Cekit Documentation

Release 2.2.7

Marek Goldmann

Jul 30, 2019

Contents

CHAPTER 1

About

Container image creation tool. Cekit was previously known as Concreate. If your migrating from concreate tool, please follow upgrade instructions http://cekit.readthedocs.io/en/develop/installation.html#installing-cekit.

Cekit helps to build container images from image definition files with strong focus on modularity and code reuse.

CHAPTER 2

Features

- Building container images from YAML image definitions
- Integration/unit testing of images
- Releasing container images by building it in Red Hat supported build system

CHAPTER 3

Installation

If you are running Fedora, you can install Cekit easily via:

```
dnf copr enable @cekit/cekit
dnf install python3-cekit
```

For other platforms, please refer to documentation.

CHAPTER 4

Usage

First steps tutorial is under construction, for now please refer to the `cekit --help` output.

CHAPTER 5

Documentation

Documentation is available here.

5.1 Operation Guide

This chapter will guide you through all the Cekit basic usage. After reading you should have Cekit installed and be familiar with building and testing images with it.

5.1.1 Installation

This chapter will guide you through all the steps needed to setup Cekit on your operating system.

Contents

Installing Cekit

We provide RPM packages for Fedora, CentOS, RHEL distribution. Cekit installation on other platforms is still possible via *pip*

Fedora / CentOS / RHEL

On RHEL derivatives we strongly suggest installing Cekit using the YUM/DNF package manager. We provide a COPR repository for Cekit which contains everything needed to install Cekit.

Fedora

Supported versions: 27, 28.

For Fedora we provide custom Copr repository. To enable the repository and install Cekit on your system please run:

```
yum install yum-plugin-copr
yum copr enable @cekit/cekit
yum install python2-cekit
```

RHEL

Supported versions: 7.x

For RHEL we provide custom Copr repository. To enable the repository and install Cekit on your system please run:

```
curl https://copr.fedorainfracloud.org/coprs/g/cekit/cekit/repo/epel-7/group_cekit-
↪cekit-epel-7.repo -o /etc/yum.repos.d/cekit-epel-7.repo
yum install python2-cekit
```

Other systems

We strongly advise to use Virtualenv to install Cekit. Please consult your package manager for the correct package name.

To create custom Python virtual environment please run following commands on your system:

```
# Prepare virtual environment
virtualenv ~/cekit
source ~/cekit/bin/activate

# Install Cekit
# Execute the same command to upgrade to latest version
pip install -U cekit

# Now you are able to run Cekit
cekit --help
```

Note: Every time you want to use Cekit you must activate Cekit Python virtual environment by executing *source ~/cekit/bin/activate*

Build

To build container images you need one of the following:

- docker
- buildah

Test

For running tests you need:

- docker
- docker python bindings
- behave
- python-lxml

Upgrading

Fedora / CentOS / RHEL

On this platform you should be using RPM and our COPR repository for Cekit

Note: We assume, that you have this repository enabled on your system

Fedora

Supported versions: 25, 26, 27.

```
dnf update python3-cekit
```

CentOS & RHEL

Supported versions: 7.

```
yum update python2-cekit
```

Other systems

We suggest using pip and Virtualenv to host you Cekit.

```
# Activate virtual environment
source ~/cekit/bin/activate

pip install -U cekit
```

Upgrading from Concreate

Cekit and Concreate are the very same tool. Concreate was rename to Cekit in 2.0 release.

Fedora / CentOS / RHEL

You should be using RPM and yum/dnf to manage Cekit/Concreate installation here.

Fedora

Supported versions: 25, 26, 27.

```
dnf remove python3-concreate
dnf copr remove goldmann/concreate

dnf copr enable @cekit/cekit
dnf install python3-cekit
```

CentOS

Supported versions: 7.

```
yum remove python2-concreate
rm -rf  /etc/yum.repos.d/_copr_goldmann-concreate.repo

yum copr enable @cekit/cekit
yum install python2-cekit
```

RHEL

Supported versions: 7.

```
yum remove python2-concreate
rm -rf /etc/yum.repos.d/goldmann-concreate-epel-7.repo

curl https://copr.fedorainfracloud.org/coprs/g/cekit/cekit/repo/epel-7/group_cekit-
↪cekit-epel-7.repo -o /etc/yum.repos.d/cekit-epel-7.repo
yum install python2-cekit
```

Other systems

We strongly advise to use Virtualenv to install Cekit. Please consult your package manager of choice for the correct package name.

```
# Activate virtual environment
source ~/cekit/bin/activate

pip uninstall concreate
pip install -U cekit
```

Dotfile migration

Concreate used *~/.concreate.d* and *~/.concreate* dot files to held its configuration. This was changed with Cekit. Cekit uses only *~/.cekit* directory to host all its configuration files.

To migrate your configuration please run:

```
mv ~/.concreate.d ~/.cekit
mv ~/.concreate ~/.cekit/config
```

5.1.2 Building image

Cekit supports following builder engines:

- `Docker` – build the container image using docker build command and it default option
- `OSBS` – build the container image using OSBS service
- `Buildah` – build the container image using Buildah

Executing builds

You can execute an container image build by running:

```
$ cekit build
```

Options affecting builder:

- `--tag` – an image tag used to build image (can be specified multiple times)
- `--redhat` – build image using Red Hat defaults. See *Configuration section for Red Hat specific options* for additional details.
- *–add-help'* – add generated *help.md* file to the image
- *–no-add-help'* – don't add generated *help.md* file to the image

- `--work-dir` – sets Cekit works directory where dist_git repositories are cloned into See *Configuration section for work_dir*
- `--package-manager` – allows selecting between different package managers such as `yum` or `microdnf`. Defaults to `yum`
- `--build-engine` – a builder engine to use `osbs`, `buildah` or `docker`[1]
- `--build-pull` – ask a builder engine to check and fetch latest base image
- `--build-osbs-stage` – use `rhpkg-stage` tool instead of `rhpkg`
- `--build-osbs-release`[2] – perform a OSBS release build
- `--build-osbs-user` – alternative user passed to *rhpkg –user*
- `--build-osbs-target` – overrides the default `rhpkg` target
- `--build-osbs-commit-msg` – custom commit message for dist-git
- `--build-osbs-nowait` – run *rhpkg container-build* with *–nowait* option specified
- `--build-tech-preview`[2] – updates image descriptor `name` key to contain `-tech-preview` suffix in family part of the image name

  Example: If your `name` in image descriptor is: `jboss-eap-7/eap7`, generated name will be: `jboss-eap-7-tech-preview/eap7`.

Docker build

This is the default way to build an container image. The image is build using `docker build`.

Example: Building a docker image

```
$ cekit build
```

OSBS build

This build engine is using `rhpkg container-build` to build the image using OSBS service. By default it performs scratch build. If you need a release build you need to specify `--build-osbs-release` parameter.

Example: Performing scratch build

```
$ cekit build --build-engine=osbs
```

Example: Performing release build

```
$ cekit build --build-engine=osbs --build-osbs-release
```

Buildah build

This build engine is based on Buildah. Buildah still doesn't support non-privileged builds so you need to have **sudo** configured to run *buildah* as a root user on your desktop.

[1] docker build engine is default
[2] option is valid on for `osbs` build engine

Note: If you need to use any non default registry, please update */etc/containers/registry.conf* file.

Example: Building image using Buildah

```
$ cekit build --build-engine=buildah
```

5.1.3 Artifact Caching

Cekit is automatically caching all artifacts used to build the image. This means that if your image descriptor contains following artifact:

```
artifacts:
      # File will be downloaded and verified.
    - name: jolokia-1.3.6-bin.tar.gz
      url: https://github.com/rhuss/jolokia/releases/download/v1.3.6/jolokia-1.3.6-
↪bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
```

It will be automatically cached into `~/.cekit/cache/` directory during image build. This is useful as the artifact will be automatically copied from cache instead of downloading it again on any rebuild.

Note: Artifacts in cache are discovered by a hash value. So even if you define same artifact by different name it will be discovered in cache and copied into your image. This also means that Cekit is using cache only for artifacts which define at least one hash.

Managing Cache

Cekit contains command line tool called `cekit-cache` which is used to manage its cache.

Options affecting cekit-cache:

- `--verbose` – setups verbose output
- `--work-dir` – sets Cekit works directory where cache directory is located. See *Configuration section for work_dir*
- `--version` – prints Cekit version

Note: All cache related files are places in your `--work-dir` inside `cache` subdirectory. This is `~/.cekit/cache` by default. This means that cache is realted to your `--work-dir` and switching your `--work-dir` will use different artifact cache.

Caching an artifact manually

Cekit supports caching an artifact manually. This is very use full if you need to introduce non-public artifact to a Cekit. To cache an artifact you need to specify path to the artifact on filesystem or its URL and one of the supported hashes (md5, sha256, sha512).

Example: Specifying an artifact via path:

```
$ cekit-cache add path/to/file --md5 checksum
```

Example: Specifying an artifact via url:

```
$ cekit-cache add https://foo.bar/baz --sha256 checksum
```

Options affecting cekit-cache add:

- `--md5` – contains md5 hash of an artifact
- `--sha256` – contains sha256 hash of an artifact
- `--sha512` – contains sha512 hash of an artifact

Listing cached artifacts

To list all artifact known to a Cekit cache you need to run following command:

```
$ cekit-cache ls
```

After running the command you can see following output: .. code:

```
Cached artifacts:
912c3cc4-7bd3-445d-9927-5063ba3b3bc1:
  sha256: 04b95a87ee88e1cba7682884ea7f89d5ec097c0fa513e7aca1366d79fb3290a8
  sha1: 9cbe5393b6837849edbc067fe1a1405ff0c43605
  md5: f97f623e5b614a7b6d1eb5ff7158027b
  names:
    hawkular-javaagent-1.0.1.Final-redhat-2-shaded.jar
d9171217-744e-43af-8d2f-5ee04f2fd741:
  sha256: 223d394c3912028ddd18c6401b3aa97fe80e8d0ae3646df2036d856f35f18735
  sha1: 7c32933edaea4ba40bdcc171e25a0a9c36e2de20
  md5: d31c6b1525e6d2d24062ef26a9f639a8
  names:
   jolokia-jvm-1.5.0.redhat-1-agent.jar
```

As you can see, we've got listing of two artifacts and they're represented by uuid. One is **912c3cc4-7bd3-445d-9927-5063ba3b3bc1** which is `hawkular-javaagent-1.0.1.Final-redhat-2-shaded.jar`. Second one is **d9171217-744e-43af-8d2f-5ee04f2fd741** which is `jolokia-jvm-1.5.0.redhat-1-agent.jar`. The artifacts uuids are auto generated when artifact is cached and serves as an unique id of an artifact.

Note: Artifact uuid is also used as a filename for an artifact, you can see them in your `~/.cekit/cache` directory.

Removing cached artifact

If you are not interested in particular artifact being at your cache you can delete it by executing following command:

```
$ cekit-cache rm uuid
```

Note: You can get uuid of any artifact by invoking `cekit-cache ls` command. Please consult *Listing cached artifacts*

Wiping whole cache

To wipe whole artifact cache you need to manually remove `cache` subdirectory inside your `--work-dir`.

Example: To remove your cache located in `~/.cekit/cache` directory run:

```
$ rm -rf ~/.cekit/cache
```

5.1.4 Overrides

During an image life cycle there can be a need to do a slightly tweaked builds - using different base images, injecting newer libraries etc. We want to support such scenarios without a need of duplicating whole image sources. To achieve this Cekit supports overrides mechanism for its image descriptor. You can override almost anything in image descriptor. The overrides are based on overrides descriptor - a YAML object containing overrides for the image descriptor.

To use an override descriptor you need to pass `--overrides-file` argument to a Cekit. You can also pass JSON/YAML object representing changes directly via `--overrides` command line argument.

Example: To use overrides.yaml file located in current working directory run:

```
$ cekit build --overrides-file overrides.yaml
```

Example: To override a label via command line run:

```
$ cekit build --overrides "{'labels': [{'name': 'foo', 'value': 'overridden'}]}"
```

Overrides Chaining

You can even specify multiple overrides. Overrides are resolved that last specified is the most important one. This means that values from *last override specified overrides all values from former ones*.

Example: If you run following command, label *foo* will be set to *baz*.

```
$ cekit build --overrides "{'labels': [{'name': 'foo', 'value': 'bar'}]} --overrides "
↪{'labels': [{'name': 'foo', 'value': 'baz'}]}"
```

How overrides works

Cekit is using YAML format for its descriptors. Overrides in cekit works on YAML node level.

Scalar nodes

Scalar nodes are easy to override, if Cekit finds any scalar node in an overrides descriptor it updates its value in image descriptor with the overridden one.

Example: Overriding scalar node:

image descriptor

```
schema_version: 1
name: "dummy/example"
version: "0.1"
from: "busybox:latest"
```

overrides descriptor

```
schema_version: 1
from: "fedora:latest"
```

overridden image descriptor

```
schema_version: 1
name: "dummy/example"
version: "0.1"
from: "fedora:latest"
```

Sequence nodes

Sequence nodes are little bit tricky, if they're representing plain arrays, we cannot easily override any value so Cekit is just replacing the whole sequence.

Example: Overriding plain array node:

image descriptor

```
schema_version: 1
name: "dummy/example"
version: "0.1"
from: "busybox:latest"
run:
  cmd:
  - "echo"
  - "foo"
```

overrides descriptor

```
schema_version: 1
run:
  cmd:
  - "bar"
```

overridden image descriptor

```
schema_version: 1
name: "dummy/example"
version: "0.1"
from: "busybox:latest"
run:
  cmd:
  - "bar"
```

Mapping nodes

Mappings are merged via *name* key. If Cekit is overriding an mapping or array of mappings it tries to find a **name** key in mapping and use and identification of mapping. If two **name** keys matches, all keys of the mapping are updated.

Example: Updating mapping node:

image descriptor

```
schema_version: 1
name: "dummy/example"
version: "0.1"
from: "busybox:latest"
envs:
- name: "FOO"
  value: "BAR"
```

overrides descriptor

```
schema_version: 1
envs:
- name: "FOO"
  value: "new value"
```

overridden image descriptor

```
schema_version: 1
name: "dummy/example"
version: "0.1"
from: "busybox:latest"
envs:
- name: "FOO"
  value: "new value"
```

Removing keys

Overriding can result into a need of removing any key from a descriptor. You can achieve this by overriding a key with a YAML null value ~.

Example: Remove value from a defined variable

If you have a variable defined in a following way:

```
envs:
  - name: foo
    value: bar
```

you can remove `value` key via following override:

```
envs:
  - name: foo
    value: ~
```

It will result into following variable definition:

```
envs:
  - name: foo
```

5.1.5 Testing images

Cekit is able to run behave based tests for images. We suggest you read the Behave documentation before reading this chapter.

An image can be tested by running:

```
$ cekit test
```

Test options

- `--test-wip` – only run tests tagged with the `@wip` tag.
- `--test-steps-url` – a git repository url containing steps for tests.
- `--tag altname` – overrides the name of the Image used for testing to `altname`. Only the first occurrence of this argument is honoured.
- `--test-name` – part of the Scenario name to be executed

About Tests

Behave tests are defined in two separate parts: steps and features.

You can place the files defining tests in a `tests` directory next to the image descriptor, module descriptor or in a root of a git repository which contains the modules.

The tests directory is structured as follows:

```
tests/features
tests/features/amq.feature
tests/steps
tests/steps/custom_steps.py
```

The `tests/features` directory is the place where you can drop your behave features.

The `tests/steps` directory is optional and contains custom steps for the specific image/module.

We strongly recommend that a test is written for every feature that is added to the image.

Cekit comes with a list of build-in steps that are available for use in tests. See the steps repository.

Where necessary we encourage people to add or extend these steps.

Tags

Cekit selects which tests to run via the *tags* mechanism. Here are several examples of ways ways that tags could be used for managing tests across a set of related images:

1. *Product tags*

   Tags based on image names. Cekit derives two test tag names from the name of the Image being tested. The whole image name is converted into one tag, and everything before the first '/' character is converted into another. **Example**: If you are testing the `jboss-eap-7/eap7` image, tests will be invoked with tags `@jboss-eap-7` and `@jboss-eap-7/eap7`.

   If `--tag` is specified, then the argument is used in place of the Image name for the process above. **Example** If you provided `--tag foo/bar`, then the tags used would be `@foo` and `@foo/bar`.

2. *Wip tags*

This is very special behavior used mainly in development. Its purpose is to to limit the tests to be run to a subset you are working on. To achieve this you should mark your in-development test scenarios with the `@wip` tag and run `cekit test --test-wip`. All other scenarios not tagged `@wip` will be ignored.

3. *The @ci tag*

   If `cekit` is not running as a user called `jenkins`, the tag `@ci` is added to the list of ignored tags, meaning any tests tagged `@ci` are ignored and not executed.

   The purpose of this behavior is to ease specifying tests that are only executed when run within Jenkins CI.

Running specific test

Cekit enables you to run specific Scenario only. To do it you need to run Cekit with `--test-name <name of the tests>` command line argument.

Example: If you have following Scenario in your feature files:

```
Scenario: Check custom debug port is available
When container is started with env
| variable   | value |
| DEBUG      | true  |
| DEBUG_PORT | 8798  |
Then check that port 8798 is open
```

Then you can instruct Cekit to run this test in a following way:

```
$ cekit test --test-name 'Check custom debug port is available'
```

Note: `--test-name` switch can be specified multiple times and only the Scenarios matching all of the names are executed.

Skipping tests

If there is a particular test which needs to be temporally disabled, you can use `@ignore` tag to disable it.

For example to disable following Scenario:

```
Scenario: Check custom debug port is available
When container is started with env
| variable   | value |
| DEBUG      | true  |
| DEBUG_PORT | 8798  |
Then check that port 8798 is open
```

You need to tag it with `@ignore` tag in a following way:

```
@ignore
Scenario: Check custom debug port is available
When container is started with env
| variable   | value |
| DEBUG      | true  |
| DEBUG_PORT | 8798  |
Then check that port 8798 is open
```

5.1.6 Developing modules locally

Cekit enables you to use a work in progress modules to build the image by exploiting its overrides system. As an example, imagine we have very simple image which is using one module from a cct_module repository like this:

```
schema_version: 1
name: "dummy/example"
version: "0.1"
from: "jboss/openjdk18-rhel7:1.1"
modules:
  repositories:
    - git:
      url: https://github.com/jboss-openshift/cct_module.git
      ref: master
  install:
    - name: s2i-common
```

Now imagine, we have found a bug in its s2i-common module. We will clone the module repository localy by executing:

1. Clone cct_module to your workstation to `~/repo/cct_module`

```
$ git clone https://github.com/jboss-openshift/cct_module.git /home/user/repo/cct_
↪module
```

2. Then we will create override.yaml next to the image.yaml, override.yaml should look like:

```
schema_version: 1
modules:
  repositories:
    - path: "/home/user/repo/cct_module"
```

3. We now can build the image using overridden module by executing:

```
$ cekit generate --overrides-file overrides.yaml
```

4. When your work is finished, commit and push your changes to a module repository and remove overrides.yaml

5.1.7 Injecting local artifacts

During module/image development there can be a need to use locally built artifact instead of a released one. The easiest way to inject such artifact is to use override mechanism.

To override an artifact imagine, that you have an artifact defined in a way:

```
- md5: d31c6b1525e6d2d24062ef26a9f639a8
  name: jolokia.jar
  url: https://maven.repository.redhat.com/ga/org/jolokia/jolokia-jvm/1.5.0.redhat-1/
↪jolokia-jvm-1.5.0.redhat-1-agent.jar
```

And you want to inject a local build of new version of our artifact. To archive it you need to create following override:

```
- name: jolokia.jar
  path: /tmp/build/jolokia.jar
```

Whenever you override artifact, all previous checksums are removed too. If you want your new artifact to pass integrity checks you need to define checksum also in overrides in a following way:

```
- md5: d31c6b1525e6d2d24062ef26a9f639a8
  name: jolokia.jar
  path: /tmp/build/joloika.jar
```

Note: If the artifacts lacks the name key, its automatically created by using basename of the artifact path or url.

5.1.8 Repository management

One of the hardest challenges we faced with Cekit is how to manage and define package repositories correctly. Our current solution works in following scenarios:

1) Building CentOS or Fedora based images
2) Building RHEL based images on subscribed hosts
3) Building RHEL based images on unsubscribed hosts

Best Practices

To achieve such behavior in Red Hat Middleware Container Images we created following rules and suggestions.

Defining repositories in container images

You should use Plain repository definition for every repository as this will work easily on Red Hat subscribed host and will assume everyone can rebuild are RHEL based images.

Example: Define Software Collections repository

```
packages:
    repositories:
        - name: SCL
          id: rhel-server-rhscl-7-rpms
```

If you have repository defined this way, Cekit will not try to inject it and will expect the repository to be already available inside your container image. If it's not provided by the image (for example repository definition already available in `/etc/yum.repos.d/` directory) or the host (for example on via subscribed RHEL host) you need to override this repository. To override a repository definition you need to specify a repository with same `name`. By overriding Plain repository type, you are actually saying that you have an external mechanism to inject the repository inside the image. This can be any supported repository type.

Note: You can view Plain repository type as an abstract classes and ODCS, RPM and URL repositories as an actual implementation.

Example: Override Software Collection repository for CentOS base

```
packages:
    repositories:
        - name: SCL
          rpm: centos-release-scl
```

Example: Override Software Collections repository with a custom yum repository file

```
packages:
    repositories:
        - name: SCL
          url:
            repository: https://foo.lan/scl.repo
            gpg: https://foo.lan/scl.gpg
```

Example: Override Software Collections repository with an ODCS

```
packages:
    repositories:
        - name: SCL
          odcs:
            pulp: rhel-server-rhscl-7-rpms
```

Note: See *Red Hat Repository* chapter which describes how Plain repositories are handled inside Red Hat Infrastructure.

5.1.9 Image Help Pages

At image build-time, Cekit generates a "help" documentation page, which is saved adjacent to the generate image sources. The help page can optionally be included into the image. The template used to generate the help page can be overridden by the user's configuration file, or the input image configuration, either via the central image.yaml file, included modules or overrides.

Adding the help page to your image

There are two ways to instruct Cekit to add the help page to your image: either Specify `--add-help` on the command-line when running the *build* phase, or via your configuration file, in the *doc* section:

```
[doc]
addhelp = true
```

Providing your own help page template

The default help template is supplied within Cekit. You can override it for every image via your configuration, or on a per-image basis in the image definition.

Via configuration

Example:

```
[doc]
help_template = /home/jon/something/my_help.md
```

Via image definition

This could be in the master `image.yaml`, or in a module referenced from the `image.yaml`, or on the command-line via `--overrides` or `--overrides-file`:

Example:

```
...
help:
  template: /home/jon/something/my_help.md
```

5.1.10 Red Hat Environment

If you are running Cekit in Red Hat internal infrastructure it behaves differently. This behavior is triggered by changing *redhat configuration option* in Cekit configuration file.

Tools

Cekit integration with following tools is changed in following ways:

- runs `rhpkg` instead of `fedpkg`
- runs `odcs` command with `--redhat` option set

Environment Variables

Following variables are added into the image:

- `JBOSS_IMAGE_NAME` - contains name of the image
- `JBOSS_IMAGE_VERSION` - contains version of the image

Labels

Following labels are added into the image:

- `name` - contains name of the image
- `version` - contains version of the image

Repositories

In Red Hat we are using ODCS/OSBS integration to access repositories for building our container images. To make our life easier for local development Cekit is able to ask ODCS to create `content_sets.yml` based repositories even for local Docker builds. This means that if you set *redhat configuration option* to True, your content_sets repositories will be injected into the image you are building and you can successfully build an image on non-subscribed hosts.

Artifacts

In Red Hat environment we are using Brew to build our packages and artifacts. Cekit provides an integration layer with Brew and enables to use artifact directly from Brew. To enable this set *redhat configuration option* to True and define artifact **only** by specifying its `md5` checksum.

Example: Following artifact will be fetched directly from brew for Docker build and uses Brew/OSBS inegration for OSBS build.

```
artifacts:
  - md5: d31c6b1525e6d2d24062ef26a9f639a8
    name: jolokia-jvm-1.5.0.redhat-1-agent.jar
```

5.2 Reference guide

This chapter provides overview of all posibble option for Cekit configuration, descriptors and command line switches.

5.2.1 Image descriptor

Image descriptor contains all information Cekit needs to build and test a container image.

Contents

Name

This key is **required**.

Image name without the registry part.

```
name: "jboss-eap-7/eap70-openshift"
```

Version

This key is **required**.

Version of the image.

```
version: "1.4"
```

Description

Short summary of the image.

Value of the `description` key is added to the image as two labels: `description` and `summary` unless such labels are already defined in the image descriptor's *Labels* section.

```
description: "Red Hat JBoss Enterprise Application 7.0 - An application platform for␣
↪hosting your apps that provides an innovative modular, cloud-ready architecture,␣
↪powerful management and automation, and world class developer productivity."
```

From

This key is **required**.

Base image of your image.

```
from: "jboss-eap-7-tech-preview/eap70:1.2"
```

help

The optional help sub-section defines a single key `template`, which can be used to define a filename to use for generating image documentation at build time. By default, a template supplied within Cekit is used.

At image build-time, the template is interpreted by the Jinja2 template engine. For a concrete example, see the default help.jinja supplied in the Cekit source code.

```
help:
  template: myhelp.jinja
```

Environment variables

Similar to labels – we can specify environment variables that should be present in the container after running the image. We provide `envs` section for this.

Environment variables can be divided into two types:

1. **Information environment variables** – these are set and available in the image. This type of environment variables provide information to the image consumer. In most cases such environment variables *should not* be modified.
2. **Configuration environment variables** – this type of variables are used to define environment variables used to configure services inside running container.

   These environment variables are **not** set during image build time but *can* be set at run time.

   Every configuration enviromnent variable should provide an example usage (`example`) and short description (`description`).

Please note that you could have an environment variable with both: a `value` and `example` set. This suggest that this environment variable could be redefined.

Note: Configuration environment variables (without `value`) are not generated to the build source. These can be used instead as a source for generating documentation.

```
envs:
    - name: "STI_BUILDER"
      value: "jee"
    - name: "JBOSS_MODULES_SYSTEM_PKGS"
      value: "org.jboss.logmanager,jdk.nashorn.api"
    - name: "OPENSHIFT_KUBE_PING_NAMESPACE"
      example: "myproject"
      description: "Clustering project namespace."
    - name: "OPENSHIFT_KUBE_PING_LABELS"
      example: "application=eap-app"
      description: "Clustering labels selector."
```

Labels

Note: Learn more about standard labels in container images.

Every image can include labels. Cekit makes it easy to do so with the `labels` section.

```
labels:
    - name: "io.k8s.description"
      value: "Platform for building and running JavaEE applications on JBoss EAP 7.0"
    - name: "io.k8s.display-name"
      value: "JBoss EAP 7.0"
```

Artifacts

It's common for images to require external artifacts like jar files, installers, etc. In most cases you will want to add files into the image and use them during image build process.

Artifacts section is meant exactly for this. *Cekit will automatically fetch any artifacts* specified in this section and check their consistency by computing checksum of the downloaded file and comparing it with the desired value. Currently supported algorithms are: md5, sha1 and sha256. If no algorithm is provided, artifact will be fetched **every** time.

All artifacts are automatically cached during an image build. To learn more about cache please take a look at *Artifact Caching*

The output name for downloaded resources will match the `target` attribute, which defaults to the base name of the file/URL.

Note: For artifacts that are not publicly available Cekit provides a way to add a description detailing a location from which the artifact can be obtained.

```
artifacts:
  - path: jboss-eap-6.4.0.zip
    md5: 9a5d37631919a111ddf42ceda1a9f0b5
    description: "Red Hat JBoss EAP 6.4.0 distribution available on Customer Portal:␣
↪https://access.redhat.com/jbossnetwork/restricted/softwareDetail.html?
↪softwareId=37393&product=appplatform&version=6.4&downloadType=distributions"
```

If Cekit is not able to download an artifact and this artifact has a `description` defined – the build will fail but a message with the description will be printed together with information on where to place the manually downloaded artifact.

Plain

This is the easiest way of defining an artifact. You are just specifying its name and **md5** checksum. This approach relies on *Artifact Caching* to provide the artifact in cache. This section should be used to show that a particular artifact is needed for the image but its not publicly available.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
      target: jolokia.jar
```

Note: See *Red Hat Environment* for a description how Plain Artifacts are affected by Red Hat switch.

URL

This way of defining repository ask Cekit to download and artifact from a specified URL.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      url: https://github.com/rhuss/jolokia/releases/download/v1.3.6/jolokia-1.3.6-
↪bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
```

Path

This way of defining artifact is mostly used in development overrides and enables you to inject an artifact from a local filesystem.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      path: local-artifacts/jolokia-1.3.6-bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
```

Note: If you are using relative `path` to define an artifact, path is considered relative to an image descriptor which introduced that artifact.

Example: If an artifact is defined inside */foo/bar/image.yaml* with a path: *baz/1.zip* the artifact will be resolved as */foo/bar/baz/1.zip*

Packages

To install additional RPM packages you can use the `packages` section where you specify package names and repositories to be used.

```
packages:
    install:
        - mongodb24-mongo-java-driver
        - postgresql-jdbc
        - mysql-connector-java
        - maven
        - hostname
```

Packages are defined in the `install` subsection.

Repositories

Cekit uses all repositories configured inside the image. You can also specify additional repositories using repositories subsection. Cekit currently supports following multiple ways of defining additional repositories:

- Plain

- RPM
- URL
- ContentSets

Note: See *Repository mangement* to learn about best practices for repository definitions.

Plain

This is the default option. With this approach you specify repository id and Cekit will not perform any action and expect the repository definition exists inside the image. This is useful as a hint which repository must be present for particular image to be buildable. The definition can be overridden by your preferred way of injecting repositories inside the image.

```
packages:
    repositories:
        - name: extras
          id: rhel7-extras-rpm
          description: "Repository containing extras RHEL7 extras packages"
```

RPM

This ways is using repository configuration files and related keys packaged as an RPM.

Example: To enable CentOS SCL inside the image you should define repository in a following way:

```
packages:
    repositories:
        - name: scl
          rpm: centos-release-scl
```

URL

This approach enables you to download a yum repository file and corresponding GPG key. To do it, define repositories section in a way of:

```
packages:
    repositories:
        - name: foo
          url:
            repository: https://web.example/foo.repo
            gpg: https://web.exmaple/foo.gpg
```

Content sets

Content sets are tightly integrated to OSBS style of defining repositories in `content_sets.yml` file. If this kind of repository is present in the image descriptor it overrides all other repositories types. For local Docker based build these repositories are ignored similarly to Plain repository types and we expect repository definitions to be available inside image. See upstream docs for more details about content sets.

Note: Behavior of Content sets repositories is changed when running in *Red Hat Environment*.

There are two possibilities how to define Content sets type of repository:

Embedded

In this approach content sets are embedded inside image descriptor under the `content_sets` key.

```
packages:
    content_sets:
        x86_64:
        - server-rpms
        - server-extras-rpms
```

Linked

In this approach Contet sets file is linked from a separate yaml file next to image descriptor via `content_sets_file` key.

Image descriptor:

```
packages:
    content_sets_file: content_sets.yml
```

`content_sets.yml` located next to image descriptor:

```
x86_64:
  - server-rpms
  - server-extras-rpms
```

Ports

This section is used to mark which ports should be exposed in the container. If we want to highlight a port used in the container, but not necessary expose it – we should set the `expose` flag to `false` (`true` by default).

You can provide additional documentation as to the usage of the port with the keys `protocol`, to specify which IP protocol is used over the port number (e.g TCP, UDP...) and `service` to describe what network service is running on top of the port (e.g. "http", "https"). You can provide a human-readable long form description of the port with the `description` key.

```
ports:
    - value: 8443
      service: https
    - value: 8778
      expose: false
      protocol: tcp
      description: internal port for frob communication.
```

User

Specifies the user (can be username or uid) that should be used to launch the entrypoint process.

```
run:
    user: "alice"
```

Volumes

In case you want to define volumes for your image, just use the `volumes` section!

```
volumes:
    - name: "volume.eap"
      path: "/opt/eap/standalone"
```

Note: The `name` key is optional. If not specified the value of `path` key will be used.

Modules

Module repositories

Module repositories specify location of modules that are to be incorporated into the image. These repositories may be `git` repositories or directories on the local file system (`path`). Cekit will scan the repositories for `module.xml` files, which are used to encapsulate image details that may be incorporated into multiple images.

```
modules:
  repositories:
      # Modules pulled from Java image project on GitHub
    - git:
          url: https://github.com/jboss-container-images/redhat-openjdk-18-openshift-
↪image
          ref: 1.0

      # Modules pulled locally from "custom-modules" directory, collocated with image␣
↪descriptor
    - path: custom-modules
```

Module installation

The `install` section is used to define what modules should be installed in the image in what order. Name used to specify the module is the `name` field from the module descriptor.

```
modules:
  install:
      - name: xpaas.java
      - name: xpaas.amq.install
```

You can even request specific module version via *version* key as follows:

```
modules:
  install:
      - name: xpaas.java
        version: 1.2-dev
      - name: xpaas.amq.install
```

Workdir

Sets the current working directory of the entrypoint process in the container.

```
run:
    workdir: "/home/jboss"
```

Run

The `run` section encapsulates instructions related to launching main process in the container including: `cmd`, `entrypoint`, `user` and `workdir`. All subsections are described later in this paragraph.

Below you can find full example that uses every possible option.

```
run:
    cmd:
        - "argument1"
        - "argument2"
    entrypoint:
        - "/opt/eap/bin/wrapper.sh"
    user: "alice"
    workdir: "/home/jboss"
```

Cmd

Command that should be executed by the container at run time.

```
run:
    cmd:
        - "some cmd"
        - "argument"
```

Entrypoint

Entrypoint that should be executed by the container at run time.

```
run:
    entrypoint:
        - "/opt/eap/bin/wrapper.sh"
```

OSBS

This section represents object we use to hint OSBS builder with a configuration which needs to be tweaked for successful and reproducible builds.

It contains two main keys:

- `repository`
- `configuration`

Repository

This key serves as a hint which DistGit repository and its branch we use to push generated sources into.

Example:

```
osbs:
    repository:
          name: containers/redhat-openjdk-18
          branch: jb-openjdk-1.8-openshift-rhel-7
```

Configuration

This key is holding OSBS `container.yaml` file (**:ref:'docs<https://osbs.readthedocs.io/en/latest/users.html?highlight=container configuration>'_**) `container.yaml` file can be embedded in `container` key or inject from a file specified in `container_file` key.

Embedded

In this case whole `container.yaml` file is embedded in an image descriptor.

```
osbs:
    configuration:
       container:
          compose:
              pulp_repos: true
```

Linked

In this case `container.yaml` file is read from a file located next to the image descriptor.

```
osbs:
    configuration:
        container_file: container.yaml
```

and `container.yaml` file contains:

```
compose:
    pulp_repos: true
```

5.2.2 Module descriptor

Module descriptor contains all information Cekit needs to introduce a feature to an image. Modules are used as libraries or shared building blocks across images.

It is very important to make a module self-containg which means that executing scripts defined in the module's definition file should always end up in a state where you could define the module as being *installed*.

Modules can be stacked – some modules will be run before, some after you module. Please keep that in mind that at the time when you are developing a module – you don't know how and when it'll be executed.

Contents

name

This key is **required**.

Module name.

```
name: "python_flask_module"
```

Version

This key is **required**.

Version of the image.

```
version: "1.4"
```

Description

Short summary of the image.

Value of the `description` key is added to the image as two labels: `description` and `summary` unless such labels are already defined in the image descriptor's *Labels* section.

```
description: "Red Hat JBoss Enterprise Application 7.0 - An application platform for␣
↪hosting your apps that provides an innovative modular, cloud-ready architecture,␣
↪powerful management and automation, and world class developer productivity."
```

From

This key is **required**.

Base image of your image.

```
from: "jboss-eap-7-tech-preview/eap70:1.2"
```

Environment variables

Similar to labels – we can specify environment variables that should be present in the container after running the image. We provide `envs` section for this.

Environment variables can be divided into two types:

1. **Information environment variables** – these are set and available in the image. This type of environment variables provide information to the image consumer. In most cases such environment variables *should not* be modified.
2. **Configuration environment variables** – this type of variables are used to define environment variables used to configure services inside running container.

   These environment variables are **not** set during image build time but *can* be set at run time.

   Every configuration enviromnent variable should provide an example usage (`example`) and short description (`description`).

Please note that you could have an environment variable with both: a `value` and `example` set. This suggest that this environment variable could be redefined.

Note: Configuration environment variables (without `value`) are not generated to the build source. These can be used instead as a source for generating documentation.

```
envs:
    - name: "STI_BUILDER"
      value: "jee"
    - name: "JBOSS_MODULES_SYSTEM_PKGS"
      value: "org.jboss.logmanager,jdk.nashorn.api"
    - name: "OPENSHIFT_KUBE_PING_NAMESPACE"
      example: "myproject"
      description: "Clustering project namespace."
    - name: "OPENSHIFT_KUBE_PING_LABELS"
      example: "application=eap-app"
      description: "Clustering labels selector."
```

Labels

Note: Learn more about standard labels in container images.

Every image can include labels. Cekit makes it easy to do so with the `labels` section.

```
labels:
    - name: "io.k8s.description"
      value: "Platform for building and running JavaEE applications on JBoss EAP 7.0"
    - name: "io.k8s.display-name"
      value: "JBoss EAP 7.0"
```

Artifacts

It's common for images to require external artifacts like jar files, installers, etc. In most cases you will want to add files into the image and use them during image build process.

Artifacts section is meant exactly for this. *Cekit will automatically fetch any artifacts* specified in this section and check their consistency by computing checksum of the downloaded file and comparing it with the desired value. Currently supported algorithms are: md5, sha1 and sha256. If no algorithm is provided, artifact will be fetched **every** time.

All artifacts are automatically cached during an image build. To learn more about cache please take a look at *Artifact Caching*

The output name for downloaded resources will match the `target` attribute, which defaults to the base name of the file/URL.

Note: For artifacts that are not publicly available Cekit provides a way to add a description detailing a location from which the artifact can be obtained.

```
artifacts:
  - path: jboss-eap-6.4.0.zip
    md5: 9a5d37631919a111ddf42ceda1a9f0b5
    description: "Red Hat JBoss EAP 6.4.0 distribution available on Customer Portal:␣
↪https://access.redhat.com/jbossnetwork/restricted/softwareDetail.html?
↪softwareId=37393&product=appplatform&version=6.4&downloadType=distributions"
```

(continues on next page)

(continued from previous page)

If Cekit is not able to download an artifact and this artifact has a `description` defined – the build will fail but a message with the description will be printed together with information on where to place the manually downloaded artifact.

Plain

This is the easiest way of defining an artifact. You are just specifying its name and **md5** checksum. This approach relies on *Artifact Caching* to provide the artifact in cache. This section should be used to show that a particular artifact is needed for the image but its not publicly available.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
      target: jolokia.jar
```

Note: See *Red Hat Environment* for a description how Plain Artifacts are affected by Red Hat switch.

URL

This way of defining repository ask Cekit to download and artifact from a specified URL.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      url: https://github.com/rhuss/jolokia/releases/download/v1.3.6/jolokia-1.3.6-
↪bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
```

Path

This way of defining artifact is mostly used in development overrides and enables you to inject an artifact from a local filesystem.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      path: local-artifacts/jolokia-1.3.6-bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
```

Note: If you are using relative `path` to define an artifact, path is considered relative to an image descriptor which introduced that artifact.

Example: If an artifact is defined inside */foo/bar/image.yaml* with a path: *baz/1.zip* the artifact will be resolved as */foo/bar/baz/1.zip*

Packages

To install additional RPM packages you can use the `packages` section where you specify package names and repositories to be used.

```
packages:
    install:
        - mongodb24-mongo-java-driver
        - postgresql-jdbc
        - mysql-connector-java
        - maven
        - hostname
```

Packages are defined in the `install` subsection.

Repositories

Cekit uses all repositories configured inside the image. You can also specify additional repositories using repositories subsection. Cekit currently supports following multiple ways of defining additional repositories:

- Plain
- RPM
- URL
- ContentSets

Note: See *Repository mangement* to learn about best practices for repository definitions.

Plain

This is the default option. With this approach you specify repository id and Cekit will not perform any action and expect the repository definition exists inside the image. This is useful as a hint which repository must be present for particular image to be buildable. The definition can be overridden by your preferred way of injecting repositories inside the image.

```
packages:
    repositories:
        - name: extras
          id: rhel7-extras-rpm
          description: "Repository containing extras RHEL7 extras packages"
```

RPM

This ways is using repository configuration files and related keys packaged as an RPM.

Example: To enable CentOS SCL inside the image you should define repository in a following way:

```
packages:
    repositories:
        - name: scl
          rpm: centos-release-scl
```

URL

This approach enables you to download a yum repository file and corresponding GPG key. To do it, define repositories section in a way of:

```
packages:
    repositories:
        - name: foo
          url:
            repository: https://web.example/foo.repo
            gpg: https://web.exmaple/foo.gpg
```

Content sets

Content sets are tightly integrated to OSBS style of defining repositories in `content_sets.yml` file. If this kind of repository is present in the image descriptor it overrides all other repositories types. For local Docker based build these repositories are ignored similarly to Plain repository types and we expect repository definitions to be available inside image. See upstream docs for more details about content sets.

Note: Behavior of Content sets repositories is changed when running in *Red Hat Environment*.

There are two possibilities how to define Content sets type of repository:

Embedded

In this approach content sets are embedded inside image descriptor under the `content_sets` key.

```
packages:
    content_sets:
        x86_64:
        - server-rpms
        - server-extras-rpms
```

Linked

In this approach Contet sets file is linked from a separate yaml file next to image descriptor via `content_sets_file` key.

Image descriptor:

```
packages:
    content_sets_file: content_sets.yml
```

`content_sets.yml` located next to image descriptor:

```
x86_64:
  - server-rpms
  - server-extras-rpms
```

Ports

This section is used to mark which ports should be exposed in the container. If we want to highlight a port used in the container, but not necessary expose it – we should set the `expose` flag to `false` (`true` by default).

You can provide additional documentation as to the usage of the port with the keys `protocol`, to specify which IP protocol is used over the port number (e.g TCP, UDP...) and `service` to describe what network service is running on top of the port (e.g. "http", "https"). You can provide a human-readable long form description of the port with the `description` key.

```
ports:
    - value: 8443
      service: https
    - value: 8778
      expose: false
      protocol: tcp
      description: internal port for frob communication.
```

User

Specifies the user (can be username or uid) that should be used to launch the entrypoint process.

```
run:
    user: "alice"
```

Volumes

In case you want to define volumes for your image, just use the `volumes` section!

```
volumes:
    - name: "volume.eap"
      path: "/opt/eap/standalone"
```

Note: The `name` key is optional. If not specified the value of `path` key will be used.

Modules

Module repositories

Module repositories specify location of modules that are to be incorporated into the image. These repositories may be `git` repositories or directories on the local file system (`path`). Cekit will scan the repositories for `module.xml` files, which are used to encapsulate image details that may be incorporated into multiple images.

```
modules:
  repositories:
      # Modules pulled from Java image project on GitHub
    - git:
          url: https://github.com/jboss-container-images/redhat-openjdk-18-openshift-
↪image
          ref: 1.0
```

(continues on next page)

(continued from previous page)

```
    # Modules pulled locally from "custom-modules" directory, collocated with image␣
↪descriptor
    - path: custom-modules
```

Module installation

The `install` section is used to define what modules should be installed in the image in what order. Name used to specify the module is the `name` field from the module descriptor.

```
modules:
  install:
      - name: xpaas.java
      - name: xpaas.amq.install
```

You can even request specific module version via *version* key as follows:

```
modules:
  install:
      - name: xpaas.java
        version: 1.2-dev
      - name: xpaas.amq.install
```

Workdir

Sets the current working directory of the entrypoint process in the container.

```
run:
    workdir: "/home/jboss"
```

Run

The `run` section encapsulates instructions related to launching main process in the container including: `cmd`, `entrypoint`, `user` and `workdir`. All subsections are described later in this paragraph.

Below you can find full example that uses every possible option.

```
run:
    cmd:
        - "argument1"
        - "argument2"
    entrypoint:
        - "/opt/eap/bin/wrapper.sh"
    user: "alice"
    workdir: "/home/jboss"
```

Cmd

Command that should be executed by the container at run time.

```
run:
    cmd:
        - "some cmd"
        - "argument"
```

Entrypoint

Entrypoint that should be executed by the container at run time.

```
run:
    entrypoint:
        - "/opt/eap/bin/wrapper.sh"
```

Execute

Execute section defines what needs to be done to install this module in the image. Every execution listed in this section will be run at image build time in the order as defined.

```
execute:
      # The install.sh file will be executed first as root user
    - script: install.sh
      # Then the redefine.sh file will be executed as jboss user
    - script: redefine.sh
      user: jboss
```

Note: When no `user` is defined, `root` user will be used to execute the script.

5.2.3 Overrides descriptor

Contents

Name

This key is **required**.

Image name without the registry part.

```
name: "jboss-eap-7/eap70-openshift"
```

Version

This key is **required**.

Version of the image.

```
version: "1.4"
```

Description

Short summary of the image.

Value of the `description` key is added to the image as two labels: `description` and `summary` unless such labels are already defined in the image descriptor's *Labels* section.

```
description: "Red Hat JBoss Enterprise Application 7.0 - An application platform for␣
↪hosting your apps that provides an innovative modular, cloud-ready architecture,␣
↪powerful management and automation, and world class developer productivity."
```

From

This key is **required**.

Base image of your image.

```
from: "jboss-eap-7-tech-preview/eap70:1.2"
```

Environment variables

Similar to labels – we can specify environment variables that should be present in the container after running the image. We provide `envs` section for this.

Environment variables can be divided into two types:

1. **Information environment variables** – these are set and available in the image. This type of environment variables provide information to the image consumer. In most cases such environment variables *should not* be modified.
2. **Configuration environment variables** – this type of variables are used to define environment variables used to configure services inside running container.

   These environment variables are **not** set during image build time but *can* be set at run time.

   Every configuration enviromnent variable should provide an example usage (`example`) and short description (`description`).

Please note that you could have an environment variable with both: a `value` and `example` set. This suggest that this environment variable could be redefined.

Note: Configuration environment variables (without `value`) are not generated to the build source. These can be used instead as a source for generating documentation.

```
envs:
    - name: "STI_BUILDER"
      value: "jee"
    - name: "JBOSS_MODULES_SYSTEM_PKGS"
      value: "org.jboss.logmanager,jdk.nashorn.api"
    - name: "OPENSHIFT_KUBE_PING_NAMESPACE"
      example: "myproject"
      description: "Clustering project namespace."
    - name: "OPENSHIFT_KUBE_PING_LABELS"
      example: "application=eap-app"
      description: "Clustering labels selector."
```

Labels

Note: Learn more about standard labels in container images.

Every image can include labels. Cekit makes it easy to do so with the `labels` section.

```
labels:
    - name: "io.k8s.description"
      value: "Platform for building and running JavaEE applications on JBoss EAP 7.0"
    - name: "io.k8s.display-name"
      value: "JBoss EAP 7.0"
```

Artifacts

It's common for images to require external artifacts like jar files, installers, etc. In most cases you will want to add files into the image and use them during image build process.

Artifacts section is meant exactly for this. *Cekit will automatically fetch any artifacts* specified in this section and check their consistency by computing checksum of the downloaded file and comparing it with the desired value. Currently supported algorithms are: md5, sha1 and sha256. If no algorithm is provided, artifact will be fetched **every** time.

All artifacts are automatically cached during an image build. To learn more about cache please take a look at *Artifact Caching*

The output name for downloaded resources will match the `target` attribute, which defaults to the base name of the file/URL.

Note: For artifacts that are not publicly available Cekit provides a way to add a description detailing a location from which the artifact can be obtained.

```
artifacts:
  - path: jboss-eap-6.4.0.zip
    md5: 9a5d37631919a111ddf42ceda1a9f0b5
    description: "Red Hat JBoss EAP 6.4.0 distribution available on Customer Portal:␣
↪https://access.redhat.com/jbossnetwork/restricted/softwareDetail.html?
↪softwareId=37393&product=appplatform&version=6.4&downloadType=distributions"
```

If Cekit is not able to download an artifact and this artifact has a `description` defined – the build will fail but a message with the description will be printed together with information on where to place the manually downloaded artifact.

Plain

This is the easiest way of defining an artifact. You are just specifying its name and **md5** checksum. This approach relies on *Artifact Caching* to provide the artifact in cache. This section should be used to show that a particular artifact is needed for the image but its not publicly available.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
      target: jolokia.jar
```

Note: See *Red Hat Environment* for a description how Plain Artifacts are affected by Red Hat switch.

URL

This way of defining repository ask Cekit to download and artifact from a specified URL.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      url: https://github.com/rhuss/jolokia/releases/download/v1.3.6/jolokia-1.3.6-
↪bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
```

Path

This way of defining artifact is mostly used in development overrides and enables you to inject an artifact from a local filesystem.

```
artifacts:
    - name: jolokia-1.3.6-bin.tar.gz
      path: local-artifacts/jolokia-1.3.6-bin.tar.gz
      md5: 75e5b5ba0b804cd9def9f20a70af649f
```

Note: If you are using relative `path` to define an artifact, path is considered relative to an image descriptor which introduced that artifact.

Example: If an artifact is defined inside */foo/bar/image.yaml* with a path: *baz/1.zip* the artifact will be resolved as */foo/bar/baz/1.zip*

Packages

To install additional RPM packages you can use the `packages` section where you specify package names and repositories to be used.

```
packages:
    install:
        - mongodb24-mongo-java-driver
        - postgresql-jdbc
        - mysql-connector-java
        - maven
        - hostname
```

Packages are defined in the `install` subsection.

Repositories

Cekit uses all repositories configured inside the image. You can also specify additional repositories using repositories subsection. Cekit currently supports following multiple ways of defining additional repositories:

- Plain
- RPM
- URL
- ContentSets

Note: See *Repository mangement* to learn about best practices for repository definitions.

Plain

This is the default option. With this approach you specify repository id and Cekit will not perform any action and expect the repository definition exists inside the image. This is useful as a hint which repository must be present for particular image to be buildable. The definition can be overridden by your preferred way of injecting repositories inside the image.

```
packages:
    repositories:
        - name: extras
          id: rhel7-extras-rpm
          description: "Repository containing extras RHEL7 extras packages"
```

RPM

This ways is using repository configuration files and related keys packaged as an RPM.

Example: To enable CentOS SCL inside the image you should define repository in a following way:

```
packages:
    repositories:
        - name: scl
          rpm: centos-release-scl
```

URL

This approach enables you to download a yum repository file and corresponding GPG key. To do it, define repositories section in a way of:

```
packages:
    repositories:
        - name: foo
          url:
            repository: https://web.example/foo.repo
            gpg: https://web.exmaple/foo.gpg
```

Content sets

Content sets are tightly integrated to OSBS style of defining repositories in `content_sets.yml` file. If this kind of repository is present in the image descriptor it overrides all other repositories types. For local Docker based build these repositories are ignored similarly to Plain repository types and we expect repository definitions to be available inside image. See upstream docs for more details about content sets.

Note: Behavior of Content sets repositories is changed when running in *Red Hat Environment*.

There are two possibilities how to define Content sets type of repository:

Embedded

In this approach content sets are embedded inside image descriptor under the `content_sets` key.

```
packages:
    content_sets:
        x86_64:
        - server-rpms
        - server-extras-rpms
```

Linked

In this approach Contet sets file is linked from a separate yaml file next to image descriptor via `content_sets_file` key.

Image descriptor:

```
packages:
    content_sets_file: content_sets.yml
```

`content_sets.yml` located next to image descriptor:

```
x86_64:
  - server-rpms
  - server-extras-rpms
```

Ports

This section is used to mark which ports should be exposed in the container. If we want to highlight a port used in the container, but not necessary expose it – we should set the `expose` flag to `false` (`true` by default).

You can provide additional documentation as to the usage of the port with the keys `protocol`, to specify which IP protocol is used over the port number (e.g TCP, UDP...) and `service` to describe what network service is running on top of the port (e.g. "http", "https"). You can provide a human-readable long form description of the port with the `description` key.

```
ports:
    - value: 8443
      service: https
    - value: 8778
      expose: false
      protocol: tcp
      description: internal port for frob communication.
```

User

Specifies the user (can be username or uid) that should be used to launch the entrypoint process.

```
run:
    user: "alice"
```

Volumes

In case you want to define volumes for your image, just use the `volumes` section!

```
volumes:
    - name: "volume.eap"
      path: "/opt/eap/standalone"
```

Note: The `name` key is optional. If not specified the value of `path` key will be used.

Modules

Module repositories

Module repositories specify location of modules that are to be incorporated into the image. These repositories may be `git` repositories or directories on the local file system (`path`). Cekit will scan the repositories for `module.xml` files, which are used to encapsulate image details that may be incorporated into multiple images.

```
modules:
  repositories:
      # Modules pulled from Java image project on GitHub
    - git:
          url: https://github.com/jboss-container-images/redhat-openjdk-18-openshift-
↪image
          ref: 1.0

      # Modules pulled locally from "custom-modules" directory, collocated with image␣
↪descriptor
    - path: custom-modules
```

Module installation

The `install` section is used to define what modules should be installed in the image in what order. Name used to specify the module is the `name` field from the module descriptor.

```
modules:
  install:
      - name: xpaas.java
      - name: xpaas.amq.install
```

You can even request specific module version via *version* key as follows:

```
modules:
  install:
      - name: xpaas.java
        version: 1.2-dev
      - name: xpaas.amq.install
```

Workdir

Sets the current working directory of the entrypoint process in the container.

```
run:
    workdir: "/home/jboss"
```

Run

The `run` section encapsulates instructions related to launching main process in the container including: `cmd`, `entrypoint`, `user` and `workdir`. All subsections are described later in this paragraph.

Below you can find full example that uses every possible option.

```
run:
    cmd:
        - "argument1"
        - "argument2"
    entrypoint:
        - "/opt/eap/bin/wrapper.sh"
    user: "alice"
    workdir: "/home/jboss"
```

Cmd

Command that should be executed by the container at run time.

```
run:
    cmd:
        - "some cmd"
        - "argument"
```

Entrypoint

Entrypoint that should be executed by the container at run time.

```
run:
    entrypoint:
        - "/opt/eap/bin/wrapper.sh"
```

5.2.4 Configuration file

Cekit can be configured using a configuration file. We use the properties file format.

Cekit will look for this file at the path `~/.cekit/config`. Its location can be changed via command line `--config` option.

Example Running Cekit with different config file:

```
$ cekit --config ~/alternative_path build
```

Below you can find description of available sections together with options described in detail.

Contents

common

work_dir

Contains location of Cekit working directory, which is used to store some persistent data like dist_git repositories and artifact cache.

```
[common]
work_dir=/tmp
```

ssl_verify

Controls verification of SSL certificates for example when downloading artifacts. Default: `True`.

```
[common]
ssl_verify = False
```

cache_url

Specifies a different location that could be used to fetch artifacts. Usually this is a URL to some cache service. By default it is not set.

You can use following substitutions:

- `#filename#`–the file name from the url of the artifact
- `#algorithm#`–has algorithm specified for the selected artifact
- `#hash#`–value of the digest.

Example

Consider you have an image definition with artifacts section like this:

```
artifacts:
    - url: "http://some.host.com/7.0.0/jboss-eap-7.0.0.zip"
      md5: cd02482daa0398bf5500e1628d28179a
```

If we set the `cache_url` parameter in following way:

```
[common]
cache_url = http://cache.host.com/fetch?#algorithm#=#hash#
```

The JBoss EAP artifact will be fetched from: `http://cache.host.com/fetch?md5=cd02482daa0398bf5500e1628d28179a`.

And if we do it like this:

```
[common]
cache_url = http://cache.host.com/cache/#filename#
```

The JBoss EAP artifact will be fetched from: `http://cache.host.com/cache/jboss-eap-7.0.0.zip`.

Note: In all cases digest will be computed from the downloaded file and compared with the expected value.

redhat

This option changes Cekit default options to comply with Red Hat internal infrastructure and policies.

Example: To enable this flag add following lines into your `~/.cekit/config` file:

```
[common]
redhat = true
```

Note: If you are using Cekit within Red Hat infrastructure you should have valid Kerberos ticket.

doc

This section collects together configuration options relating to documentation.

addhelp

This option instructs Cekit to install the generated *help.md* file into the generate image sources. The file is inserted at the root path (*/*). The default value is False.

Example: To enable this flag add following lines into your `~/.cekit/config` file:

```
[doc]
addhelp = true
```

help_template

This option overrides the default Jinja template used in the generation of *help.md* files.

Example:

```
[doc]
help_template = /home/jon/something/my_help.md
```

5.3 Cekit Tutorial

Welcome in Cekit Tutorial, we will guide you through building your first image with Cekit. We expect you to have Cekit installed already. IF you need help with installation process please follow our installation intructions.

5.3.1 Creating Image Descriptor

Work in progress

5.3.2 Moving shareable code inside Modules

Work in progress

5.3.3 Creating Test

Work in progress