Eskimo Service Developer Guide

eskimo.sh / https://www.eskimo.sh / 2019-2020

Table of Contents

1. Introduction 1
    1.1. Eskimo 1
    1.2. Key Features 2
    1.3. The Service Developer Guide 3
2. Introducing the Service Development Framework 4
    2.1. Principle schema 4
    2.2. Core principles 4
    2.3. A note on images template download 5
3. Docker Images Development Framework 7
    3.1. Requirements 7
    3.2. Principle 7
    3.3. property 'system.packagesToBuild' in eskimo.properties 8
    3.4. Standards and conventions over requirements 8
    3.5. Typical build.sh process 9
        3.5.1. Operations performed 9
    3.6. Look for examples and get inspired 10
    3.7. Apache Mesos Building 10
        3.7.1. Building Mesos 10
    3.8. Specific and various notes related to individual components shipped with Eskimo . 11
        3.8.1. Zeppelin building 11
    3.9. Setting up a remote packages repository 11
4. Services Installation Framework 14
    4.1. Principle 14
        4.1.1. Gluster share mounts 14
        4.1.2. OS System Users creation 15
    4.2. Standards and conventions over requirements 15
    4.3. Typical setup.sh process 15
        4.3.1. Operations performed 16
        4.3.2. Standard and conventions 17
        4.3.3. Look for examples and get inspired 17
    4.4. Eskimo services configuration 17
        4.4.1. Configuration file `services.json` 17
        4.4.2. Eskimo Topology and dependency management 21
        4.4.3. Master Election strategy 21
        4.4.4. Memory allocation 23
            Native (SystemD) services memory configuration 24

Examples of memory allocation 24
Marathon services memory configuration 25
4.4.5. Topology file on cluster nodes 26
4.5. Proxying services web consoles 26
4.5.1. Rewrite rules 27
4.5.2. Standard rewrite rules 27
4.5.3. Custom rewrite rules 28
Appendix A: Copyright and License 30

Chapter 1. Introduction

1.1. Eskimo

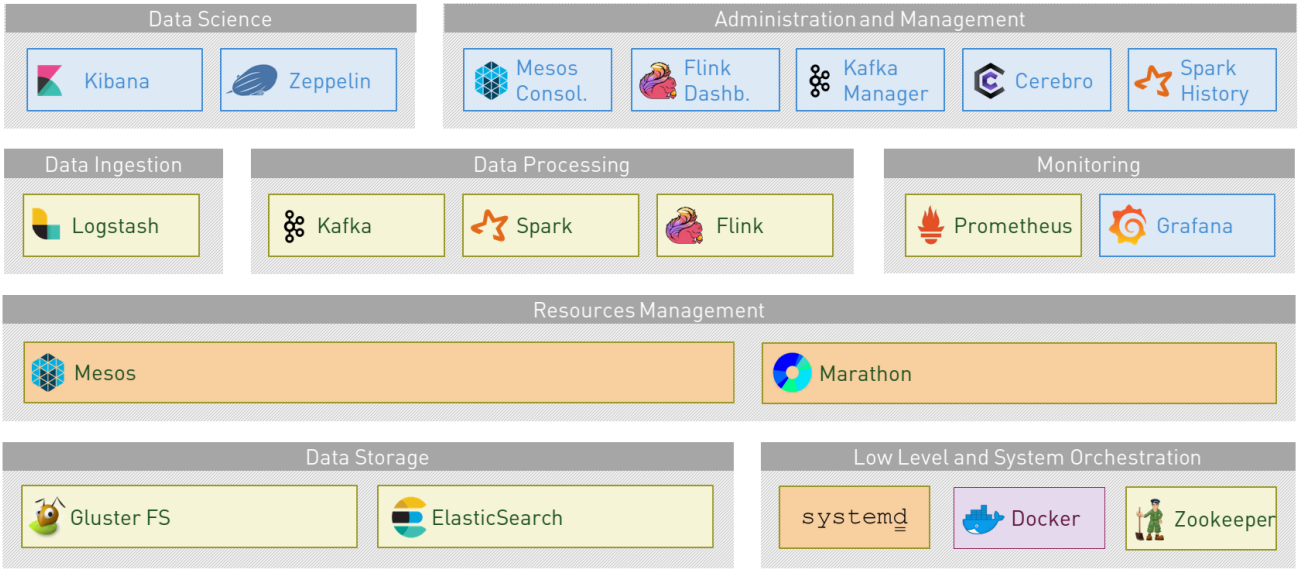
A state of the art *Big Data Infrastructure* and *Management Web Console* to *build*, *manage* and *operate* **Big Data 2.0 Analytics clusters**

Eskimo is in a certain way the Operating System of your Big Data Cluster:

- A *plug and play*, working *out of the Box*, **Big Data Analytics platform** fulfilling *enterprise environment requirements*.
- A **state of the art Big Data 2.0 platform**
  - based on *Docker*, *Marathon*, *Mesos* and *Systemd*
  - packaging *Gluster*, *Spark*, *Kafka*, *Flink*, *Nifi* and *ElasticSearch*
  - with all the administration and management consoles such as *Cerebro*, *Kibana*, *Zeppelin*, *Kafka-Manager*, *Grafana* and *Prometheus*.
- An *Administration Application* aimed at drastically simplifying the **deployment, administration and operation** of your Big Data Cluster
- A *Data Science Laboratory* and *Production environment* where Data Analytics is both
  - developed and
  - operated in production

Eskimo is as well:

- a collection of ready to use docker containers packaging fine-tuned and highly customized plug and play services with all the *nuts and bolds* required to make them work well together.
- a framework for building and deploying Big Data and NoSQL services based on docker and systemd



1.2. Key Features

Eskimo key features are as follows:

	Abstraction of Location Just define where you want to run which services and let eskimo take care of everything. Move services between nodes or install new services in just a few clicks. Don't bother remembering where you installed Web consoles and UI applications, Eskimo wraps them all in a single and consistent UI.
	Eskimo Web Console Eskimo's tip of the iceberg is its flagship web console. The Eskimo Console is the single and only entry point to all your cluster operations, from services installation to accessing Kibana, Zeppelin and other UI applications. The Eskimo Console also provides SSH consoles, File browser access and monitoring to your cluster.

	**Services Framework** Eskimo is a Big Data Components service development and integration framework based on Docker and Systemd. Eskimo provides out of the box ready-to use components such as Spark, Flink, ElasticSearch, Kafka, Mesos, Zeppelin, etc. Eskimo also enables the user to develop his own services very easily.

1.3. The Service Developer Guide

This documentation is related to the last of the key features presented above : The Services Development Framework.
It presents everything a developer needs to understand and know to develop his own services and let eskimo distribute and operate them, or extend current services.

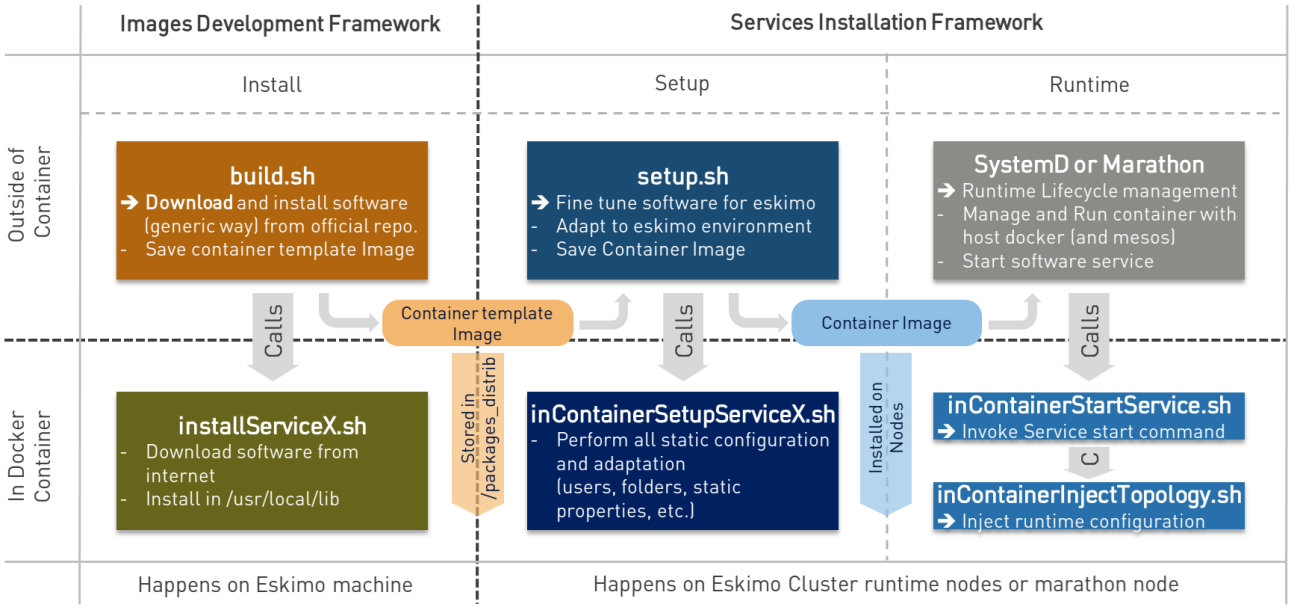
Chapter 2. Introducing the Service Development Framework

The **Service Development framework** is actually composed by two distinct parts:

- The Docker Images Development Framework which is used to **build** the docker images deployed on the eskimo cluster nodes
- The Services Installation Framework which is used to **install and setup** these images as services on the eskimo cluster nodes.

2.1. Principle schema

The whole services development framework can be represented as follows:



- The Docker Images Development Framework is used to develop and build Container templates images that are later used to build the actual docker containers images that are fine-tuned to Eskimo deployed on invidiual Eskimo cluster nodes
- The Services Installation Framework which create the actual container image from the template and adapts it to the specific situation of the eskimo node it is running on.

The different scripts involved in the different stages are presented on the schema above alon with their responsibilities and the environment in which they are executed (outside of the container - on the eskimo host machine or the eskimo cluster node - or from within the docker container).

2.2. Core principles

The core principles on which both the *Docker Images Development Framework* and the *Services Installation Framework* are built are as follows:

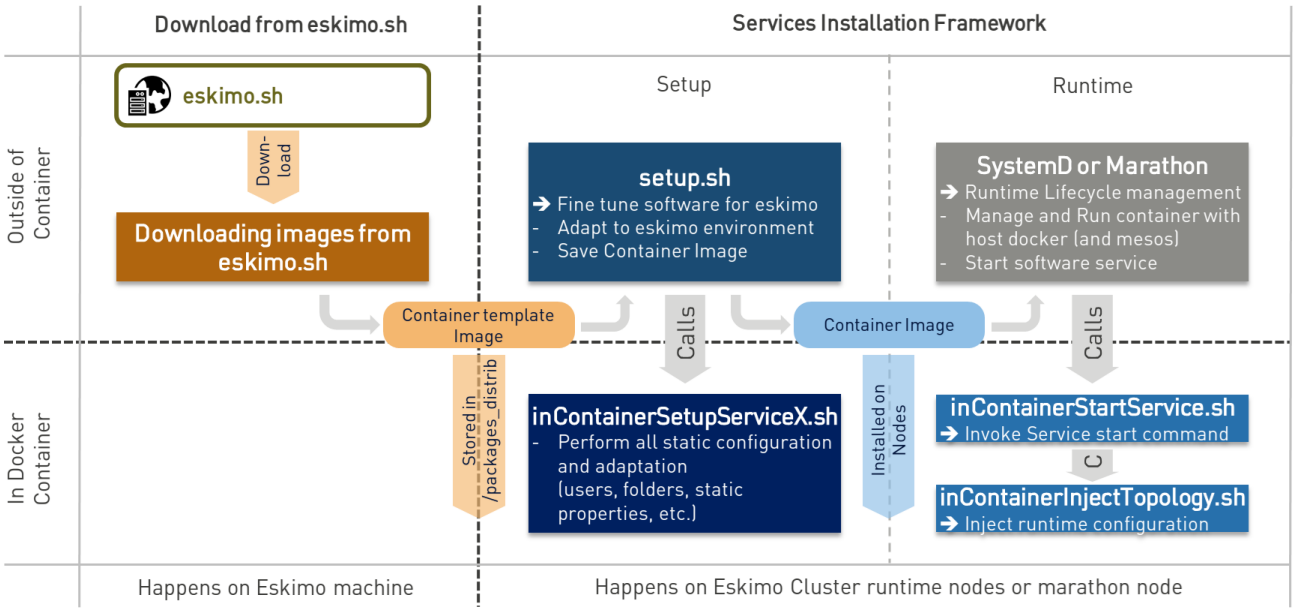
- A service is eventually two things
  - a docker image packaging the software component and its dependencies
  - a set of shell scripts aimed at installing and setting up the service
  - a systemd unit configuration file aimed at operate it
- Everything - from building the docker image to installing it on cluster nodes - is done using simple shell scripts. With Eskimo a system administrator desiring to leverage on eskimo to implement his own services in order to integrate additional software components on Eskimo doesn't need to learn any new and fancy technology, just plain old shell scripting, docker, systemd and perhaps a little bit of marathon configuration; that's it.
- Eskimo leverages on unix standards. Software components are installed in `/usr/local`, log files are in sub-folders of `/var/log`, persistent data is in sub-folders of `/var/lib`, etc.

2.3. A note on images template download.

At eskimo first start, the user (administrator) needs to choose whether he wants to build the services images (docker container templates images) locally or download them from *eskimo.sh*.

- **Buidling the images locally** means creating each and every docker template image using the `build.sh` script. Only the vanilly software packages are downloaded from internet (such as the elasticsearch distribution, spark, flink, etc.).
  This can take a significant time, several dozens of minutes.
- **Downloading the images from internet** means downloading *ready-to-use* docker template images from *eskimo.sh*

The whole setup process when downloading the images from internet can be represented this way:


Download from eskimo.sh
Services Installation Framework
Outside of Container
In Docker Container
eskimo.sh
Down-load
Downloading images from eskimo.sh
Setup
Runtime
setup.sh
➔ Fine tune software for eskimo
- Adapt to eskimo environment
- Save Container Image
SystemD or Marathon
➔ Runtime Lifecycle management
- Manage and Run container with host docker (and mesos)
- Start software service
Container template Image
Calls
Container Image
Calls
Stored in /packages_distrib
inContainerSetupServiceX.sh
- Perform all static configuration and adaptation (users, folders, static properties, etc.)
Installed on Nodes
inContainerStartService.sh
➔ Invoke Service start command
C
inContainerInjectTopology.sh
➔ Inject runtime configuration
Happens on Eskimo machine
Happens on Eskimo Cluster runtime nodes or marathon node

Chapter 3. Docker Images Development Framework

The Docker Images Development Framework provides tools and define standards to build the docker images deployed on the eskimo cluster nodes.

Eskimo is leveraging on docker to deploy services across the cluster ndes and to ensure independence from host OSes and specificities as well as proper isolation between services.

Docker images can be downloaded from internet (or the internal corporation network) or build locally on the machine running eskimo.
The later is achieved using the packages development framework which is presented in this guide.

3.1. Requirements

In order for one to be able to either build the eskimo software components packages locally (or have eskimo building them locally), there are a few requirements:

- Docker must be installed and available on the user local computer (or the eskimo host machine in case Eskimo builds them)
- At least 10Gb of hard drive space on the partition hosting `/tmp` must be available.
- At least 15Gb of hard drive space on the partition histing the eskimo installation folder must be available.

3.2. Principle

The principle is pretty straightforward

- Every docker image (or package) is built by calling `build.sh` from the corresponding package folder, i.e. a sub-folder of this very folder `packages_dev`
- That `build.sh` script is free to perform whatever it wants as long as the result is a docker image with the expected name put in the folder `packages_distrib` in the parent folder and packaging the target software component.

Build files are provided for each and every service pre-packaged within eskimo.

The user is welcome to modify them in order to fine tune everything the way he wants or implement his own packages for other software components not yet bundled with eskimo. Again, the only requirement is that at the end of the build process a corresponding image is put in `packages_distrib` as well as that some conventions are properly followed as explained below.

Internet is usually required on the eskimo machine to build or re-build the eskimo provided

pre-packages images since the target software component is downloaded from Internet.

3.3. property 'system.packagesToBuild' in eskimo.properties

Eskimo needs a way during initial setup time to know which packages need to be built or downloaded.

This is indicated by the property `system.packagesToBuild` in the configuration file `eskimo.properties`, such as, for instance:

default `system.packagesToBuild` *property*

```
system.packagesToBuild=base-
eskimo,ntp,zookeeper,gluster,gdash,elasticsearch,cerebro,kibana,logstash,p
rometheus,grafana,kafka,kafka-manager,mesos-master,spark,zeppelin
```

Whenever one wants to implement a new package for a new service to be managed by eskimo, one needs to declare this new package in the given list.

3.4. Standards and conventions over requirements

There are no requirements when building an eskimo package docker image. The image developer is free to install the software package within the image the way he wants and no specific requirement is enforced by eskimo.

As long as eventually, *The Service Installation framework* for the new software package provides a systemd unit configuration file enabling the eskimo framework to manipulate the service, a service developer has all the freedom to design his docker container the way he wants.

However, adhering to some conventions eases a lot the maintenance and evolution of these images.
These standard conventions are as follows:

- All docker images are based on the `base-eskimo` image which is itself based on a lightweight debian stretch image.
- A software component named *X* with *version Y* should be installed in `/usr/local/lib/x-y` (if and only if it is not available in apt repositories and installed using standard debian `apt-get install x`).
  - In this case, a simlink preventing the further **Services Installation framework** from the need to mess with version numbers should be created : `/usr/local/lib/x` pointing to `/usr/local/lib/x-version`.
  - This simlink is pretty important. In the second stage, when installing services on eskimo cluster nodes, it is important that setup and installation scripts can be prevented from knowing about the version of the software component. Hence te

important of that simlink.

- The presence of a single `build.sh` script is a requirement. That script is called by eskimo to build the package if it is not yet available (either built or downloaded)
- A helper script `../common/common.sh` cane be linked in the new package build folder using `ln -s ../common/common.sh common.sh`. This common script provides various helpre to build the docker container, save it to the proper location after build, etc.
- The script performing the *in container installation* of the software component X should be called `installX.sh`.
- Software versions to be downloaded and extracted within docker images are coded *once and for all* in the file `../common/common.sh`. This is actually a requirement since most of the time software version for common components such as ElasticSearch or scala are used in several different packages. Defining version numbers of software components to be downloaded and installed in a common place helps to enforce consistency.

An eskimo component package developer should look at pre-packaged services to get inspired and find out how eskimo packaged services are usually implemented.

3.5. Typical build.sh process

3.5.1. Operations performed

The build process implemented as a standard in the `build.sh` script has two different stages:

1. The container preparation and all the container setup operated from outside the container
2. The software installation done from inside the container

As a convention, all dependencies installation is performed from outside the container but then the installation of the software packaged in the container is performed from a script called within the container (script `installX.sh` for package X).

The build process thus typically looks this way:

1. From outside the container:
    - Creation of the container from the base eskimo image (debian based)
    - Installation of the prerequisites (Java JDK, Scala, python, etc.) using `docker exec …`
    - Calling of the software installation script : `docker exec -it … installX.sh`
2. From inside the container:

- Downloading of the software from internet
- Installation in a temporary folder
- Moving of the installation software to `/usr/local/lib/X-Version` or else
- symlinking the software from `/usr/local/lib/X` (without version number)

And that's it.

The package installation is limited to this, all the customizations is done at a later stage, during the **Service Installation** on eskimo cluster noes.

3.6. Look for examples and get inspired

Look at the eskimo pre-packaged component packages development scripts for examples and the way they are built to get inspired for developing your own packages.

3.7. Apache Mesos Building

Bulding Apache Mesos is a different story. There are two different components:

- The **Mesos Master** which runs in a docker container just as every other service within Eskimo. The Mesos Master is responsible for orchestrating the resources requests and manages offering.
- The **Mesos Agent** which runs natively on the Host OS. The Mesos Agent is responsible for understanding the available resources on every node of the Eskimo cluster and answers offers.

The reason why the Mesos Agent runs natively is that it needs to have low level access to the machine to understand its topology and the available resources. To be transparent, it could run as a docker container as well but that comes with some limitations that are not acceptable for a production cluster.

The Mesos Agent is the single and only component running natively within Eskimo (understand, not in a docker container).

As such, Mesos is build natively and mesos packages are tarballs (`tar.gz`) to be extracted on the Host OS on every node.

In this very folder (`packages_dev`) eskimo provides a build framework for Mesos. Currently Mesos is built for three targets : RHEL based (RHEL, CentOS, Fedora, etc.) and Debian based (Debian, Ubuntu, etc.) and SUSE.

3.7.1. Building Mesos

The Eskimo build system for Mesos is based on Vagrant and either VirtualBox or libvirt +

QEMU/kvm. Vagrant takes care of building VMs using either VirtualBox or LibVirt, installs all the required packages for building mesos, builds mesos and creates installable packages.

For instance, toBuild Mesos package for Debian

```
..../packages_dev$ ./build.sh mesos-deb # for building mesos debian
distrib with libvirt
..../packages_dev$ ./build.sh -b mesos-deb # for building mesos debian
distrib with VirtualBox
```

These both commands require vagrant and VirtualBox, respectively libvirt, kvm and the libvirt vagrant provider properly installed.

3.8. Specific and various notes related to individual components shipped with Eskimo

This section presents different important notes related to some specific services shipped with Eskimo building aspects.

3.8.1. Zeppelin building

Zeppelin can be built from a checkout of the latest git repository master or from an official release.

The file `common/common.sh` defines a variable `ZEPPELIN_IS_SNAPSHOT` which, when defined to true, causes the build system to work from git and rebuilt zeppelin from the sources instead of downloading a released package.

```
export ZEPPELIN_IS_SNAPSHOT="false" # set to "true" to build zeppelin from
zeppelin git master
```

Importantly, zeppelin will be build in the folder `/tmp/` of the host machine running eskimo (using a docker container though) which maps `/tmp` to its own `/tmp`).
At least 20 GB of storage space needs to be available in `/tmp` of the machine running eskimo for the build to succeed.

3.9. Setting up a remote packages repository

When running eskimo, software packages - either service docker images or mesos binary packages - can be either built or downloaded from a **remote packages repository**.

Setting up a remote packages repository is extremely simple:

- The software packages need to be downloadable from internet at a specific URL using either HTTP or HTTPS.

- at the same location where packages are downloaded, a meta-data file should be downloadable and present the various available packages.

For instance the following layout should be available from internet or the local network:

- **https://www.niceideas.ch/eskimo/eskimo_packages_versions.json**
- https://www.niceideas.ch/eskimo/docker_template_base-eskimo_0.2_1.tar.gz
- https://www.niceideas.ch/eskimo/docker_template_cerebro_0.8.4_1.tar.gz
- https://www.niceideas.ch/eskimo/docker_template_elasticsearch_6.8.3_1.tar.gz
- https://www.niceideas.ch/eskimo/docker_template_flink_1.9.1_2.tar.gz
- https://www.niceideas.ch/eskimo/docker_template_gdash_0.0.1_1.tar.gz
- https://www.niceideas.ch/eskimo/docker_template_gluster_debian_09_stretch_1.tar.gz
- etc.
- https://www.niceideas.ch/eskimo/eskimo_mesos-debian_1.8.1_1.tar.gz
- https://www.niceideas.ch/eskimo/eskimo_mesos-redhat_1.8.1_1.tar.gz
- etc.

A software package should be named as follows:

- `docker_template_[software]_[software_version]_[eskimo_version].tar.gz` for service docker images
- `eskimo_mesos-[platform]_[software_version]_[eskimo_version].tar.gz` for service mesos packages

The file `eskimo_packages_versions.json` describes the repository of packages and the available packages.

Example eskimo_packages_versions.json

```
{
  "base-eskimo" : {
    "software" : "0.2",
    "distribution" : "1"
  },
  "cerebro": {
    "software": "0.8.4",
    "distribution": "1"
  },
  "elasticsearch" : {
    "software": "6.8.3",
    "distribution": "1"
  },
  "flink" : {
    "software" : "1.9.1",
    "distribution": "1"
  },
  "gdash": {
    "software" : "0.0.1",
    "distribution": "1"
  },
  "gluster": {
    "software" : "debian_09_stretch",
    "distribution": "1"
  },
  ...
  "mesos-redhat": {
    "software": "1.8.1",
    "distribution": "1"
  },
  "mesos-debian": {
    "software": "1.8.1",
    "distribution": "1"
  },
  ...
}
```

It's content should be aligned with the following properties from the configuration file `eskimo.properties`:

- `system.packagesToBuild` giving the set of docker images for packages to be or downloaded
- `system.mesosPackages` giving the name of the mesos packages to built or downloaded

Chapter 4. Services Installation Framework

The Services Installation Framework provides tools and standards to install the packaged docker images containing the target software component on the eskimo cluster nodes.

Eskimo is leveraging on *docker* to run the the services on the cluster nodes and either *SystemD* or *Marathon* to operate them. Marathon is used specifically to provide HA - High Availability - and resilience to UI applications and other single instances services

- An eskimo package has to be a docker image
- An eskimo package has to provide
  - either a **systemd unit configuration file** to enable eskimo to operate the component on the cluster.
  - or a **marathon JSON configuration file** to delegate this marathon

The Eskimo Nodes Configuration process takes care of installing the services defined in the *services.json* configuration file and and copies them over to the nodes where they are intended to run. After the proper installation, eskimo relies either on plain old `systemctl` commands to operate (start / stop / restart / query / etc.) the services or on *marathon*

4.1. Principle

The principle is pretty straightforward:

- Whenever a service `serviceX` is configured on a node, eskimo makes an archive of the folder `services_setup/serviceX`, copies that archive over to the node and extracts it in a subfolder of `/tmp`.
- Then eskimo calls the script `setup.sh` from within that folder. The script `setup.sh` can do whatever it wants but has to respect the following constraint:
- After that `setup.sh` script is properly executed, the service should be
  - either installed on the node along with a systemd system unit file with name `serviceX.service` which is used for commands such as `systemctl start serviceX` to be able to control service *serviceX*,
  - or properly registered in marathon and operated by marathon with ID *serviceX*.

Aside from the above, nothing is enforced and service developers can implement services the way they want.

4.1.1. Gluster share mounts

Services running through System mount gluster shares directly on the host machine through scripts operated with `ExecStartPre` directives of their respective SystemD unit

files. Then the gluster shares they require is mounted in their runtime docker containers.

The same approach is impossible for marathon services since marathon cannot hook any command on the host machine in prior of the docker container start. For this reason, services operated by marathon mount gluster shares directly in their runtime docker containers.

4.1.2. OS System Users creation

OS system users required to execute marathon services are required on every node of the cluster. For this reason, the linux system users to be created on every node are not created in the individual services `setup.sh` scripts. They are created in the eskimo base system installation script `install-eskimo-base-system.sh` in the function `create_common_system_users`.

4.2. Standards and conventions over requirements

There are no requirements when setting up a service on a node aside from the constraints mentioned above. Services developers can set up services on nodes the way then want and no specific requirement is enforced by eskimo.

However, adhering to some conventions eases a lot the maintenance and evolution of these services.
These standard conventions are as follows (illustrated for a service called `serviceX`).

- Services should put their persistent data (to be persisted between two docker container restart) in `/var/lib/serviceX`
- Services should put their log files in `/var/log/serviceX`.
- If the service required a file to track its PID, that file should be stored under `/var/run/serviceX`
- Whenever a service `serviceX` requires a subfolder of `/var/log/serviceX` to be shared among cluster nodes, a script `setupServiceXGlusterSares.sh` should be defined that calls the common helper script (define at eskimo base system installation on every node) `/usr/local/sbin/gluster_mount.sh` in the following way, for instance to define the *flink data* share : `/usr/local/sbin/gluster_mount.sh flink_data /var/lib/flink/data flink`

At the end of the day, it's really plain old Unix standards. The only challenge comes from the use of docker which requires to play with docker mounts a little.
Just look at eskimo pre-packaged services to see examples.

4.3. Typical setup.sh process

4.3.1. Operations performed

The setup process implemented as a standard in the `setup.sh` script has three different stages:

1. The container instantiation from the pre-packaged image performed from outside the container
2. The software component setup performed from inside the container
   - The registration of the service to *SystemD* or *marathon*
3. The software component configuration applied at runtime, i.e. at the time the container starts, re-applied everytime.

The fourth phase is most of the time required to apply configurations depending on environment dynamically at startup time and not statically at setup time.
The goal is to address situations where, for instance, master services are moved to another node. In this case, applying the master setup configuration at service startup time instead of statically enables to simply restart a slave service whenever the master node is moved to another node instead of requiring to entirely re-configure them.

The install and setup process thus typically looks this way:

1. From outside the container:
   - Perform required configurations on host OS (create `/var/lib` subfolder, required system user, etc.)
   - Run docker container that will be used to create the set up image
   - Call in container setup script
2. From inside the container:
   - Create the in container required folders and system user, etc.
   - Adapt configuration files to eskimo context (static configuration only !)
3. At service startup time:
   - Adapt configuration to topology (See Eskimo Topology and dependency management below)
   - Start service

And that's it.

Again, the most essential configuration, the adaptation to the cluster *topology* is not done statically at container setup time but dynamically at service startup time.

4.3.2. Standard and conventions

While nothing is really enforced as a requirement by eskimo (aside of systemd and the name of the `setup.sh` script, there are some standards that should be followed (illustrated for a service named `serviceX`:

- The in container setup script is usually called `inContainerSetupServiceX.sh`
- The script taking care of the dynamic configuration and the starting of the service - the one actually called by systemd upon service startup - is usually called `inContainerStartServiceX.sh`
- The systemd system configuration file is usually limited to stopping and starting the docker container

4.3.3. Look for examples and get inspired

Look at examples and the way the packages provided with eskimo are setup and get inspired for developing your own packages.

4.4. Eskimo services configuration

Creating the service setup folder and writing the `setup.sh` script is unfortunately not sufficient for eskimo to be able to operate the service.
A few additional steps are required, most importantly, defining the new service in the configuration file `services.json`.

4.4.1. Configuration file `services.json`

In order for a service to be understood and operable by eskimo, it needs to be declared in the **services configuration file** `services.json`.

A service declaration in `services.json` for instance for `serviceX` would be defined as follows:

ServiceX declaration in `services.json`

```
"serviceX" : {

  "config": {

    ## [mandatory] giving the column nbr in status table
    "order": [0-X],

    ## [mandatory] whether or not it has to be instaled on every node
    "mandatory": [true,false],

    ## [unique] whether the service is a unique service (singpe instance)
or multiple
    "unique": [true,false],
```

```
    ## [unique] whether the service is managed through marathon (true) or
SystemD (false)
    "marathon": [true,false],

    ## [optional] name of the group to associate it in the status table
    "group" : "{group name}",

    ## [mandatory] name of the service. miust be consistent with service
under
    ## 'service_setup'
    "name" : "{service name},

    ## [mandatory] where to place the service in 'Service Selection
Window'
    "selectionLayout" : {
      "row" : [1-X],
      "col" : [1-X]
    },

    ## memory to allocate to the service
    ## (mesos and neglectable means the service is excluded from the
hardware
    ## memory split computation. Meos services are put in the 'mesos-
agent' memory
    ## share)
    "memory": "[mesos|neglectable|small|medium|large|verylarge]",

    ## [mandatory] The logo to use whenever displaying the service in the
UI is
    ##     required
    ## Use "images/{logo_file_name}" for resources packaged within eskimo
web app
    ## Use "static_images/{logo_file_name}" for resources put in the
eskimo
    ##    distribution folder "static_images"
    ## (static_images is configurable in eskimo.properties with property
    ##    eskimo.externalLogoAndIconFolder)
    "logo" : "[images|static_images]/{logo_file_name}"

    ## [mandatory] The icon to use ine the menu for the service
    ## Use "images/{icon_file_name}" for resources packaged within eskimo
web app
    ## Use "static_images/{icon_file_name}" for resources put in the
eskimo
    ##    distribution folder "static_images"
    ## (static_images is configurable in eskimo.properties with property
    ##    eskimo.externalLogoAndIconFolder)
    "icon" : "[images|static_images]/{icon_file_name}"
  },

  ## [optional] configuration of the serice web console (if anym)
  "ui": {

    ## [optional] (A) either URL template should be configured ...
    "urlTemplate": "http://{NODE_ADDRESS}:{PORT}/",

    ## [optional] (B) .... or proxy configuration in case the service has
```

```
    ## to be proxied by eskimo
    "proxyTargetPort" : {target port},

    ## [mandatory] the time  to wait for the web console to initialize
before
    ## making it available
    "waitTime": {1000 - X},

    ## [mandatory] the name of the menu entry
    "title" : "{menu name}",

    ## [optional] Whether standard rewrite rules need to be applied to
this service
    ## (Standard rewrite rules are documented hereunder)
    ## (default is true)
    "applyStandardProxyReplacements": [true|false],

    ## [optional] List of custom rewrite rules for proxying of web
consoles
    "proxyReplacements" : [

      ## first rewrite rule. As many as required can be declared
      {

        ## [mandatory] Type of rwrite rule. At the moment only PLAIN is
supported
        ## for full text search and replace.
        ## In the future REGEXP type shall be implemented
        "type" : "[PLAIN]",

        ## [optional] a text searched in the URL. this replacement is
applied only
        ## if the text is found in the URL
        "urlPattern" : "{url_pattern}", ## e.g. controllers.js

        ## [mandatory] source text to be replaced
        "source" : "{source_URL}", ## e.g. "/API"

        ## [mandatory] replacement text
        "target" : "{proxied_URL}" ## e.g. "/eskimo/kibana/API"
      }
  },

  ## [optional] array of dependencies that need to be available and
configured
  "dependencies": [
    {

      ## [mandatory] THIS IS THE MOST ESSENTIAL CONFIG :
      ## THE WAY THE MASTER IS IDENTIFIED FOR A SLSAVE SERVICE
      "masterElectionStrategy":
"[NONE|FIRST_NODE|SAME_NODE_OR_RANDOM|RANDOM|RANDOM_NODE_AFTER|SAME_NODE]"

      ## the service relating to this dependency
      "masterService": "{master service name}",

      ## The number of master expected
      "numberOfMasters": [1-x],
```

```
      ## whether that dependency is mandatory or not
      "mandatory": [true,false],
    }
  ]

  ## [optional] array of configuration properties that should be editable
using the Eskim UI
  ## These configuration properties are injected
  "editableConfigurations": [
    {

      ## the name of the configuration file to search for in the software
installation
      ## directory (and sub-folders)
      "filename": "{configuration file name}", ## e.g. "server.properties"

      ## the name of the service installation folder under /usr/local/lib
      ## (eskimo standard installation path)
      "filesystemService": "{folder name}", ## e.g. "kafka"

      ## the type of the property syntax
      ## Currently only "variable" supported
      "propertyType": "variable",

      ## The format of the property definition in the configuration file
      ## Supported formats are:
      ##  - "{name}: {value}" or
      ##  - "{name}={value}" or
      ##  - "{name} = s{value}"
      "propertyFormat": "property format", ## e.g. "{name}={value}"

      ## The prefix to use in the configuration file for comments
      "commentPrefix": "#",

      ## The list of properties to be editable by administrators using the
eskimo UI
      "properties": [
        {

          ## name of the property
          "name": "{property name}", ## e.g. "num.network.threads"

          ## the description to show in the UI
          "comment": "{property description}",

          ## the default value to use if undefined by administrators
          "defaultValue": "{default property value}" ## e.g. "3"
        }
      ]
    }
  ],

  ## [optional] array of custom commands that are made available from the
context menu
  ## on the System Status Page (when clicking on services status
(OK/KO/etc.)
  "commands" : [
```

```
    {

      ## ID of the command. Needs to be a string with only [a-zA-Z_]
      "id" : "{command_id}", ## e.g. "show_log"

      ## Name of the command. This name is displayed in the menu
      "name" : "{command_name}", ## e.g. "Show Logs"

      ## The System command to be called on the node running the service
      "command": "{system_command}", ## e.g. "cat /var/log/ntp/ntp.log"

      ## The font-awesome icon to be displayed in the menu
      "icon": "{fa-icon}" ## e.g. "fa-file"
    }
  ]
}
```

(Bear in mind that since json actually doesn't support such thing as comments, the example above is actually not a valid JSON snippet - comments starting with '##' would need to be removed.)

Everything is pretty straightforward and one should really look at the services pre-packaged within eskimo to get inspiration when designing a new service to be operated by eskimo.

4.4.2. Eskimo Topology and dependency management

As stated above, the most essential configuration property in a *service definition* is the `masterElectionStrategy` of a dependency.
The whole master / slave topology management logic as well as the whole dependencies framework of eskimo relies on it.

4.4.3. Master Election strategy

Let's start by introducing what are the supported values for this `masterElectionStrategy` property:

- `NONE` : This is the simplest case. This enables a service to define as requiring another service without bothering where it should be installed. It just has to be present somewhere on the cluster and the first service doesn't care where.
  It however enforces the presence of that dependency service somewhere and refuses to validate the installation if the dependency is not available somewhere on the eskimo nodes cluster.
- `FIRST_NODE` : This is used to define a simple dependency on another service. In addition, `FIRST_NODE` indicates that the service where it is declared wants to know about at least one node where the dependency service is available.
  That other node should be the *first node* found where that dependency service is available.

First node means that the nodes are processed by their order of declaration. The first node than runs the dependency service will be given as dependency to the declaring service.

- SAME_NODE_OR_RANDOM : This is used to define a simple dependency on another service. In details, SAME_NODE_OR_RANDOM indicates that the first service wants to know about at least one node where the dependency service is available.
  In the case of SAME_NODE_OR_RANDOM, eskimo tries to find the dependency service on the very same node than the one running the declaring service if that dependent service is available on that very same node.
  If no instance of the dependency service is not running on the very same node, then any other random node running the dependency service is used as dependency.
- RANDOM : This is used to define a simple dependency on another service. In details, RANDOM indicates that the first service wants to know about at least one node where the dependency service is available. That other node can be any other node of the cluster where the dependency service is installed.
- RANDOM_NODE_AFTER : This is used to define a simple dependency on another service. In details, RANDOM_NODE_AFTER indicates that the first service wants to know about at least one node where that dependency service is available.
  That other node should be any node of the cluster where the second service is installed yet with a **node number** (internal eskimo node declaration order) greater than the current node where the first service is installed.
  This is useful to define a chain of dependencies where every node instance depends on another node instance in a circular way (pretty nifty for instance for elasticsearch discovery configuration).
- SAME_NODE : This means that the dependency service is expected to be available on the same node than the first service, otherwise eskimo will report an error during service installation.

The best way to understand this is to look at the examples in eskimo pre-packaged services declared in the bundled `services.json`.

For instance:

- Cerebro tries to use the co-located instance of elasticsearch if it is available or any random one otherwise for instance by using the following dependency declaration:

cerebro dependency on elasticsearch

```
  "dependencies": [
    {
      "masterElectionStrategy": "SAME_NODE_OR_RANDOM",
      "masterService": "elasticsearch",
      "numberOfMasters": 1,
      "mandatory": true
    }
  ]
```

- elasticsearch instances on the different nodes search for each other in a round robin fashion by declaring the following dependencies (mandatory false ise used to support single node deployments):

elasticsearch dependency on next elasticsearch instance

```
"dependencies": [
  {
    "masterElectionStrategy": "RANDOM_NODE_AFTER",
    "masterService": "elasticsearch",
    "numberOfMasters": 1,
    "mandatory": false
  }
],
```

- logstash needs both elasticsearch and gluster. In contrary to elasticsearch, gluster is required on every node in a multi-node cluster setup. Hence the following dependencies declaration for gluster:

gluster dependencies definition

```
"dependencies": [
  {
    "masterElectionStrategy": "SAME_NODE_OR_RANDOM",
    "masterService": "elasticsearch",
    "numberOfMasters": 1,
    "mandatory": true
  },
  {
    "masterElectionStrategy": "SAME_NODE",
    "masterService": "gluster",
    "numberOfMasters": 1,
    "mandatory": false
  }
```

- kafka uses zookeeper on the first node (in the order of declaration of nodes in the eskimo cluster) on which zookeeper is available:

kafka dependency on zookeeper

```
"dependencies": [
  {
    "masterElectionStrategy": "FIRST_NODE",
    "masterService": "zookeeper",
    "numberOfMasters": 1,
    "mandatory": true
  }
```

Look at other examples to get inspired.

4.4.4. Memory allocation

Another pretty important property in a service configuration in `services.json` is the

memory consumption property: `memory`.

This setting only applies to native (or SystemD) services, marathon services memory is defined in another way.

Native (SystemD) services memory configuration

The possible values for that property are as follows :

- `neglectable` : the service is not accounted in memory allocation
- `small` : the service gets a single share of memory
- `medium` : the service gets two shares of memory
- `large` : the service gets three shares of memory

The system then works by computing the sum of shares for all nodes and then allocating the available memory on the node to every service by dividing it amongst shares and allocating the corresponding portion of memory to every service.
Of course, the system first removes from the available memory a significant portion to ensure some room for kernel and filesystem cache.

Examples of memory allocation

Let's imagine the following services installed on a cluster node, along with their memory setting:

- **ntp** - neglectable
- **prometheus** - neglectable
- **gluster** - neglectable
- **mesos agent** - **verylarge**
- **elasticsearch** - large
- **logstash** - small
- **kafka** - medium
- **zookeeper** - neglectable

The following table gives various examples in terms of memory allocation for three different total RAM size values on the cluster node running these services.
The different columns gives how much memory is allocated to the different services in the different rows for various size of total RAM.

Node total RAM	8 Gb	16 Gb	20 Gb
ntp	-	-	-
prometheus	-	-	-

Node total RAM	8 Gb	16 Gb	20 Gb
gluster	-	-	-
mesos agent	2500m	5357m	6786m
elasticsearch	1500m	3214m	4071m
logstash	500m	1071m	1357m
kafka	1000m	2143m	2714m
zookeeper	-	-	-
Filesystem cache reserve	1500m	3214m	4071m
OS reserve	1000m	1000m	1000m

Importantly, all marathon services - such as Kibana, Cerebro, Kafka-manager, etc. - as well as all services operated by mesos - such as the spark executors and flink workers - don't get any specific amount of memory assigned here.
Instead, they share the memory available for mesos-agents accross all nodes.
The amount of memory they will request from mesos is explained in the next section.

Marathon services memory configuration

Marathon services define the memory they will request from mesos in the **marathon service configuration file**.

For Instance the file `cerebro.marathon.json` configures the resources that *Cerebro* will request from Mesos as follows:

portion of cerebro.marathon.json

```
{
  "id": "cerebro",
  "cmd": "/usr/local/sbin/inContainerStartService.sh",
  "cpus": 0.1,
  "mem": 300,
  "disk": 100,
  "instances": 1,
  ...
```

So cerebro requests only 300 MB from Mesos.

Another example would be the Zeppelin configuration file:

portion of zeppelin.marathon.json

```
{
  "id": "zeppelin",
  "cmd": "/usr/local/sbin/inContainerStartService.sh",
  "cpus": 0.5,
  "mem": 4500,
  "disk": 800,
  "instances": 1,
  ...
```

So Zeppelin request 4.5 GB of RAM from mesos (which wouldn't be sufficient in a production environment BTW.)

4.4.5. Topology file on cluster nodes

Every time the cluster nodes / services configuration is changed. Eskimo will verify the global services topology and generate for every node of the cluster a "**topology definition file**".

That topology definition file defines all the dependencies and where to find them (using the notion of MASTER) for every service running on every node.

The "topology definition file" can be fond on nodes in `/etc/eskimo_topology.sh`.

4.5. Proxying services web consoles

Many services managed by eskimo have web consoles used to administer them, such as mesos-agents, mesos-master, kafka-manager, etc. Some are even only web consoles used to administer other services or perform Data Science tasks, such as Kibana, Zeppelin or GDash, etc.

Eskimo supports two modes for providing these web consoles in its own UI as presented in configuration above:

1. (A) Configuration of an `urlTemplate` which is used by eskimo to show an iframe displaying directly the web console from the node on which it is installed. **This method is supported for backwards compatibility purpose but it is not recommended**
2. (B) Configuration of a `proxyTargetPort` for full proxying and tunnelling (using SSH) of the whole HTTP flow to the web console using eskimo embedded proxying and tunneling feature. **This is the recommended way** and this is the way used by all eskimo pre-packaged services and web consoles.

Proxying works as explained in the User Guide in the section "SSH Tunnelling".

Proxying is however a little more complicated to set up since eskimo needs to perform a lot of rewriting on the text resources (javascript, html and json) served by the proxied web console to rewrite served URLs to make them pass through the proxy.

Eskimo provides a powerful rewrite engine that one can use to implement the rewrite rules defined in the configuration as presented above.

4.5.1. Rewrite rules

Proxying web consoles HTTP flow means that a lot of the text resources served by the individual target web consoles need to be processed in such a way that absolute URLs are rewritten. This is unfortunately tricky and many different situations can occur, from URL build dynamically in javascript to static resources URLs in CSS files for instance.

An eskimo service developer needs to analyze the application, debug it and understand every pattern that needs to be replaced and define a **rewrite rule** for each of them.

4.5.2. Standard rewrite rules

A set of standard rewrite rules are implemented once and for all by the eskimo HTTP proxy for all services. By default these standard rewrite rules are enabled for a service unless the service config declares `"applyStandardProxyReplacements": false` in which case they are not applied to that specific service.
This is useful when a standard rule is actually harming a specific web console behaviour.

The standard rewrite rules are as follows:

Standard rewrite rules

```
{
  "type" : "PLAIN",
  "source" : "src=\"/",
  "target" : "src=\"/{PREFIX_PATH}/"
},
{
  "type" : "PLAIN",
  "source" : "action=\"/",
  "target" : "action=\"/{PREFIX_PATH}/"
},
{
  "type" : "PLAIN",
  "source" : "href=\"/",
  "target" : "href=\"/{PREFIX_PATH}/"
},
{
  "type" : "PLAIN",
  "source" : "href='/",
  "target" : "href='/{PREFIX_PATH}/"
},
{
  "type" : "PLAIN",
  "source" : "url(\"/",
  "target" : "url(\"/{PREFIX_PATH}/"
},
{
  "type" : "PLAIN",
  "source" : "url('/",
  "target" : "url('/{PREFIX_PATH}/"
},
{
  "type" : "PLAIN",
  "source" : "url(/",
  "target" : "url(/{PREFIX_PATH}/"
},
{
  "type" : "PLAIN",
  "source" : "/api/v1",
  "target" : "/{PREFIX_PATH}/api/v1"
},
{
  "type" : "PLAIN",
  "source" : "\"/static/",
  "target" : "\"/{PREFIX_PATH}/static/"
},
```

4.5.3. Custom rewrite rules

In addition to the standard rewrite rules - that can be used or not by a service web console - an eskimo service developer can define as many custom rewrite rules as he wants in the service configuration in `services.json` as presented above.

Some patterns can be used in both the `source` and `target` strings that will be replaced by

the framework before they are searched, respectively injected, in the text stream:

- `CONTEXT_PATH` will be resolved by the context root at which the eskimo web application is deployed, such as for instance `eskimo`
- `PREFIX_PATH` will be resolved by the specific context path of the service web console context, such as for instance for kibana `{CONTEXT_PATH}/kibana`, e.g. `eskimo/kibana` or `kibana` if no context root is used.

Appendix A: Copyright and License