Forwarder is a bit opinionated about namespaces. The namespace for images map to the pre-existing docker naming conventions. Team/Project map directly to Dockers repository field.

Example Docker namespace:

```
repository/name:tag
```

Example Freight Forwarder namespace:

```
team/project:tag
```

1.3.3 Tagging

When tagging your images Freight Forwarder will use the data center and/or environment provided in the configuration file. Freight Forwarder will prepend those when tagging images.

Example tag:

```
datacenter-environment-user_defined_tag
```

Real Life Example:

```
us-east-1-development-beta
```

1.3.4 Configuration File

The *Configuration File* is required and is the what the consumer uses to define their pipeline.

1.3.5 Configuration Injection Integration

If there is interest in integrating with the injector please start by referring to the *Injector*.

1.3.6 Example Projects

- **‘Docker Example‘_**
- **‘CIApi‘_**

1.3.7 Jenkins Integration

1.4 Workflows

1.4.1 Overview

The following section will define multiple methods for the building, exporting, and deploying of containers with freight-forwarder. It will be broken down into examples by individual scenarios to allow for expansion. The scenarios will assume that standard root keys are in the configuration file are present.

There are some best practices to follow when iterating on the *freight-forwarder.yaml* configuration file.

1. When defining the Dockerfile, add all source code near the end of the Dockerfile to promote the use of cached images during development. Use finalized images for configuration injection or build without using cache. This reduces any potential issues associated with cached images leaving traces of previous builds.
2. Reduce the amount of dependencies that are installed in the final image. As an example, when building a java or go project, separate the *src* or *build* container into a separate container that can provide the go binary or jar for consuming in another container.

3. Begin the Dockerfile with more *RUN* directives, but once it is tuned in, combine the statements into one layer.

Example:

```
RUN ls -la
RUN cp -rf /home/example /example
# configures this into one layer if possible
RUN ls -la \
    && cp -rf /home/exampe /example
```

4. Examine other projects. Determine if the image needs to be more dynamic and to be utilized for multiple definitions or purposes. For example, an elasticsearch node can be defined as a master, data, or client node. These are configuration changes that can be changed by environment variables. Is this needed to fulfill the specification or will there exist defined images for different nodes that need to remain complete without a dynamic nature?

1.4.2 Scenario #1 - Single Service No Dependencies

THe service below requires no dependencies (external services) and can run entirely by itself.

configuration:

```
api:
  build: ./
  ports:
    - "8000:8000"
  env_vars:
    - ...
```

1.4.3 Scenario #2 - Single Service with Cache

The service requires memcach/redis/couchbase as a caching service. When locally deployed or in quality-control, this will allow for the defined *cache* container to be started to facilitate the shared cache for the api.

configuration:

```
api:
  build: ./
  ports:
    - "8000:8000"
  env_vars:
    - ...

cache:
  image: default_registry/repo/image:<tag>
  ports:
    - "6379:6739"

environments:
  development:
    local:
      hosts: ...
      api:
        links:
          - cache
```

This would suffice for most local development. But what happens you need to run a container with a defined service that is in staging or production? You can define the service as a separate dependency that is pre-configured to meet the specs for your service to operate. Ideally, this should be **configured** as a Dockerfile inside your project. This provides the additional benefit of providing a uniform development environment for all develops to work in unison on the project.

export configuration:

```
staging_hostname:
  image: default_registry/repo/image:tag
  ports:
    - "6379:6379"

environments:
  development:
    use01:
      export:
        api:
          image: default_registry/repo/baseimage_for_service:tag
          links:
            - staging_hostname
          # or
          extra_hosts:
            - "staging_hostname:ip_address"
            - "staging_hostname:ip_address"
          # or
          extra_hosts:
            staging_hostname: ip_address
```

1.4.4 Scenario #3 - Single Service with Multiple Dependencies

This would be an example of a majority of services that required multiple dependencies for a variety of reasons. For example, it might require a shared cache with a database for relational queries, and an ElasticSearch cluster for analytics, metrics, logging, etc.

configuration:

```
esmaster:
  ...
esdata:
  links:
    - esmaster
api:
  links:
    - esdata
    - mysql
    - cache
nginx:
  env_vars:
    - "use_ssl=true"
mysql:
  ...
cache:
  ...

environments:
  development:
    quality-control:
      nginx:
        links:
          - api
```

When *quality-control* or *deploy* is performed as the action, this will start all associated containers for the service. Internally, all *dependents* and *dependencies* will be analyzed and started in the required order. The list below represents the order in which containers will be created and started.

1. mysql or cache
2. cache or mysql
3. esmaster
4. esdata
5. api
6. nginx

When attempting to export a service, all dependencies will be started; but no dependents. For example, if attempting to export the *api*, *mysql*, *cache*, *esmaster* and then *esdata* will be started before the api is built from the Dockerfile or the image is pulled and started.

General Overview A general description of the project. Something to get you acquainted with Freight Forwarder.

Install Freight Forwarder How do I install this thing? Some simple install instructions.

Project Integration How do I integrate Freight Forwarder with my project? Explanation of how to integrate with Freight Forwarder, expectations, and a few examples.

Workflows Examples of different implementations for a variety of services. Single Service definition, Single Server with One dependency, Multi-dependency services, etc.

CHAPTER 2

Basic Usage

2.1 Configuration File

2.1.1 Overview

This is the blueprint that defines an applications CI/CD workflow, container configuration, container host configuration, and its dependencies per environment and data-center. The file should be written in yaml (json is also supported). The objects in the configuration file have a cascading effect. Meaning that the objects defined deepest in the object structure take precedent over previously defined values. This allows for common configuration values to be shared as well as allowing the flexibility to override values per individual operation.

Warning: Configuration injection isn't included in this configuration file.

2.1.2 Terminology Explanation

Definitions:

freight-forwarder.yml	Handles the organization of application services, environments and data-centers.
hosts	A server physical/virtual server that has the docker daemon running. The daemon must be configured to communicate via tcp.
service	Multiple services are defined to make a project. A service could be an api, proxy, db, etc.
environments	An object that is used to define where/how containers and images are being deployed, exported, and tested for each environment.
data centers	An object that is used to define where/how containers and images are being deployed, exported, and tested for each data center.
registry	The docker registry where images will be pushed.

2.1.3 Root Level Properties

All of the properties at the root of the configuration file either correlate to a service, project metadata, or environments.

Name	Required	Type	Description
team	True	string	Name of the development team.
project	True	string	The project that is being worked on.
repository	True	string	The projects git repo.
services	True	object	Refer to *Service Properties*
registries	False	object	Refer to *Registries Properties*
environments	True	object	Refer to *Environments properties*

```yaml
---
# team name
team: "itops"

# current project
project: "cia"

# git repo
repository: "git@github.com:Adapp/cia.git"

# Service definition
api:
  build: "./"
  test: "./tests/"
  ports:
      - "8000:8000"

# environments object is collection of environments and data centers
environments:

  # development environment
  development:

    # local datacenter
    local:

    # sea1 data center
    sea1:

```

```
  # staging environment
  staging:

  # production environment
  production:
```

2.1.4 Service Properties

Each service object is a component that defines a specific service of a project. An example would be an api or database. Services can be built from a Dockerfile or pulled from an image in a docker registry. The container and host configuration can be modified on a per service bases.

Name	Required	Type	Description
build	one of	string	Path to the service Dockerfile.
test	False	string	Path to a test Dockerfile that should be used to verify images before pushing them to a docker registry.
image	one of	string	The name of the docker image in which the service depends on. If its being pulled from a registry the fqdn must be provided. Example: *registry/itops/cia:latest*. If the image property is spectified it will always take precedent over the build property. If a service object has both an image and build specified the image will exclusively be used.
export_to	False	string	Registry alias where images will be push. This will be set to the default value if nothing is provided. The alias is defined in *Registries Properties*
Container Config	any of		Refer to *Container Config Properties*

```
---
# service alias.
couchdb:

  # Docker image to use for service.
  image: "registry_alias/itops/cia-couchdb:local-development-latest"

  # Path to Dockerfile.
  build: ./docker/couchdb/Dockerfile

  # Synonyms with -d from the docker cli.
  detach: true

  # Synonyms with -p from the docker cli.
  ports:
    - "6984:6984"
    - "5984:5984"
```

2.1.5 Registries Properties

The registries object is a grouping of docker registries that images will be pulled from or pushed to. The alias of each registry can be used in any image definition *image: docker_hub/library/centos:6.6*. By default docker_hub is provided for all users. The default property will be set to docker_hub unless overridden with any of the defined registries.

Name	Required	Type	Description
registry (alias)	True	object	Refer to *Registry Properties*
default	False	object	Refer to *Registry Properties*

```
# define
registries:
  # define development registry
  tune_development: &default_registry
    address: "https://example_docker_registry"
    verify: false

  # define production registry
  tune_production:
    address: "https://o-pregister-sea1.ops.tune.com"
    ssl_cert_path: /path/to/certs
    verify: false

    # define auth for production registry
    auth:
        type: "registry_rubber"
        address: "https://o-regrubber-sea1.ops.tune.com"
        ssl_cert_path: /path/to/certs
        verify: false

  # define default registry. If this isn't defined default will be docker_hub.
  default: *default_registry
```

2.1.6 Registry Properties

The docker registry that will be used to pull or push validated docker images.

Name	Required	Type	Description
address	True	string	Address of docker host, must provide http scheme. Example: https://your_dev_box.office.priv:2376
ssl_cert_path	False	string	Full system path to client certs. Example: /etc/docker/certs/client/dev/
verify	False	bool	Validate certificate authority?
auth	False	object	Refer to *Registry Auth Properties*

```
---
registries:
  # registry definition
  default:
    address: "https://docker-dev.ops.tune.com"
    verify: false
```

2.1.7 Registry Auth Properties

These are properties required for authentication with a registry. Currently basic and registry_rubber auth are support. Dynamic auth uses Registry Rubber to support nonce like basic auth credentials. Please refer to Registry Rubber documentation for a deeper understanding of the service.

Name	Required	Type	Description
address	True	string	Address of docker host, must provide http scheme. Example: https://your_dev_box.office.priv:2376
ssl_cert_path	False	string	Full system path to client certs. Example: /etc/docker/certs/client/dev/
verify	False	bool	Validate certificate authority?
type	False	string	Type of auth. Currently supports basic and registry_rubber. Will default to basic.

```yaml
---
registries:
  # registry definition
  default:
    address: "https://example-docker-registry.com"
    ssl_cert_path: /path/to/certs
    verify: false

    # optional: required if using registry rubber.
    auth:
      type: "registry_rubber"
      address: "https://o-regrubber-seal.ops.tune.com"
      ssl_cert_path: "/etc/docker/certs/registry/build"
      verify: false
```

2.1.8 Environments properties

The Environments object is a grouping of instructions and configuration values that define the behavior for a CI/CD pipeline based on environment and data center. The environments and data centers are both user defined.

Warning: If using CIA: The environments and data centers need to match what is defined in CIA. Freight Forwarder will pass these values to the injector to obtain the correct configuration data.

Name	Required	Type	Description
environment	True	object	Refer to *Environment Properties* valid environments are ci, dev, development, test, testing, perf, performance, stage, staging, integration, prod, production.
service	False	object	Refer to *Service Properties*
host	False	object	Refer to *Hosts Properties*

```yaml
---
# environments object definition
environments:
  # define a host as a variable to use later
  boot2docker: &boot2docker
    - address: "https://192.168.99.100:2376"
      ssl_cert_path: /path/to/certs
      verify: false

  # override api service APP_ENV environment variable.
  api:
    env_vars:
      - "APP_ENV=development"

  # define development environment
  development:

    # define local datacenter for development
    local:
      hosts:
        default: *boot2docker

  # define staging environment
  staging:

    # define staging datacenter in sea1
    sea1: {}

  # define production environment
  production:

    # define us-east-01 for production datacenter.
    us-east-01: {}
```

2.1.9 Environment Properties

The environment of the application. An application can and one or many environments. Valid environments are ci, dev, development, test, testing, perf, performance, stage, staging, integration, prod, production.

Name	Required	Type	Description
hosts	False	object	Refer to *Hosts Properties* if not defined freight forwarder will use the docker environment variables.
data centers	True	object	Refer to *Data Center Properties*
services	False	object	Refer to *Service Properties*

```
---
environments:

  # define development environment
  development:

    # define local datacenter.
    local:

      # define development local hosts.
      hosts:

        # define default hosts
        default:
          - address: "https://192.168.99.100:2376"
            ssl_cert_path: /path/to/certs
            verify: false

      # override api service APP_ENV environment variable for development local.
      api:
        env_vars:
          - "APP_ENV=development"

  # define production environment
  production:

    # define production hosts
    hosts:
      # define hosts specificly for the api service.
      api:
        - address: "https://192.168.99.102:2376"
          ssl_cert_path: /path/to/certs
          verify: false

      # define default hosts
      default:
        - address: "https://192.168.99.101:2376"
          ssl_cert_path: /path/to/certs
          verify: false

    # override api service APP_ENV environment variable for production.
    api:
      env_vars:
        - "APP_ENV=production"
```

2.1.10 Data Center Properties

Each environment can have multiple data center objects. Some examples of data centers: local, sea1, use-east-01, and us-west-02

Name	Required	Type	Description
hosts	False	object	Refer to *Hosts Properties* if not defined freight forwarder will use the docker environment variables.
service	False	object	Refer to *Service Properties*
deploy	one of	object	Refer to *Deploy Properties*
export	one of	object	Refer to *Export Properties*
quality_control	one of	object	Refer to *Quality Control Properties*

```
---
environments:

  # define development environment.
  development:

    # define local datacenter.
    local:

      # define hosts for development local.
      hosts:

        # define default hosts.
        default:
          - address: "https://192.168.99.100:2376"
```

```
            sslCertPath: "/Users/alexb/.docker/machine/machines/ff01-dev"
            verify: false

         # define host to use during export
         export:
           - address: "https://your-ci-server.sea1.office.priv:2376"
             sslCertPath: "/path/to/your/certs/"
             verify: false

      # define deploy command orderides
      deploy:

        # override ui service properties.
        ui:
          image: registry_alias/itops/cia-ui:local-development-latest
          volumes:
            - /var/tune/cia-ui/public/

        # override static-assets service properties.
        static-assets:
          image: registry_alias/itops/cia-static-assets:local-development-latest
          volumes:
            -  /static/
          volumes_from: []

      export:
        ui:
          export_to: registry_alias

        static-assets:
          export_to: registry_alias
```

2.1.11 Deploy Properties

The deploy object allows development teams to define unique deployment behavior for specific service, environment, and data center.

Name	Required	Type	Description
service	True	object	Refer to *Service Properties*

```
---
registries:

  registry_alias: &registry_alias
    address: https://docker-registry-example.com
    ssl_cert_path: /path/to/certs
    verify: false

  default: *registry_alias

environments:
  production:
    # define datacenter
    us-west-02:

```

```yaml
      # define deploy action
      deploy:

        # deployment overrides for static-assets
        static-assets:
          image: registry_alias/itops/cia-static-assets:latest
          volumes:
            - /static/
          volumes_from: []
          restart_policy: null

        # deployment overrides for api
        api:
          image: registry_alias/itops/cia-api:o-ciapi03-2b-production-latest

        # deployment overrides for nginx
        nginx:
          image: registry_alias/itops/cia-nginx:us-west-02-production-latest
```

2.1.12 Export Properties

The export object allows development teams to define unique artifact creation behavior for a specific service, environment, and data center. Export is the only action that allows you to have a specific unique hosts definition (this is a good place for a jenkins or build host).

Note: To remove Freight Forwarders tagging scheme pass –no-tagging-scheme to the cli export command.

Warning: When exporting images Freight Forwarder will use the service definition in deploy for any dependencies/dependents. In addition, if a command is provided in the config for the service being exported Freight Forwarder assumes any changes made should be committed into the image.

Name	Required	Type	Description
service	True	object	Refer to *Service Properties*
tags	False	array[string]	A list of tags that should be applied to the image before exporting.

```yaml
---
# environments
environments:
  production:
  # datacenter definition
    us-west-02:
      # hosts for us-west-02
      hosts:

        # default hosts
        default:
          - address: "https://dev_hostname:2376"
            ssl_cert_path: /path/to/certs
            verify: false

        # host specific to the export action. will default to hosts defined in
        # default if not provided.
        export:
          - address: "https://127.0.0.1:2376"
            ssl_cert_path: /path/to/certs
            verify: false

      # overrides for the export action.
      export:

        # api service export specific overrides.
        api:
          env_vars:
            - APP_ENV=production

          # specify what registry to export to.
          export_to: registry_alias
```

2.1.13 Quality Control Properties

The quality control object allows developers a way to test containers, images, and workflows locally before deploying or exporting.

Name	Required	Type	Description
service	True	object	Refer to *Service Properties*

```yaml
---
# quality control action.
quality_control:

  # couchdb service overrides.
  couchdb:
    log_config:
      type: json-file
      config: {}

    ports:
      - "6984:6984"

  # api service overrides.
  api:
    links: []
    env_vars:
      - APP_ENV=development
```

2.1.14 Hosts Properties

The hosts object is a collection of docker hosts in which Freight Forwarder will interact with when deploying, exporting, or testing. Each service can have a collection of its own hosts but will default to the defaults definition or the standard Docker environment variables: DOCKER_HOST, DOCKER_TLS_VERIFY, DOCKER_CERT_PATH.

Name	Required	Type	Description
service_name (alias)	one of	list[*Host Properties*]	List of *Host Properties*
export	one of	list[*Host Properties*]	List with as single element of *Host Properties*
default	one of	list[*Host Properties*]	List of *Host Properties*

```yaml
---
# development environment definition.
development:
  # development environment local datacenter definition.
  local:

    # hosts definition.
    hosts:

      # default hosts.
      default:
        - address: "https://192.168.99.100:2376"
          ssl_cert_path: /path/to/certs
          verify: false
        - address: "https://192.168.99.110:2376"
          ssl_cert_path: /path/to/certs
          verify: false
```

2.1.15 Host Properties

The host object is metadata pertaining to docker hosts. If using ssl certs they must be the host where Freight Forwarder is run and be able to be read by the user running the commands.

Name	Required	Type	Description
address	True	string	Address of docker host, must provide http scheme. Example: https://your_dev_box.office.priv:2376
ssl_cert_path	False	string	Full system path to client certs. Example: /etc/docker/certs/client/dev/
verify	False	bool	Validate certificate authority?

```yaml
---
# development environment definition.
development:
  # development environment local datacenter definition.
  local:

    # hosts definition.
    hosts:

      # default hosts.
      default:
        - address: "https://192.168.99.100:2376"
          ssl_cert_path: /path/to/certs
          verify: false

      # host to build and export from.
      export:
        - address: "https://192.168.99.120:2376"
          ssl_cert_path: /path/to/certs
          verify: false

      # specific hosts for the api service.
      api:
        - address: "https://192.168.99.110:2376"
          ssl_cert_path: /path/to/certs
          verify: false
```

2.1.16 Host Config Properties

Host configuration properties can be included as a part of the the service definition. This allows for greater control when configuring a container for specific requirements to operate. It is suggested that a root level definition of a service be minimalistic compared to how it should be deployed in a specific environment or data-center.

Refer to Docker Docs for the full list of of potential properties.

Name	Required	Type	Default Value	Description
binds	False	list	['/dev/log:/dev/log:rw']	Default value applied to all containers. This allows for inherit use of */dev/log* for logging by the container
cap_add	False	string	None	Defined system capabilities to add to the container from the host. Refer to http://linux.die.net/man/7/capabi for a full list of capabilities
cap_drop	False	string	None	Defined system capabilities to remove from the container from the host. Refer to http://linux.die.net/man/7/capabi for a full list of capabilities
devices	False	list	None	Device to add to the container from the host Format of devices should match as shown below. Permissions need to be set appropriately. "/path/to/dev:/path/inside/contai
links	False	list	[]	Add link to another container
lxc_conf	False	list	[]	Add custom lxc options
readonly_root_fs	False	boolean	False	Read-only root filesystem
readonly_rootfs				
security_opt	False	list	None	

2.1.17 Container Config Properties

Container config properties are container configuration settings that can be changed by the developer to meet the container run time requirements. These properties can be set at any level but the furthest in the object chain will take presidents. Please refer to Docker Docs for a full list of properties.

Name	Required	Type	Default Value	Description
attach_stderr	False	boolean	False	Attach to stderr
attach_stdin	False	boolean	False	Attach to stdin and pass input into Container
attach_stdout	False	boolean	False	Attach to stdout
cmd	False	list	None	Override the command directive on the container
command				
domain_name	False	string	‘’	Domain name for the container
domainname				
entry_point	False	list	‘’	Defined entrypoint for the container
entrypoint				
env	False	list	‘’	Defined environment variables for the container
env_vars				
exposed_ports	False	list	‘’	Exposes port from the container. This allows a container without an ‘EXPOSE’ directive to make it available to the host
hostname	False	string	‘’	hostname of the container
image	False	string	‘’	defined image for the container
labels	False	dict|none	{}	labels to be appended to the container

2.2 CLI

2.2.1 Overview

Freight Forwarder CLI consumes the SDK and makes requests to a docker registry api and the docker client api. The CLI must be run in the same location as the configuration file (freight-forwarder.yml). Additional information regarding the configuration files can be found *Config documentation*.

For full usage information:

```
freight-forwarder --help
```

Note: Example Service Definition

```
api:
    build: "./"
    test: "./tests/"
    ports:
        - "8000:8000"
    links:
        - db
```

2.2.2 Info

class `freight_forwarder.cli.info.`**`InfoCommand`**(*args*)

Display metadata about Freight Forwarder and Python environment.

Options

- `-h, --help` (info) - Show the help message.

Example:

```
$ freight-forwarder info
Freight Forwarder: 1.0.0
docker-py: 1.3.1
Docker Api: 1.19
CPython version: 2.7.10
elapsed: 0 seconds
```

Returns exit_code

Return type int

2.2.3 Deploy

class `freight_forwarder.cli.deploy.`**`DeployCommand`**(*args*)

The deploy command pulls an image from a Docker registry, stops the previous running containers, creates and starts new containers, and cleans up the old containers and images on a docker host. If the new container fails to start, the previous container is restarted and the most recently created containers and image are removed.

Options

- `-h, --help` (info) - Show the help message

- `--data-center` (**required**) - The data center to deploy. example: sea1, sea3, or us-east-1
- `--environment` (**required**) - The environment to deploy. example: development, test, or production
- `--service` (**required**) - The Service that will be built and exported.
- `--tag` (optional) - The tag of a specific image to pull from a registry. example: sea3-development-latest
- `-e, --env` (optional) - list of environment variables to create on the container will override existing. example: MYSQL_HOST=172.17.0.4

Returns exit_code

Return type integer

2.2.4 Export

class `freight_forwarder.cli.export.`**`ExportCommand`**(*args*)

The export command builds a "service" Docker image and pushes the image to a Docker registry. A service is a defined in the configuration file.

The export command requires a Registry to be defined in the configuration file or it will default to Docker Hub, private registries are supported.

The export command by default will build the container and its defined dependencies. This will start the targeted service container after it's dependencies have been satisfied. If the container is successfully started it will push the image to the repository.

If test is set to true a test Dockerfile is required and should be defined in the configuration file. The test Dockerfile will be built and ran after the "service" Dockerfile. If the test Dockerfile fails the application will exit 1 without pushing the image to the registry.

The configs flag requires integration with CIA. For more information about CIA please to the documentation.

When the export command is executed with `--no-validation` it will perform the following actions.

1. Build the defined Dockerfile or pull the image for the service.
2. Inject a configuration if defined with credentials.
3. Push the Image to the defined destination registry or the defined default, if no default is defined, it will attempt to push the image to Docker Hub.

To implement with a Continuous Integration solution (i.e. Jenkins, Travis, etc); please refer to below and use the `-y` option to not prompt for confirmation.

Options

- `-h, --help` (info) - Show the help message.
- `--data-center` (**required**) - The data center to deploy. example: us-east-02, dal3, or us-east-01.
- `--environment` (**required**) - The environment to deploy. example: development, test, or production.
- `--service` (**required**) - The Service that will be built and exported.
- `--clean` (optional) - Clean up anything that was created during current command execution.

- `--configs` (optional) - Inject configuration files. Requires CIA integration.
- `--tag` (optional) - Metadata to tag Docker images with.
- `--no-tagging-scheme` (optional) - Turn off freight forwarders tagging scheme.
- `--test` (optional) - Build and run test Dockerfile for validation before pushing image.
- `--use-cache` (optional) - Allows use of cache when building images defaults to false.
- `--no-validation` (optional) - The image will be built, NOT started and pushed to the registry.
- `-y` (optional) - Disables the interactive confirmation with `--no-validation`.

Returns exit_code

Return type integer

2.2.5 Offload

class `freight_forwarder.cli.offload.`**`OffloadCommand`**(*args*)

The offload Command removes all containers and images related to the service provided.

Options

- `-h, --help` (info) - Show the help message.
- `--data-center` (**required**) - The data center to deploy. example: sea1, sea3, or us-east-1
- `--environment` (**required**) - The environment to deploy. example: development, test, or production
- `--service` (**required**) - This service in which all containers and images will be removed.

Returns exit_code

Return type integer

2.2.6 Quality Control

class `freight_forwarder.cli.quality_control.`**`QualityControlCommand`**(*args*)

The quality-control command allows development teams to validate freight forwarder work flows without actually deploying or exporting.

Options

- `-h, --help` (info) - Show the help message
- `--data-center` (**required**) - The data center to deploy. example: sea1, sea3, or us-east-1.
- `--environment` (**required**) - The environment to deploy. example: development, test, or production.
- `--service` (**required**) - The service that will be used for testing.
- `--attach` (optional) - Attach to the service containers output.
- `--clean` (optional) - Remove all images and containers after run.
- `-e, --env` (optional) - list of environment variables to create on the container will override existing. example: MYSQL_HOST=172.17.0.4

- `--configs` (optional) - Inject configuration files. Requires CIA integration.
- `--test` (optional) - Run test Dockerfile must be provided in the configuration file.
- `--use-cache` (optional) - Allows use of cache when building images defaults to false.

Returns exit_code

Return type integer

2.2.7 Test

class `freight_forwarder.cli.test.`**`TestCommand`**(*args*)

The test command allows developers to build and run their test docker file without interfering with their current running application containers. This command is designed to be ran periodically throughout a developers normal development cycle. Its a nice encapsulated way to run a projects test suite.

Warning: This command requires your service definition to have a test Dockerfile.

Options

- `-h, --help` (info) - Show the help message
- `--data-center` (**required**) - The data center to deploy. example: sea1, sea3, or us-east-1
- `--environment` (**required**) - The environment to deploy. example: development, test, or production
- `--service` (**required**) - The service that will be used for testing.
- `--configs` (optional) - Inject configuration files. Requires CIA integration.

Returns exit_code

Return type integer

2.2.8 Marshalling Yard

class `freight_forwarder.cli.marshaling_yard.`**`MarshalingYardCommand`**(*args*)

MarshalingYard interacts with a docker registry and provides information concerning the images and tags.

- `--alias` (optional) - The registry alias defined in freight-forwarder config file. defaults: 'default'.

One of the options is required

- `search` Searches defined registry with keyword
- `tags` Returns tag associated with value provided from the specified registry

Returns exit_code

Return type integer

Configuration File Configuration file for projects.

CLI CLI command index.

CHAPTER 3

Extending Freight Forwarder

3.1 Injector

3.1.1 Overview

The injector was built and designed to create configuration files and share them with freight forwarder during the CI process. We use the injector to make api calls to CIA an internal tool that we use to manage configuration files. The injector uses CIA's response and writes configuration files to disk, shares them with freight forwarder using a shared volume, and returns *Injector Response* to provide metadata about the configuration files. This doesn't have to be limited to configuration files and can be extended by creating a new injector container so long as it follows a few rules.

Note: If injection is required during export set –configs=true. This will be changed to –inject in the future.

3.1.2 Workflow

- Freight Forwarder pulls the injector image defined in environment variable `INJECTOR_IMAGE`. The value must be in the following format `repository/namespace:tag`.
- Freight Forwarder passes *Environment Variables* to the injector container when its created.
- Freight Forwarder then runs the injector.
- Freight Forwarder uses the data returned from the injector to create intermediate containers based on the application image.
- Freight Forwarder than commits the changes to the application image.

3.1.3 Creating Injector Images

When creating an injector image the container created from the image is required to produce something to inject into the the application image. Freight Forwarder provides *Environment Variables* to the injector container as a way to identify what resources it should create. After the injector creates the required resources it must return a valid *Injector Response*. Freight Forwarder will then use that response to commit the required resources into the application image.

After the injector image has been created and tested the end user will need to provide the `INJECTOR_IMAGE` environment variable with a string value in the following format: `repository/namespace:tag`. In addition, to the environment variable the end user will have to set –configs=true. This will tell Freight Forwarder to use the provided image to add a layer to the application image after it has been built or pulled from a docker registry. A specific registry can be defined in the configuration file with the alias of "injector". If the injector alias isn't defined the default registry will be used.

3.1.4 Environment Variables

These environment variables will be passed to the injector container every run. They will change based on the Freight Forwarder configuration file, user provided environment variables, and command line options.

Name	Required	Type	Description
INJECTOR_CLIENT_ID	False	string	OAuth client id, this must be provided by the user.
INJECTOR_CLIENT_SECRET	False	string	OAuth secret id, this must be provided by the user.
ENVIRONMENT	True	string	Current environment being worked on. example: development This maps to what is being passed to –environment option.
DATACENTER	True	string	Current data center being worked on. example: us-west-02 This maps to what is being passed to –data-center option.
PROJECT	True	string	Current project being worked on. example: itops This maps to what is in the users configuration file.
SERVICE	True	string	Current service being worked on. example: app This maps to what is being passed to –service option.

3.1.5 Injector Response

The injector container must return a list of objects each with the following properties formatted in json. This metadata will be used to copy files and configure them correctly for the application image.

Name	Required	Type	Description
name	True	string	Name of the file being written.
path	True	string	Application container Path in which to write file.
config_path	True	string	Path to find file inside of the injector container.
user	True	string	File belongs to this user, user must already exist.
group	True	string	File belongs to the this group, group must already exist.
chmod	True	int or string	File permissions.
checksum	True	string	MD5 sum of file.
notifications	False	object	Refer to *Notifications Object*

```json
[
  {
    "name": "myconf.json",
    "path": "/opt/docker-example/conf",
    "config_path": "/configs/myconf.json",
    "user": "root",
    "group": "root",
    "chmod": 755,
    "checksum": "5cdfd05adb519372bd908eb5aaa1a203",
    "notifications": {
      "info": [
        {
          "type": "configs",
          "details": "Template has not changed. Returning previous config."
        }
      ]
    }
  }
]
```

3.1.6 Notifications Object

The notifications object allows the injector to pass a message to the user or raise an exception if it fails to complete a task. If an error is provided in the notifications object freight forwarder will raise the error, this will result in a failed run.

Name	Required	Type	Description
info	False	list	Send a message to the user informing them of something. list of *Message Object*
warnings	False	list	Warn the user about a potential issue. list of *Message Object*
errors	False	list	Raise an error and terminate current freight forwarder run. list of *Message Object*

```
{
  "notifications": {
    "info": [],
    "warnings": [],
    "errors": []
  }
}
```

Warning: If an errors notification is provided freight forwarder will terminate the current run.

3.1.7 Message Object

Name	Required	Type	Description
type	True	string	Type of message.
details	True	string	The message to deploy to the end user.

```
{
  "type": "configs",
  "details": "Template has not changed. Returning previous config."
}
```

3.2 SDK

3.2.1 Overview

Coming Soon!

Injector Describes how to implement an injector.

SDK SDK Documentation.

3.3 Contributing

3.3.1 Development Environment

Docker is required follow these install instructions.

OSX:

```
# install home brew this will also install Xcode command line tool.  Follow all instructions given du
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

# update brew
$ brew update

# install pyenv
$ brew install pyenv

# You may need to manually set PYENV_ROOT, open a new terminal and see if it
# was set by the install proces:

$ echo $PYENV_ROOT

## Setting Manually
#
# If your PYENV_ROOT isn't set, you can use either $HOME/.pyenv or the
# homebrew pyenv directory, /usr/local/opt/pyenv.  Put
```

```
#
# export PYENV_ROOT=/usr/local/opt/pyenv
#
# -or-
#
# export PYENV_ROOT="$HOME"/.pyenv
#
# in your .bashrc or .bash_profile, or whatever your appropriate dotfile is.
#
# ---
#
## Setting with oh-my-zsh
#
# You can just use the pyenv plugin.  Open your .zshrc and make sure that
# this line:
#   plugins=(git rvm osx pyenv)
# contains pyenv.  Yours may have more or fewer plugins.
#
# If you just activated the pyenv plugin, you need to open a new shell to
# make sure it loads.

# install libyaml
$ brew install libyaml

# install a few plugins for pyenv
$ mkdir -p $PYENV_ROOT/plugins
$ git clone "git://github.com/yyuu/pyenv-pip-rehash.git" "${PYENV_ROOT}/plugins/pyenv-pip-rehash"
$ git clone "git://github.com/yyuu/pyenv-virtualenv.git" "${PYENV_ROOT}/plugins/pyenv-virtualenv"
$ git clone "git://github.com/yyuu/pyenv-which-ext.git"  "${PYENV_ROOT}/plugins/pyenv-which-ext"

##  Load pyenv-virtualenv when shells are created:
#
# To make sure that both of your pugins are loading, these lines should be
# in one of your dotfiles.
#
#     eval "$(pyenv init -)"
#     eval "$(pyenv virtualenv-init -)"

# Now that it will load automatically, activate the plugin for this shell:
$ eval "$(pyenv virtualenv-init -)"

# install a specific version
$ pyenv install 2.7.10

# create a virtual env
$ pyenv virtualenv 2.7.10 freight-forwarder

# list all of your virtual environments
$ pyenv virtualenvs

# activate your environment
$ pyenv activate freight-forwarder

# clone repo
$ git clone git@github.com:Adapp/freight_forwarder.git

# install requirements
$ pip install -r requirements.txt
```

3.3.2 Style Guidelines

Coming soon!

3.3.3 Release Steps

- version++; The verson can be found freight_forwarder/const.py
- Update change log.
- Git tag the version
- $ python ./setup.py bdist_wheel
- Upload to pypi.

3.3.4 Build Documentation

Docker:

```
$ pip install freight-forwarder
```

The freight-forwarder.yml file will have to be updated with your specific docker host info:

```
environments:
  alexb: &alexb
    address: "https://172.16.135.137:2376" # <- Change me
    ssl_cert_path: "/Users/alexb/.docker/machine/machines/ff02-dev" # <- Change me
    verify: false

  development:
    local:
      hosts:
        default:
          - *alexb
```

Then just run Freight to start the documentation containers:

```
$ freight-forwarder quality-control --environment development --data-center local --service docs
```

After the containers start you can find the documentation at: your_container_ip:8080

Make:

```
$ cd docs/
$ pip install -r requirements.txt
$ make html
```

The html can found here: ./docs/_build/

3.4 FAQ

3.4.1 Can I use any project/team name with Freight Forwarder?

Yes. Just set it in the manifest/CLI and the image will be tagged and stored appropriately.

3.4.2 How do I find out where the keys to my various 'containerShips' are?

SSL certs are required for connecting to Docker daemons. Even if you're running Docker locally, you'll need to enable SSL support to use that daemon.