
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# REMOTE DEPLOYMENT OF INFISPECTOR

**VZDÁLENÉ NASAZENÍ INFISPECTORU**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **MAREK ČÍŽ**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. ONDŘEJ LENGÁL, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2017**

*Bachelor's Thesis Specification/20036/2016/xcizma05*

**Brno University of Technology - Faculty of Information Technology**

Department of Intelligent Systems | Academic year 2016/2017

# Bachelor's Thesis Specification

For: **Číž Marek**
Branch of study: Information Technology
Title: **Remote Deployment of InfiSpector**
Category: Software Engineering

Instructions for project work:

1. Study Infinispan, InfiSpector, and the Red Hat OpenShift technology or any similar suitable solution for applications deployment.
2. Study the Docker technology.
3. Prepare and run an InfiSpector infrastructure in the cloud environment (i.e., instances of Kafka, Druid, Zookeeper). Prepare and run an InfiSpector NodeJS application in the cloud environment and interconnect it with the Druid database.
4. Find out the optimal working solution for the resources that are at your disposal.
5. Discuss results, achievements, and future directions.

Basic references:

- Infinispan web page: http://infinispan.org/
- InfiSpector web page: http://research.redhat.com/projects/infispector/
- OpenShift web page: https://docs.openshift.com/
- Docker web page: https://docs.docker.com/

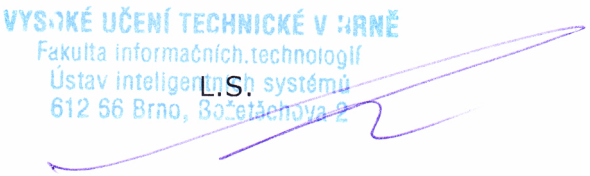
Requirements for the first semester:
Items 1 to 2.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Bachelor's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Lengál Ondřej, Ing., Ph.D.**, DITS FIT BUT
Beginning of work: November 1, 2016
Date of delivery: May 17, 2017




L.S.

Petr Hanáček
*Associate Professor and Head of Department*

# Abstract

The InfiSpector is an application that provides graphical representation of JGroups communication happening between nodes in an Infinispan cluster. We want to separate InfiSpector application from its infrastructure and let users spend time by focusing on using InfiSpector, making conclusions from data provided by InfiSpector, not setting up its working infrastructure. In order to make use and access to InfiSpector easier and quicker, we want to deploy InfiSpector to a cloud service. Cloud service is Internet-based computing that allows users to share computing resources (e.g. servers, storage, applications and services) and data through the Internet. Cloud is also easy to access. As a cloud service was chosen Openshift. Openshift has many tools for easy development, quick deployment, and many tools for running an application. It also allows to have a free user account for everybody.
There is also description of how to configure back-end of the InfiSpector in this thesis. Next step is Openshift description, including its overview and configuration files necessary for InfiSpector deployment, development, and running.



# Abstrakt

InfiSpector je aplikace, která graficky zobrazuje komunikaci mezi uzly Infinispan serverů. Chceme oddělit infrastrukturu InfiSpectoru od aplikace jako takové. Důsledkem oddělení infrastury InfiSpectoru od samotné aplikace je přenesení veškeré starosti a správy InfiSpectoru z uživatele na vzdálenou internetovou službu, kde bude aplikace běžet na vzdáleném serveru. Uživatel se ke vzdálenému serveru již jen připojí, uživateli odpadne nutnost instalovat aplikaci na svém zařízení, a tak se může místo konfigurace InfiSpectoru soustředit na jeho využívání. Vzdálená internetová služba umožňuje jejímu uživateli sdílet výpočetní prostředky, například servery a aplikace. Vzdálená internetová služba umožňuje i využívání externích internetových uložišť pro vzdálené uložení dat. Tím, že je vzdálená internetová služba volně přístupná na internetu, je i dobře přístupná z každého počítače nebo telefonu s přístupem na internet. Pro InfiSpector byla zvolena na základě výzkumu vzdálená internetová služba zvaná Openshift, umožňující spravovat, vyvíjet a provozovat aplikace. Openshift navíc nabízí testovací účet zdarma pro každého vývojáře.
V bakalářské práci se nachází i popis, přehled a nastavení souborů nejen pro části InfiSpectoru zajišťující správu a zpracování dat, ale i pro nasazení, vývoj a provoz InfiSpectoru na vzdálené internetové službě Openshift.



# Keywords

InfiSpector, Infinispan, Docker, Openshift, Zookeeper, Kafka, Druid, Cloud.

# Klíčová slova

InfiSpector, Infinispan, Docker, Openshift, Zookeeper, Kafka, Druid, Cloud.


# Reference


ČÍŽ, Marek. *Remote Deployment of InfiSpector*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Lengál Ondřej.

# Remote Deployment of InfiSpector

## Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Ondřej Lengál, Ph.D. The supplementary information was provided by Mgr. Tomáš Sýkora. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Marek Číž
May 15, 2017


## Acknowledgements

I would like to thank Ing. Ondřej Lengál, Ph.D. for accepting and leading my bachelor's thesis and providing useful advices. Also, special thanks to Mgr. Tomáš Sýkora for all the help and consultations. Without him, this thesis could never be done.

# Contents

# Chapter 1

# Introduction

Nowadays, the trend to use cloud computing is increasing. Cloud computing provides service to users via Internet from cloud computing servers. Cloud servers provide easy, quick, and scalable access to services, computing resources, and applications. Consider the following example. A small company runs an e-commerce website. To run and administer a website, it needs to buy hardware for servers, which are expensive. Enough resources have to be bought to smoothly run the website with a 45 % reserve for occasional workload peaks. The company's analyst says that they need more expensive hardware resources, because on Christmas and Easter, the website will be occupied by more than 350 % of current computing capacity of servers, eventually crashing, which will have the consequence of the company losing paying customers. There are two options how to solve this issue. The first option is to buy more expensive servers, which will be fully utilized only a few days in a year, and the second option is to rent remote cloud computing resources. Cloud resources will scale with actual website needs, the company will pay only for the used resources, and cloud servers are also maintenance-free with their own administrator.

Another common issue in the enterprise environment is explained in following paragraph. Suppose, there are a few high capacity databases in a company. The number of requests from users to the website is not permanent and workload peaks are high, therefore the database is not satisfying all the users requests in an acceptable time. The consequence of a high database response time, and therefore a high website response time, is a loss of paying customers. To get rid of a high response time of databases, there are two prospective solutions. The first solution is to buy more hardware for databases, which are expensive and slow. Another, and more elegant solution, is to get servers with large *random access memory* capacity (referred to as RAM) [16] instead of databases. The RAM servers are cheaper and faster than databases. But how can the RAM servers work as a database? The RAM servers load all data from the databases into their memory and communicate through the application server with the website, which is the same application server as was used in the case of using only databases without the RAM servers. The communication between the application server and the RAM servers is fast, therefore the website can quickly handle all user requests. Databases are expensive and slow, but store data permanently. On the contrary, the RAM servers distributed cache system is relatively cheap and fast, but looses all the data it stores after cutting off an energy supply. The ideal solution is to get a few databases with high data storage capacity and set up a cluster of RAM servers.

RAM servers have to be orchestrated by a tool to do their job correctly. One of these tools is *Infinispan*, which is a tool for managing RAM cache serves, data grids, and data store platform. An Infinispan cluster can be helpful as a distributed cache system or a high-

performance NoSQL data store. New servers can be used to create an Infinispan cluster. In the case of the distributed RAM cache Infinispan system, Infinispan cluster *nodes* are set up on all RAM servers, and Infinispan cluster loads all the data from the database to its RAM. Infinispan cluster consists of Infinispan cluster nodes, each of which is a RAM server. The nodes are interconnected with each other. The data is distributed through nodes and does not have to be consistent. Consider the example of storing the user data. The user's name is stored in `node_1` and the user's address is stored in `node_2`. When the application server asks nodes for the user's data, interconnected nodes send the complex user's data back to the application server which sends the user's data to the user on a website[7].

*InfiSpector* is an official part of *Infinispan* or, more precisely, the *Infinispan management console*, through which a user can monitor communication and data flow between the nodes in Infinispan cluster. Infinispan management console has also a graphical interface. InfiSpector is listening to the communication between Infinispan nodes in a cluster. The main purpose of *InfiSpector* is to graphically represent JGroups communication happening between Infinispan nodes in a cluster, to help users and developers better understand what is happening inside Infinispan during data replication or distribution. In order to make the use and access to InfiSpector easier and quicker, InfiSpector is being deployed to the *cloud* [19].

The aim of this thesis is to explore the way how to deploy InfiSpector to a cloud service. The secondary goal of this thesis is to familiarize Infinispan community, especially developers, with InfiSpector tool, how to set up InfiSpector backend, deploy InfiSpector on a cloud service and make a guide how to deploy complex application on the Openshift cloud service.

This thesis is dealing with general knowledge of the data store platform Infinispan, InfiSpector tool, its architecture and backend configuration including the following: the orchestration tool Zookeeper, the messaging stream Kafka, the NoSQL database Druid, the container platform Docker, the cloud container platform Openshift Online, its advantages and disadvantages, configuration settings, implementation, integration, and optimization with given computational resources of Openshift Online.

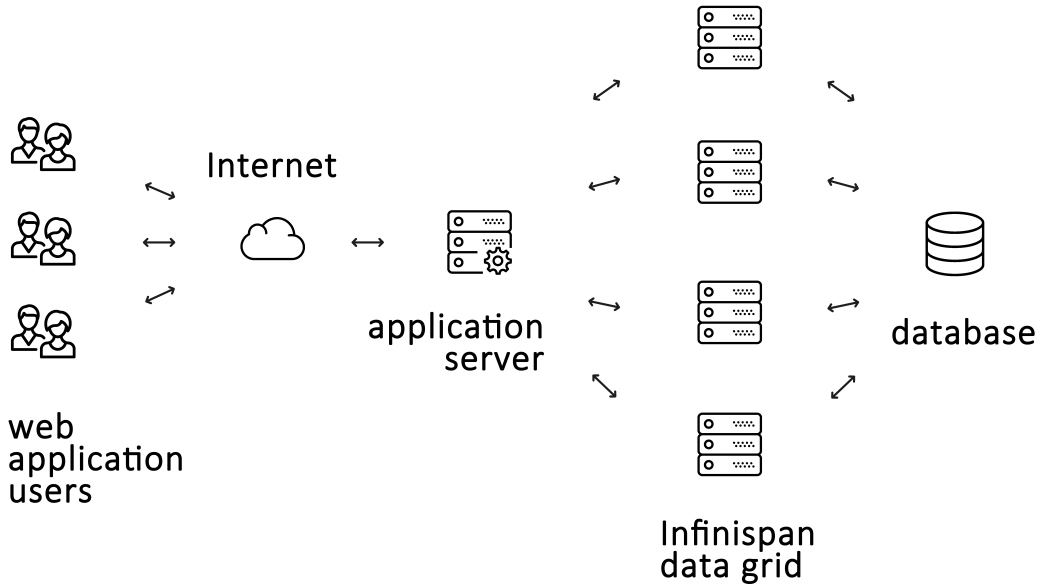



Figure 1.1: Diagram of Infinispan cluster architecture and all system composition [7].

# Chapter 2

# Infinispan

Red Hat Infinispan (referred to as Infinispan) is a JBoss [8] grid and data store platform. It is a scalable, open–source, fully transactional tool written in Java, whitch can be used as a distributed cache system, high-performance NoSQL data store, key–value store.

Why use Infinispan? Databases are expensive and cannot respond to a high amount of requests from application servers in a short period of time. Infinispan can be used to solve these problems. Infinispan cluster data grid can load databases data into its cache memory and communicate directly with application servers. Its cluster nodes communication contain a flow of loaded data and *system* communication. System communication consists of *heartbeat* messages [7], which are checking whether nodes have an *active* status, which means that the current node is *turned on* and communicating with others. Another system message type is checking if nodes are fully functional and if nodes have free space to load or store data from external source [7].

Infinispan distributed cache system is a good solution, when there is an application or a need for fast reading from slow databases. An Infinispan cluster can load all data from database in its *RAM* [16] cache cluster and satisfy a need for a fast communication with database or satisfy a high amount of requests from an application server.

InfiSpector tool is monitoring the communication between Infinispan cluster nodes including both real data flow and system messages. InfiSpector supports data injection both from real-time Infinispan Javascript client and from saved logs. Infinispan communication data are injected into InfiSpector through Kafka messaging stream, which is in-depth described in the Section 3.1.2 or directly to the Druid database.

# Chapter 3

# InfiSpector

InfiSpector is a portmanteau of the words *Infinispan* and *Inspector*. InfiSpector is meant to graphically represent JGroups communication happening between Infinispan nodes in a cluster to help users and especially developers better understand what is happening inside during data replication or distribution. InfiSpector should be able to process big data logs from Infinispan cluster communication. InfiSpector should be separated into its application part, its infrastructure and let users save time by focusing on Infinispan cluster problems using InfiSpector, making conclusions from data provided by it and not spend time setting up InfiSpector working infrastructure instead.

The main objectives of InfiSpector are to provide a convenient web console UI and graphically represent cluster communication between Infinispan nodes [19].

## 3.1 Architecture of InfiSpector

InfiSpector backend stands for the server part of InfiSpector, especially for a *Node.js* server, which is listening on port 3000, and for Druid API, which calls a Druid NoSQL database. The backend is communicating with the frontend via *Representational state transfer* (referred as to *REST*) application programming interface. The *REST* service provides a standardized and uniform communication interface between web services and the Internet [15].

The Infinispan JavaScript client is a library allowing the back-end to communicate with an Infinispan server cluster (written in Java) through JavaScript. The backend is transfering data from Infinispan and makes calls to get cache statistic logs, which InfiSpector graphically represents to a user. The diagram of the InfiSpector architecture is shown in Figure 3.1.

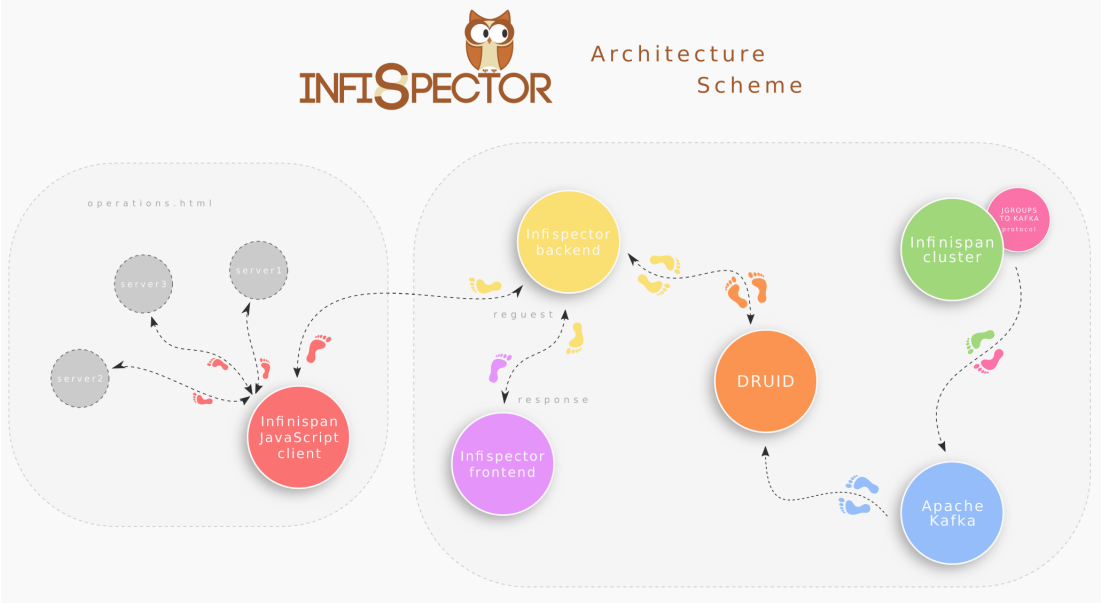



Figure 3.1: Diagram of InfiSpector architecture taken from InfiSpector [20].

### 3.1.1 Zookeeper

Apache Zookeeper is a centralized service for orchestration of synchronization, naming, information, and group services that are used by various applications [2].

In InfiSpector, Zookeeper orchestrates Kafka and Druid into a working cluster. Zookeeper extracts main functionality of various services such as naming, configuration management, synchronization, and group services into a simple, unified interface, so that user does not have to code every single service from scratch. The interface and protocols are easily adjustable for specific needs. Using Zookeeper in projects is simple. Zookeeper allows distributed services to cooperate through a shared *namespace* that is designed like a standard file system. A namespace contains data registers (*Znodes*), which are like directories and files. Znodes differ from files, because they are not meant for storage. Zookeeper data are kept in-memory, therefore accessing the data has a low latency and high performance. These facts lead us to the conclusion that Zookeeper can be used in large, distributed systems and is fast in "read-dominant" workloads such as processing InfiSpector's logs.

Zookeeper is intended to be replicated over multiple hosts, but servers that run Zookeeper have to be connected to each other and keep in a persistent store an in-memory image of the state, transaction logs, and snapshots. One of the servers is the leader. When other Zookeeper servers (called followers) start up, they are connected to the leader. When a new leader is set up, the followers connect to it using the TCP protocol.

### 3.1.2 Kafka

Apache Kafka is a distributed streaming platform capable of recording process streams, storing streams of records, and publishing streams of records. Kafka is connecting applications through real-time data pipes, which allows Kafka to operate with data during its transportation too [1].

Kafka works as a cluster consisting of one or more servers. It stores streams of data records in categories named as *topics* using key-value pairs. These pairs can be superstruc-

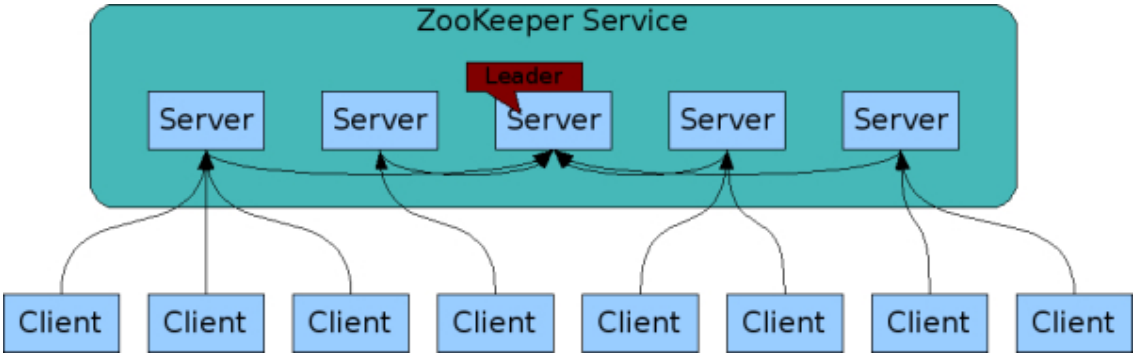



Figure 3.2: A diagram of replicated Zookeeper servers with a leader [2].

tured with *timestamp* as a *key-value-timestamp triad* [1]. Triads are useful when Kafka has to communicate with a NoSQL database such as Druid, which is used in the InfiSpector infrastructure, because Druid is sorting data using timestamps. Each topic in cluster has a log consisting of different partitions that store data using multiple *consumers*, which are subscribed to topics. Partitions are structured, ordered and cannot be changed.

Kafka, same as Zookeeper, connects with servers and clients via *TCP* [13]. The main Kafka client is written in Java, but there are also available Kafka clients written in other languages such as C/C++, PHP, or Python [1].

#### Architecture of Kafka

Kafka consists of 4 essential *API*. Every single one of them can be replicated several times to satisfy requirements given by an application. A diagram of an example of a Kafka cluster is shown in Figure 3.3.

- `Producer API` allows an application to show stream of records from Kafka cluster for a chosen number of *topics*.
- `Consumer API` makes topics and processes available to an application.
- `Connector API` builds and runs reusable consumers or producers that connect Kafka topics to applications.
- `Stream API` allows an application to turn input streams with one or more topics into output streams with one or more topics [1].

### 3.1.3 Druid

SQL servers and databases have been the preferred databases for over 20 years. The increased need to process very high volumes of different data types in a short period of time led developers to develop a type of data store that is capable of storing unstructured data, key-value pairs, graph databases, column family stores, and document databases at scale. NoSQL data stores are good for storing nested data and offer flexibility as not every record needs to store the same properties and do not need to present all data for an object in a single record [14].

Druid is a fast, scalable, open source NoSQL data store designed for analytic queries on event data, capable of working with big data. Druid's trick consists in *data indexing*. The

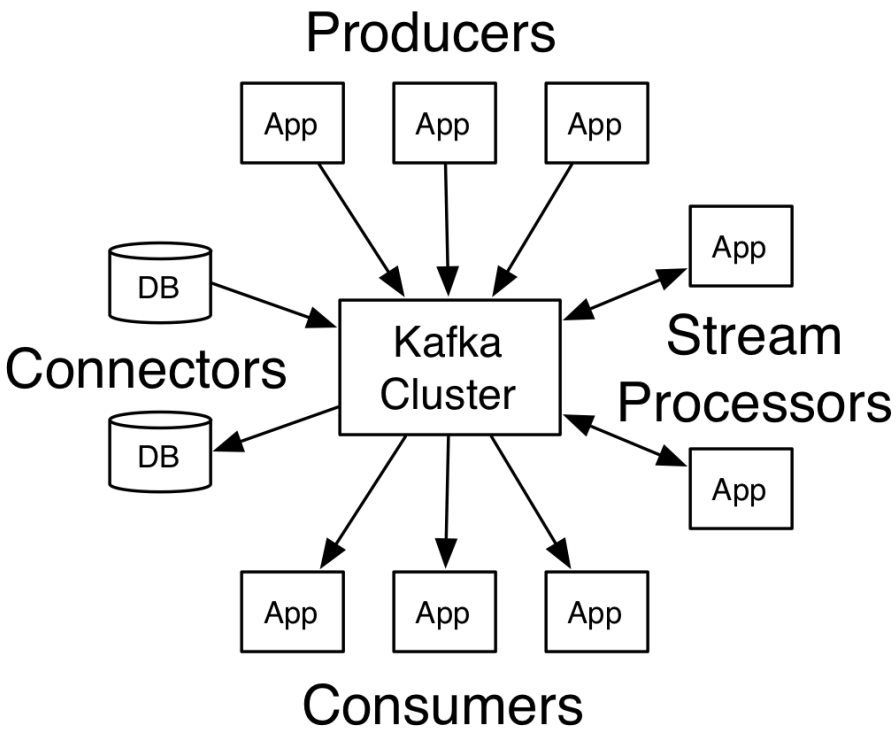



Figure 3.3: Diagram of Kafka cluster [1].

data set is composed of 3 different column components, which are stored separately. The first components are *timestamp columns*. A timestamp is the main component of a NoSQL databases, especially Druid, which can query or process billions of entries in a second, sorted by a timestamp. *Metric columns* are usually numeric values used in aggregations like computing the count, sum, or mean. *Dimension columns* are used in filtering data and contain events represented by strings.

Druid can intake data in *real-time ingestion*, which is described in the roadmap [4] and as *batch ingestion*.

Druid splits data into units known as `segments`. Segments are split implicitly by time and contain compact and sorted data [4].

### Architecture of Druid

A Druid cluster contains these 4 different types of nodes:

- `Broker nodes` serve as a tool for applications and clients to get queried data from a Druid. A Broker node also knows the distribution of segments and files (e.g. files in the JSON format [9]) in a data store, therefore a Broker node can parse given queries, find out sub-results in a data store, and put them together to form the final result, which is shown as an answer to the querying application or client.

- `Historical nodes` interconnect units in a Druid cluster. A cooperation of Historical nodes is based on a distributed computing architecture, where every node is self sufficient, independent, and no nodes share memory or disk storage. This solution

is known as the *Shared nothing architecture* [4]. Historical nodes download segments locally and are able to do queries upon these immutable segments.

- `Coordinator nodes` are organizing segments on *Historical nodes* by telling Historical nodes, when to move segments to load balance, delete loaded segments or load new segments.

- `Real-time nodes` can be run as a standalone unit and used for *real-time processing*, which is all-embracing name of processes like handling segments off to historical nodes, creating segments (indexing the data), and ingesting data. In a Druid cluster, real-time processing can be done by using real-time nodes or using an indexing service. Real-time nodes use Zookeeper to monitor metadata storage [4].

### 3.1.4 InfiSpector Node.js application

InfiSpector application engine is the *Node.js Javascript server*, which is designed to build scalable applications running on a network. Node.js server uses the *event loop* instead of a classic library. The event loop allows Node.js server to perform non-blocking input and output operations, which do not end by the *deadlock*. Deadlock is a state of a system, when all subsystems are locked and waiting for some other subsystem to release a lock, which is never going to happen. Operations from a different processes are added to the event loop poll queue, where operations are waiting to be eventually executed [10].

HTTP [5] is the main communication protocol in a Node.js servers. InfiSpector front–end is communicating with a Node.js server via HTTP Rest calls [15] on the port 8080, which offers web services.

# Chapter 4

# Docker

Docker is an open platform for developing, shipping, and running applications. It enables to separate applications from the infrastructure, so that software can be delivered quickly. By taking an advantage of Docker's methodologies for shipping, testing, and deploying code quickly, a user can significantly reduce the delay between writing code and running the code in a production.

Docker provides the ability to package and run an application in a loosely isolated environment called a *container*. The isolation and security allow to run many containers simultaneously on a given host. Because of the lightweight nature of containers, a user can run more containers on a given hardware combination than using virtual machines [3].

## 4.1 Docker engine

The Docker engine is a client-server application. One of Docker engine main components is the *Docker engine server*. The Docker engine server is a type of long-running program called the *daemon process*. Another main component is the *REST API* [15], which specifies interfaces that programs can use to talk to Docker daemon and instruct Docker daemon what to do. The last component is a *Command line interface client* (CLI). Docker daemon is communicating with the CLI through REST API, using direct commands, or scripts. The CLI creates, manages, and deletes images, containers, networks, and data volumes [3].

## 4.2 Docker architecture

Docker is based on the client-server architecture. A Docker daemon is active on a host machine and communicates with a user through client using the REST API [15]. Both the client and the daemon can be run on the same machine or from a cloud. Docker composed of: *containers*, *images*, which serve as read-only templates for containers, *registries*, which are libraries of images and *services*, which allows Docker nodes to cooperate and work together.

### 4.2.1 Containers

Docker *containers* can be understood as lightweight virtual machines that share the resources of a Docker host. Docker containers are a secured and isolated platform and every single container is built from an *image*. Containers can be controlled from Docker API using CLI commands. For example:

```
$ docker start | run | delete | move | stop | rm
```

Containers can be also *inspected* using the `docker inspect` command in the CLI. The output in the JSON format [3].

#### Running a container

To run anything in Docker, its environment has to be set up. Firstly, an *installation of Docker Engine* has to be done. A good guide for various systems is on the official Docker documentation website[1].

Containers can be run in a data center, a cloud, a local host, or on virtual machines. Containers are portable, lightweight, and fast, and multiple containers can run simultaneously on the same physical computing machine. A container is started using the API or the CLI.

After a user runs a container, the Docker engine executes the following steps in given order: *Pulls the image.* Docker image open database is called *Docker hub*[2]. In Docker hub can be found a long list of both official and adjusted or personalized images. A Docker image can be cloned from Docker hub by executing the *pull* command in the CLI, unless the image already exists on the local host. For example: pulling Zookeeper personalized for InfiSpector:

```
$ docker pull mciz/zookeeper-docker-InfiSpector
```

Then Docker engine creates a new container, mounts a read-write layer, network, IP address, executes `/bin/bash` executables command.

## 4.3 Dockerfile

When there is a need of creating a new image or updating an existing one, user can update images to Docker hub by the following CLI command:

```
$ docker commit
```

and build the image in a localhost environment by CLI command:

```
$ docker build
```

A User can also use an automated build using a *Dockerfile*, where all needed instructions and commands for assembling an image automatically are stored. For example, building Docker image from Dockerfile, which is stored in the current directory (“ . ” means that Dockerfile is in current directory). Docker daemon is running all inner actions of building the image [3]:

```
$ docker build .
```

InfiSpector's Dockerfiles are available on Docker hub[3].

Docker hub enables easy update of Dockerfiles. It also supports *link to a hosted repository service*, which links Docker hub repositories to *GitHub*[4]. GitHub is a web cloud service that allows developers who use the *git* tool [6] to store their projects. When a git repository is updated on GitHub, the update action triggers another action in Docker hub, which

[1] https://docs.docker.com/engine/installation
[2] https://hub.docker.com/
[3] https://hub.docker.com/u/mciz/
[4] https://github.com/

updates its linked repository, and therefore stored Dockerfile in Docker hub. This service is not necessary, but highly recommended, because it is useful and saves a lot of time.

### 4.3.1 Instructions in Dockerfile

This is not an enumeration of Dockerfile instructions (in Dockerfile, an instruction has the meaning of a CLI command), but only of those instructions that are being used in InfiSpector's Dockerfiles. Instructions in a Dockerfile are not case-sensitive, but the convention is to type them in *uppercase*.

The list of the used instructions is the following:

- `FROM` has to be the first instruction in Dockerfile. FROM determines the base image for a new personalized image.
- `MAINTAINER` sets the author of an image.
- `USER` sets user name for RUN and CMD instructions.
- `ENV` sets environment variable.
- `EXPOSE` informs Docker on which ports is a container listening.
- `RUN` will execute any commands and commit the results.
- `WORKDIR` sets the working directory for the COPY, ADD, RUN, and CMD instructions.
- `COPY <src> <dest>` copies files from `<src>` path and paste them to `<dest>` path. Regular expressions for choosing files and paths can be used too.
- `VOLUME` sets a mount point. The RUN instruction initializes the mount point with any existing data within the base image.
- `CMD` provides defaults for an executing container.

# Chapter 5

# Openshift

Red Hat's Openshift Online (referred as to Openshift) is a scalable cloud platform adapted for application hosting and application development, enabling to manage applications from a command line or a web client. There are various versions of Openshift. The newest stable version is 1.2, but special branch of Openshift (NextGen) for developers with limited access is being used.

## 5.1 Openshift account

Anybody can try Openshift Online platform for only one month trial version, nevertheless Red Hat granted special long term access for developers, allowing them to work with the newest Openshift features. Openshift NextGen is still in development, and can be considered a *bleeding edge technology*, which means that unlike stable versions, a lot of features may not be working correctly. The next consequence of a bleeding edge technology project is also its deprecated documentation. A big improvement recently done by Red Hat developers is Openshift web client console, which allows users to manage Openshift cluster, applications, and allows users to have the overview over all units in Openshift cluster [11]. Openshift NextGen has lots of bugs, but also a great potential to become a favourite, high-performance, worldwide platform, which can compete with the tuned giants like Amazon Web Services or Microsoft Azure.

## 5.2 Architecture of Openshift

Openshift is a layered system using Docker container images with the same purpose, as is Docker using Docker container images inside its own environment. A thorough description of Docker can be found in Section 4.2.1.

The Kubernetes tool manages the whole Openshift cluster, and containers in multiple hosts. The Kubernetes and Docker have an alike architecture and principles. Openshift Online has many new features. Openshift can do an application management at a scale, handling images, source code, and deployment.

Openshift is based on small units cooperating together, and running the Kubernetes cluster, where are stored objects, data, and relations. The objects are communicating via the REST APIs [15], using controllers for reading and managing the APIs. The whole Openshift principle is similar to Docker engine. User commands the *client*, which sends

REST API calls to the controllers, which process requests, and synchronize the system to satisfy user's requirements [11].

### 5.2.1 Pod

A pod is an important part of Openshift similar to a Docker container. It is the smallest computing unit that can be deployed and managed. A pod can contain one ore more Docker containers. For a full use of Openshift scalability, every pod should have only one Docker container. Each pod has its own IP address, and therefore access to all ports provided by Openshift system.

The major difference between Docker containers and pods is that Pods have a *lifecycle*, which means that pods are set up with a running Docker container, and after stopping or deleting the Docker container, all associated pods are cancelled. With a new run of a Docker container, all pods are recreated [11].

### 5.2.2 Service

`Services` are the Kubernetes tool to balance an internal load. Services are the REST [15] objects. Services are used to allow pods to cooperate and communicate with each other. Services are used also for setting up the communication to the cluster's outside "world". They allocate IP addresses and ports in Openshift both internally and externally. When a pod is cancelled after fulfilling its purpose, and in the future is recreated, services can be set to allocate the same port and IP address, to recreate the pod with the same settings [11].

### 5.2.3 Route

Routes are the way how to expose a service by giving it an externally reachable hostname like `www.InfiSpector.io`. Routes behave like external clients that are mapping services over Internet protocols (HTTP, HTTPS, TLS, WebSockets) [5].

### 5.2.4 Image streams

An *image stream* can be used in an application deployment to automatically create a new version of the image stored in Openshift registry. An image stream's purpose and functionality is similar to the *Docker image repository* [3]. An image stream is a virtual environment for the following: Docker images pushed to Openshift registry, Docker images from external registries, or other image streams. When an application obtains an alert that there is a change in the image stream, application triggers a new deployment. At start, a new version of an application deployment image is created, then the deployment is built, and the new version of the application deployment is automatically assigned to the application [11].

### 5.2.5 Templates

A *template* can be used to create every permitted object in Openshift. An object can be parameterized within a project.

There are two templates in InfiSpector backend for Druid and Kafka, written and defined in the *YAML* format, which is a human-readable data serialization language [18].

The following list explains the basic structures in a template:

- `Description` can be used for searching for a template through Openshift web console. Description also shows information about the template, template tags, and application related programming language.

- `Labels` are included into each object that is created from a template. Labels are assigned in the object metadata. Labels are an extension of a *tag*. For example, if pods are tagged with labels, a service can easily find and interconnect all pods with the same label.

- `Parameters` allow both users and templates set environmental variables and configurations values of Openshift objects. Values are updated whenever the parameter is referenced in Openshift, which is being done directly as a string value in the form *${name of the parameter}* [11]. Parameters can be also used for gathering user's passwords in registrations, because parameter's values can be assigned also by regular expressions [12]. There is the example of obtaining the value of user's password using the regular expression in the *YAML* form:

```
parameters:
  - name: PASSWORD
    description: "The random user password"
    generate: expression
    from: "[a-zA-Z0-9]{12}"
```

### 5.2.6 Deployments

Configuration of a deployment sets required state of an application. In InfiSpector case, there are deployments for Zookeeper, Kafka, Druid, and for Node.js InfiSpector application.

The deployment system provides a configuration template for running applications with replication scaling and automation in applying application updates. When the configuration for a deployment is created, a replication controller is set up too by Openshift. A replication controller is a tool for orchestrating a whole deployment and a replication of a new application, or replicating already deployed application due to scaling purposes. In every update of a deployment configuration, the old replication controller is terminated, and right after termination of the old controller, the new controller is created from a pod template.

### 5.2.7 Images

An image is a binary that includes all of the requirements necessary for running a single container in a pod, as well as metadata describing image's requirements. *Containers in pods* are also limited by given hardware resources, which can be increased or decreased if there is a need to do so. During application development, a single image name can refer to many versions of the same image over time. To distinguish these versions every image version has its own unique indentifier, and therefore older versions of the image can be reused.

Openshift images are in their substance Docker images identified by tags and uploaded to Openshift registry. A cluster of images tagged by the same label is called an *image stream*.

### 5.2.8 Project

*Openshift Project* (referred to as Project) provides the environment for an application development, deployment, and management. In *Openshift Online DevPreview* (referred to as Openshift DevPreview) account, where InfiSpector is deployed, it is permitted to have only one project at given time. Having more projects simultaneously is available in the paid *Openshift Enterprise* version. A project is the main tool to access resources through *web graphical client* or terminal client (called OC). Every Project detaches the user's content. A Project is described by a unique identifier, which is the *project name*. An optional attribute visible both in the web console and the *OC* is *description*, which provides more detailed information about project. A Project sets its own set of:

- `Objects` are functional parts of Openshift and can be declared with *templates*, e.g. *services*, *routes*, *pods*, *containers*, *deployments*, or *images*.
- `Service accounts` act automatically with the access to the project objects.
- `Policies` are rules and restrictions for the users over objects.
- `Constraints` are limits for each object. *Constraints* can be set in special configuration file, which is described in more detail in Section 8.

A list of projects can be displayed in the *OC* by the command:

```
$ oc get projects
```

Present project is displayed by command:

```
$ oc get project
```

Present project can be accessed by command:

```
$ oc project <name of project>
```

## 5.3 Client

Openshift client serves as a tool for handling and managing Openshift a command line.

### 5.3.1 Openshift versus Amazon Web Services

There are a lot of cloud computing services providers, for example Amazon or Red Hat. The Amazon Web Services platform(referred to as AWS) is one of the best cloud platforms for developing and running applications and projects. AWS is a sophisticated platform allowing to deploy and develop applications based on almost every commonly used technology. It has a good support, a large user base and a lot of templates for quick setups.

InfiSpector was firstly meant to be deployed on AWS, but the free computational resources offered to students were unsatisfactory. InfiSpector deployment to AWS was not possible under given circumstances.

After the cloud services providers research, Red Hat Openshift Online NextGen (referred to as Openshift) was chosen. Red Hat granted a free one year account to Openshift Preview, which is developer version of Openshift Online Enterprise. That was the main reason for picking Openshift over other platforms. Special one year free developer account is not

ordinary offer on the Red Hat webpage[1]. It was granted because of InfiSpector is an official project of Red Hat laboratory. Common length of free trial version for Openshift is one month. Your account is being deleted after one month of trial version, but user can apply for another one month trial account. Openshift is a fast growing, great platform with big further potential, which provides tools for developing, deployment and running applications, but right now it is in a developing phase and its documentation is not actual and partially deprecated. There are important actions, which are not described in documentation or documentation is outdated or invalid. For example, in documentation is a possibility to deploy application from Docker hub using only Dockerfile and configuration files, which cannot be done. Application has to be build locally by Docker, tagged in Openshift client and pushed to Openshift registry to upload an image.

### 5.3.2 Installing Openshift client

In order to run Openshift client tool on a user's local machine in a terminal, **three prerequisities have to be satisfied**. First of the compulsory requirements is to have established GitHub account[2], have Git[3] set up, and authenticate to the Openshift account with the GitHub account. Secondly, the user has to be able to access a running instance of Openshift Online. Lastly, the instance has to be pre-configured by a cluster administrator or directly from the image stream or the template.

After fulfilling the prerequisities, client installation differs on each operation system, but the core of the installation is similar. The *PATH* to an unpacked archive with Openshift client (referred to as OC) binaries has to be added to operation *system PATHS*. For Linux, downloaded *tar.gz* archive unpack to the folder and move OC binary on *PATH*. Archive can be unpacked by command from Linux terminal:

```
$ tar -xf <packed_archive>
```

To verify correctness of latest steps, run in terminal:

```
$ echo $PATH
```

First command in *OC* should be:

```
$ oc login
```

Which sets up OC and establishes a session to an Openshift server with credencials and configuration files [11]. Configuration files can be accessed by OC command:

```
$ oc config view
```

### 5.3.3 Useful *OC* commands

Here is a list of the useful *OC commands*. The commands are divided into 7 groups: *basic, build and deploy, application management, troubleshooting and debugging, advanced, settings, other* [11]. This is a listing of a selection of useful CLI commands. The rest of the CLI commands is in Section B.1.

- `$ oc --help`

[1] Red Hat Openshift website: https://www.openshift.com
[2] https://github.com
[3] https://git-scm.com/

Displays useful commands in terminal.

- `$ oc new-project <project_name>`

  Sets up a new project with given name.

- `$ oc  new-app <app_name>`

  Sets up a new application with given name.

- `$ oc status`

  Shows an overview of the current project.

- `$ oc describe <object>`

  This command shows details of a specific resource or group of resources. For example, internal substeps in a pod deployment can be monitored.

- `$ oc edit <object>`

  Edits a object on the server. It is used mainly for quick small fixes.

# Chapter 6

# InfiSpector infrastructure in Openshift

This chapter describes implementation of the InfiSpector architecture in Openshift, especially its architecture and configuration files of the InfiSpector backend.

## 6.1 InfiSpector architecture in Openshift

The core an InfiSpector application deployed in Openshift Online are 3 pods, one for each Kafka, Druid, and InfiSpector. There is no solitary Zookeeper pod because the Kafka distribution contains its own Zookeeper server, and due to optimization both Kafka and Zookeeper containers run in a single Kafka pod. More information about optimizations of InfiSpector deployment can be found in Chapter 8. The whole InfiSpector architecture is shown in Figure 6.1.

The `Kafka pod` is generated from Kafka, as described in Section 7.6. Kafka pod contains:

- *Kafka container*, where Kafka application is running.
- *Zookeeper container*, where Zookeeper application is running.
- *Kafka service* communicating with the port 9092, which exposes Kafka container to an internal access within pods or containers in Openshift.
- *Kafka service* communicating with the port 2181, which exposes Zookeeper container to an internal access within pods or containers in Openshift.
- *Kafka route* exposing Kafka pod via the port 9092 to the Infinispan data communication *outside* Openshift.

Another pod is the `Druid pod`, which is generated from Druid template. Druid pod contains:

- *Druid container*, where Druid database is running.
- *Druid service* communicating with the port 8084, which exposes Druid container to an internal access within pods or containers in Openshift.

- *InfiSpector_Druid_sample_data.json file*, which is only for testing purposes, and contains sample data from Kafka in the JSON format.

The last pod is the `InfiSpector pod`, which is generated from Openshift built–in Node.js template and InfiSpector source code from InfiSpector GitHub repository[1]. InfiSpector pod contains:

- *InfiSpector container*, where InfiSpector application is running.
- *InfiSpector service* communicating with the port 8080, which exposes InfiSpector container to an internal access within pods or containers in Openshift.
- *InfiSpector route* exposing InfiSpector pod by the port 8080, to InfiSpector web page *outside* Openshift.

[1] https://github.com/infinispan/infispector

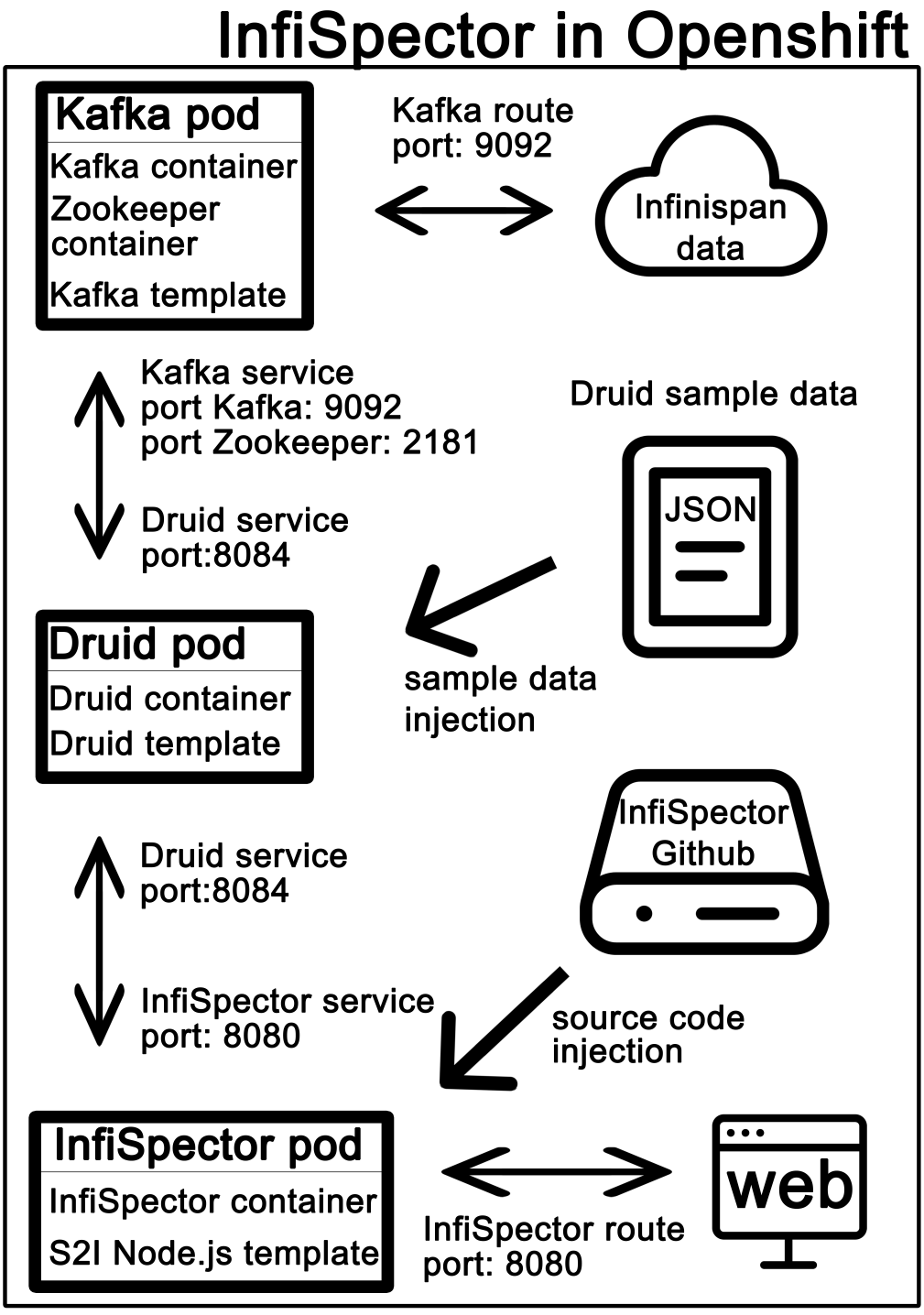



Figure 6.1: Diagram of InfiSpector architecture in Openshift

# Chapter 7

# InfiSpector deployment in Openshift

This chapter is the core of this Bachelor's thesis and describes implementation of InfiSpector backend configuration files including its Dockerfiles, templates, services, and routes.

**The *Openshift Online NextGen*, which is the version of Openshift, where InfiSpector is deployed, is still in development, and is being developed quickly. Therefore documentation, guides, quickstarts, and tools can differ in a near future.**

Deploying an application using templates is the recommended way, how to deploy an application in Openshift. To use a template, there is a need to have an uploaded *image* of the application or the parts of the application in Openshift registry.

There are two options how get an *image* ready to be processed by the template. The first option is to deploy an application by the *Source to image* (S2I) method. S2I comes handy when the user is an application developer and has access to the source code of the application. To deploy an application by S2I, there is a need to have the source code of the application. After processing the source code using S2I, the output of S2I is the wanted image, which can be further used by the application template to make a deployment. S2I also stores the image in Openshift registry automatically. InfiSpector uses the S2I to create the Node.js application. The second option is to use Dockerfiles of Kafka, Druid, and Zookeeper directly, but this method is deprecated and no longer supported. Openshift is still in development, which is the reason why Openshift documentation is inconsistent. In November 2016, the use of Dockerfiles directly was a quite easy solution. There was a simple guide in Openshift documentation how to deploy an application directly by Dockerfiles, so the Dockerfiles method was chosen as a solution in InfiSpector deployment. Despite the fact that direct deployment by using only Dockerfiles was still in Openshift documentation, it was no longer supported by Openshift. Nowadays, the application deployment using Dockerfiles is supported by Openshift with the condition that Openshift registry is being used in the process.

The complexity of the application backend deployment process depends on many factors, for example, on size of the application, the number of tools being used in the development process and the interconnection of the tools, how many requirements for scalability, optimization and speed the application has.

For a good cooperation of Zookeeper, Kafka, Druid, and InfiSpector Node.js application, there is a need for a deeper configuration, because applications downloaded by Dockerfiles

are from official repositories, and therefore blank without any configuration. Configuration files for Zookeeper, Kafka, and Druid are enclosed in Dockerfiles. InfiSpector Node.js application is not being built from Dockerfiles, but directly from a source code in InfiSpector GitHub repository[1], so there is no need to make the configuration file for InfiSpector Node.js application.

## 7.1 Zookeeper configuration

In InfiSpector, Zookeeper version 3.4.6 is being used in InfiSpector, which is accessible from official Apache Zookeeper distribution page[2].

First, for using configuration file, copy `zoo.cfg` to `zoo_sample.cfg` that is situated in folder `zookeeper-version/conf` and these parameters has been set for InfiSpector's needs:

- `tickTime` is used to do heartbeats and the minimum session timeout will be twice the tickTime. The tickTime unit is in milliseconds. The tickTime is set to 2000 value.
- `initLimit` is number of ticks seen as a maximal time limit when a follower is connected to a leader during initialization. It is set to 10, which corresponds to 20 seconds (maximal time limit = initLimit * tickTime).
- `syncLimit` is the number of ticks that can pass between sending a request from a follower to a leader and getting an acknowledgement. SyncLimit can be represented in time too. The pattern for syncLimit is similar to initLimit (syncLimit time = syncLimit * tickTime). SyncLimit is set to 5, which corresponds to 10 seconds.
- `dataDir` is a path to the folder to store in-memory data, transaction logs and database updates. The dataDir folder path value is set to `/tmp/zookeeper`.
- `clientPort` is the port for client connections on the Internet. In InfiSpector's Zookeeper the clientPort is set to 2181.
- `maxClientCnxns` is not obligatory. The MaxClinetCnxns stands for maximum number of client connections. The MaxClinetCnxns is not set in InfiSpector so far.

The Zookeeper single server with the configuration set in `zoo_sample.cfg` can be run from terminal using the command:

```
$ bash zookeeper-3.4.6/bin/zkServer.sh start
```

## 7.2 Kafka configuration

First, Zookeeper server should be started, because Kafka needs Zookeeper for its proper running. There are 2 possible ways how to start Zookeeper server. One is to start the Zookeeper server provided by Kafka: `kafka-version/bin/zookeeper-server-start.sh config/zookeeper.properties`, the second one is to start the Zookeeper server from Zookeeper itself. We are using Java client with *Scala* [17] version 2.11 and Kafka version 0.8.2.2, which is accessible from the official Apache Kafka download page mirror[3].

[1] https://github.com/infinispan/infispector
[2] http://www-us.apache.org/dist/zookeeper
[3] http://mirror.nexcess.net/apache/kafka/0.8.2.2/

In configuration files, which can be found in `config folder - kafka_2.11-0.8.2.2/config`, are 7 configuration files with the `.properties` suffix. For InfiSpector purposes, there is a need to adjust the following files with the `.properties` suffix. Other files can be set up with default values.

- `zookeeper.properties` is a configuration file for the definition of the interconnection Kafka with Zookeeper. The port for *TCP* protocol communication has to be set using the `clientPort` variable, the set value 2181 is the same as was set in Zookeeper *zoo_sample.cfg* file, and it is the value of 2181. The variable `dataDir` has to be assigned with the same value as was set in Zookeeper too, and it is the value of `/tmp/zookeeper`. The variable `maxClientCnxns` can be set as value 0.

- `server.properties` is dictating basic settings of Kafka severs. `Server.properties` is divided into 6 parts: *Server Basics, Socket Server Settings, Log Basics, Log Flush Policy, Log Retention Policy and Zookeeper*. For InfiSpector's needs almost all variables are set to the default values, except the following variables: In section *Socket Server Settings* the variable *port* has to be set for future communication via *TCP*. The communication will take place between a NoSQL database Druid and with Infinispan servers. The `port variable` has to be set to the same value 9092 as in the file `servis.properties`. In the section Zookeeper, the variable `zookeeper.connect` has to be set with the value `localhost:2181`. The localhost stands for Internet protocol (called IP) address with value of 127.0.0.1 (referred to as localhost) and the port 2181 is the port number, which has to be identical with the value of parameter in `zoo_sample.cfg` Zookeeper configuration file, and therefore identical in *zookeeper.properties* Kafka configuration file.

- `producer.properties` is coordinating streams of records. In the *Producer Basics* part set value of the variable `metadata.broker.list` to `localhost:9092`. The port value 9092 stands for the port for TCP connection, its value has to be the same as the value of port in *servis.properties*. Variable `metadata.broker.list` is setting a list of brokers, which are collecting data from the whole cluster in format `host1:port1, host2:port2, etc.`. If a big data are processed, compression for all generated data by Kafka can be set as *none* or *gzip* [1] in the variable `compression.codec`.

- `consumer.properties` file is looking after settings of communication with Zookeeper, and the applications connected to Kafka. The variable `zookeeper.connect` has to be set with pair: Internet protocol (IP) and the port number for TCP protocol connection. The value of the IP address can be set to *localhost*, and its port value has to be set as 2181, which is the same port value as was set in Zookeeper configuration file.

### Druid configuration

In InfiSpector is being used only one Real-time standalone node, not the whole Druid cluster.

Specification file of configuration is in InfiSpector `InfiSpectorDruid.spec` and path to the file is: `InfiSpector/kafka_druid_infrasctructure/InfiSpectorDruid.spec`. The `InfiSpectorDruid.spec` is written in a JSON format. The `InfiSpectorDruid.spec` has many sections. In the *ioConfig* section are set ports, addresses, other parameters for data

transfer, and connection to Zookeeper and Kafka. In the *tuningConfig* section are set types of a node (Real-time), size of memory, and rejection policy.

## 7.3 Docker operations

To make InfiSpector application backend deployment possible, Kafka, Druid, and Zookeeper Dockerfiles, Docker and Openshift registry has to be used in the process.

### 7.3.1 Zookeeper Dockerfile

Zookeeper Dockerfile sets the local environment that Docker daemon needs for creating Zookeeper image and afterwards the container. InfiSpector's Dockerfile for Zookeeper can be found at Docker hub[4].

- `FROM` is set as the source base image `jboss/base-jdk` (Java version 7), because it contains the image used as a base for JBoss community images that require the Java Development Kit, which Zookeeper requires too.
  Instruction: `FROM jboss/base-jdk:7`

- `USER` is set as a root user at the beginning. The root user is required for updating the base image, installing utilities, and changing access permissions of files and folders.
  Instruction: `USER root`
  After installing and changing access permissions, a user have to be switched to the non-root user `jboss`, because Openshift prohibits Dockerfiles, which has to execute the CMD instruction as the root user.
  Instruction: `USER jboss`
  There were issues with the Openshift non-root setting, because this issue is not properly described in the Openshift documentation[5]. There is only one working solution, which was advised by Openshift community[6].

- `ENV` sets the environmental variable `ZOOKEEPER_VERSION` to `3.4.6` value, which is used in InfiSpector. The environmental variables, which are set in Docker, are included and used in Openshift automatically as well.
  Instruction: `ENV ZOOKEEPER_VERSION 3.4.6`

- `EXPOSE` sets communication on the ports 2181 for Zookeeper, 2888 for Kafka, 3888 for Druid cluster, and 8084 for the real-time standalone Druid node.
  Instruction: `EXPOSE 2181 2888 3888 8084`

- `RUN` occurs three times in the Zookeper's Dockerfile. The first occurrence is executing chain of commands, which downloads and prepares Zookeeper, the updates base image system, and copies Zookeeper and its configuration file to auxiliary folders.
  The second RUN command changes the access permissions to 755, which is the Unix system octal encoding for granted permission to read, write, and execute files. Then bash runs Zookeeper's `start.sh` bash script, which starts the Zookeeper server in a foreground.

[4] https://hub.docker.com/r/mciz/zookeeper-docker-infispector/
[5] https://www.openshift.com/
[6] https://stackoverflow.com/questions/41148127/openshift-online-zookeeper-from-dockerfile-pod-crash-loop-back-off

Instruction: `RUN chmod 755 /opt/zookeeper/bin/run.sh`
The last RUN command changes permission for the user `jboss` to regular group read and write, because **Openshift prohibits Dockerfiles, which has to execute the CMD instruction as the root user.**.
Instruction: `RUN chown -R jboss:0 /opt/zookeeper && chmod -R g+rw /opt/zookeeper`

- `WORKDIR` sets current working directory to `/opt/zookeeper`.
  Instruction: `WORKDIR /opt/zookeeper`

- `COPY` instruction copies bash script `run.sh`, which runs started zookeeper server, to `bin` folder.
  Instruction: `run.sh ./bin/`

- `VOLUME` sets the mount point to the folder `/opt/zookeeper/conf`.
  Instruction: `VOLUME ["/opt/zookeeper/conf"]`

- `CMD` is executing the bash script `run.sh`, which runs Zookeeper's server in foreground.
  Instruction: `CMD ["/opt/zookeeper/bin/run.sh"]`

### 7.3.2 Kafka Dockerfile

Kafka Dockerfile sets the local environment that Docker daemon needs for creating Kafka image and afterwards the container. InfiSpector's Dockerfile for Kafka can be found at Docker hub [7].

- `FROM` is set as the source base image `openjdk:8-jre-alpine` (Java version 8). Kafka has a different base image than Zookeeper, because of the performance testing, where was found that `openjdk:8-jre-alpine` (Java version 8) is well suited for Kafka. The `Alpine`[8] image is a minimal Linux distribution designed with containers in a mind.
  Instruction: `FROM openjdk:8-jre-alpine`

- `USER` is set as `root` at the beginning. The root user is required for updating the base image, installing utilities and changing the access permissions of files and folders.
  Instruction: `USER root`
  After installing and changing access permissions, a user have to be switched to the non-root user (jboss), because Openshift prohibits Dockerfiles, which has to execute the CMD instruction as the root user.
  Instruction: `USER jboss`
  There were issues with Openshift non-root setting, because this issue is not properly described in Openshift documentation[9]. This only working solution solution was advised by the Openshift community[10].

- `ENV` sets the environmental variables `SCALA_VERSION` to `2.11` and `KAFKA_VERSION` to `0.8.2.2` values, which are used in InfiSpector. The environmental variables that are set in Docker are included and used in Openshift automatically as

[7] https://hub.docker.com/r/mciz/kafka-docker-infispector/
[8] https://hub.docker.com/_alpine/
[9] https://www.openshift.com/
[10] https://stackoverflow.com/questions/41148127/openshift-online-zookeeper-from-dockerfile-pod-crash-loop-back-off

well. There are also the auxiliary ENV variables for SCALA, Kafka, and for Kafka main directory. The last but not least, the ENV variable `ADVERTISED_HOSTNAME` to `127.0.0.1` for setting the Kafka's internal server IP address.
Example instruction: `ENV KAFKA_VERSION 3.4.6`

- `EXPOSE` is set to communicate on ports 2181 for Zookeeper, 2888 for Kafka, 3888 for Druid cluster, and 8084 for real-time standalone Druid node.
  Instruction: `EXPOSE 2181 2888 3888 8084`

- `RUN` occurs two times in Kafka's Dockerfile. The first occurrence is executing chain of commands, which downloads and prepares Kafka, updates the base image system, and copies Kafka to auxiliary folders.
  The second RUN command changes permissions for the user `jboss` to regular group read and write permission, because **Openshift has prohibited Dockerfiles supporting CMD with root user**.
  Instruction: `RUN chown -R jboss:0 /opt/kafka && chmod -R g+rw /opt/kafka`

- `WORKDIR` sets current working directory to `/opt/kafka`.
  Instruction: `WORKDIR /opt/kafka`

- `VOLUME` sets mount point to folder `/opt/kafka/bin`.
  Instruction: `VOLUME ["/opt/kafka/bin"]`

- `CMD` is executing Kafka's internal server script `kafka-server-start.sh` for running Kafka with configuration set in file `server.properties`.
  Instruction: `CMD ["opt/kafka/bin/kafka-server-start.sh", "opt/kafka/config/server.properties"]`

### 7.3.3 Druid Dockerfile

Druid Dockerfile sets the local environment that Docker daemon needs for creating Druid image and afterwards the container. InfiSpector's Dockerfile for Druid can be found at Docker hub[11].

This Dockerfile is similar to Kafka's and Zookeeper's Dockerfile, therefore only differences and important instructions will be described.

- `FROM` is set as the source base image `progrium/busybox`, which is Linux lightweight distribution, which is the base for minimal Linux distribution `Alpine` used in Kafka's Dockerfile 7.3.2.
  Instruction: `FROM progrium/busybox`

- `USER` - both the `root` and the `jboss` users are applied.

- `EXPOSE` is set to communicate on ports 2181 for Druid, 2888 for Kafka, 3888 for Druid cluster, and 8084 for the real-time standalone Druid node.
  Instruction: `EXPOSE 2181 2888 3888 8084`

- `COPY` instruction copies specification file `InfiSpectorDruid.spec` to the `config` folder.
  Instruction: `COPY InfiSpectorDruid.spec /opt/druid/config`

[11] https://hub.docker.com/r/mciz/druid-docker-infispector/

- `VOLUME` sets mount point to folder `/opt/Druid/bin`.
  Instruction: `VOLUME ["/opt/Druid/bin"]`

- `CMD` is executing Java command to run a real-time standalone Druid node with detailed configuration in the specification file `InfiSpectorDruid.spec`.
  Instruction:

```
CMD  [''java'', ''-Xmx512m'', ''-Duser.timezone=UTC'', ''-
   Dfile.encoding=UTF-8'', ''-Ddruid.realtime.specFile=/
   opt/druid/config/InfiSpectorDruid.spec'', ''-classpath
   '', ''config/_common:config/realtime:lib/*'', ''io.
   druid.cli.Main server'', ''realtime'']
```

### 7.3.4 Building Dockerfiles

Firstly, Docker and also Docker daemon have to be active and running in the local machine. Secondly, Dockerfiles for Kafka, Druid and Zookeeper have to be available in a file system of the local machine. It is recommended, to have Dockerfiles in the root folder of the project. The next step is to build Kafka, Druid, and Zookeeper images from Dockerfiles in the local machine by Docker. Build Kafka image with Docker command:

```
$ docker build -t kafka:latest .
```

There is the analysis of Docker command for building Kafka image: `docker build` is the part which tells Docker daemon to build Docker image, `-t kafka:latest` informs Docker daemon that the built image will be called `kafka` and the version of Kafka image is being called `latest`. The last Docker command argument is the `dot` informing Docker daemon that Kafka Dockerfile is in the root directory of the project. After the execution of Docker command in terminal, Docker daemon starts to execute Kafka Dockerfile instructions in a given order.

Similar commands are being executed for the Druid and Zookeeper Dockerfiles. All the three above mentioned applications have their own solitary folders, where their Dockerfiles are being stored:

```
$ docker build -t zookeeper:latest <path to Zookeeper
Dockerfile>
```

```
$ docker build -t druid:latest <path to Druid Dockerfile>
```

The Docker images created by the Docker *build* command are now stored in a local machine in Docker registry tool. Now, built Docker images in Docker registry has to be transported into Openshift registry tool. To check if InfiSpector backend is built and stored in Docker registry properly can be seen after executing the command:

```
$ docker images
```

The `Docker images` command lists all built and pulled Docker images in the local machine. There are significant columns in the Docker images listing: *Repository*, which represents concrete names of images. *Tag* column displays version of built images, for

example, latest, v2, and 2.0. *Image ID* stands for the image's unique indentifier in Docker. *Created* is showing the date of images creation, pulling, or the last update. The last column *size* informs about the size of the images. If building process of the Kafka, Druid and Zookeeper images was successful, Docker images listing should be showing at least 4 images. The first image is the *CentOS*[12] system, which delivers environment for Java applications (Druid, Kafka, Zookeeper), other images should be images for support InfiSpector backend environment.

## 7.4 Openshift operations

The next step in the InfiSpector application backend deployment is a cooperation with Openshift. The Openshift Online *client* (CLI) is being used for communication with Openshift and also for Openshift managing from a terminal. To get logged in *CLI*, the Openshift Online web console has to be used to generate the unique token, which serves as an identification of a user in Openshift system. But for logging into the Openshift Online web console, a *GitHub account* has to be set up and confirmed by Red Hat. After logging into web console with the GitHub account, the drop–down menu in the top right hand corner of the web console should be opened, which is symbolized by the *down arrow* next to the user's name. Then the *Command Line Tools* from the drop–down menu should be chosen. In the Command Line Tools page, the first Openshift command specimen with clickable part *–token=...click to show token...* can be found under the introduction text. To get the token, command part *...click to show token...* should be clicked. After clicking on it, the Openshift Online web console generates the whole *oc login* command, which should be copied and executed in a terminal in a local machine:

```
$ oc login https://api.preview.openshift.com
  --token=...click to show token...
```

Finally, user is now logged into Openshift by the *CLI* in a terminal. Now, a new project has to be set up from the CLI command line or the web console, but using CLI is recommended, because executing a command in a CLI has higher priority and also better functionality than clicking a button in the web console. Openshift is still in a development phase and some functions are not working properly, especially in the web console, for example, clicking the button *Delete the deployment* in the web console shows the message of terminating of the specific deployment. In the reality, the chosen deployment is never going to be terminated, but a simple command in a CLI terminates the specific deployment immediately.

Every application developed or deployed in Openshift, has to have at least one *project* established. The new *project* especially for InfiSpector can be set up by the command:

```
$ oc new-project InfiSpector
```

## 7.5 Openshift registry

The Openshift *registry* is a scalable tool for storing and distributing Docker images. Openshift registry is a tool based on Docker registry tool [3].

[12] https://www.centos.org/

Openshift is manipulating with the internal Docker registry. In the case of InfiSpector, Docker formatted images were built in a local machine, pushed into the Docker registry, which is running also in a local machine by active Docker daemon, and then pushed built images from Docker registry to Openshift registry by Docker itself.

When the environment for using the Openshift registry is prepared by both Docker and Openshift, Docker can be granted to have an access to Openshift registry after the authentication in the CLI by the command:

```
$ docker login -u ‘oc whoami‘ -p ‘oc whoami -t‘
  registry.preview.openshift.com
```

The next step is to *tag* the images. The *tagging* has 2 parts. The first part of the *tagging* is adding a unique *Image ID* of the chosen image in the Docker registry in a local machine. The second part is adding the future path in Openshift registry in the form: *<Openshift registry url>/<project name>/<image name>*. To find out what is the value *Image ID* of Kafka image, use the `docker images` command:

```
$ docker images

REPOSITORY  TAG     IMAGE ID        CREATED          SIZE
druid       latest  4229d43c709c    3 weeks ago      675MB
zookeeper   latest  934d12924de8    3 weeks ago      175MB
kafka       latest  22bb463f6f63    3 weeks ago      422MB
centos      latest  98d35105a391    6 weeks ago      193MB
```

```
$ docker tag 22bb463f6f63 registry.preview.openshift.com/Infi
  Spector/kafka
```

The same operations, which have been done at Kafka image tagging, do tagging for both the Druid Docker image and the Zookeeper Docker image:

```
$ docker tag 4229d43c709c registry.preview.openshift.com/Infi
  Spector/druid

$ docker tag 934d12924de8 registry.preview.openshift.com/Infi
  Spector/zookeeper
```

The next step is to upload the tagged images to the Openshift registry by executing the commands:

```
$ docker push registry.preview.openshift.com/Infi
  Spector/kafka

$ docker push registry.preview.openshift.com/Infi
  Spector/druid

$ docker push registry.preview.openshift.com/Infi
  Spector/zookeeper
```

The Docker images, which are pushed into the Openshift registry, are saved as *image streams*. A verification if all the images were pushed from the Docker registry to the

Openshift registry successfully can be done by Openshift image streams listing executed by the CLI command:

```
$ oc get is

NAME       DOCKER REPO                                    TAGS
druid      172.30.47.227:5000/InfiSpector/druid           latest
zookeeper  172.30.47.227:5000/InfiSpector/zookeeper       latest
kafka      172.30.47.227:5000/InfiSpector/kafka           latest
```

## 7.6 Templates

A *template* describes a composition of Openshift objects, their metadata and settings. Objects are generated or updated from a template, but the object update progresses differently. Every created object in Openshift that has to be updated is terminated and recreated with updated settings.

A template can be written in both the JSON and the YAML form [18]. In InfiSpector, templates are written in the YAML form and they are stored in InfiSpector GitHub repository[13]. For InfiSpector Node.js application is a template created in the Openshift web console from the built-in Node.js template, but for Kafka, Zookeeper, and Druid are templates created by hand.

### 7.6.1 Kafka template with Zookeeper

Kafka template stores every setting, which is needed to create both Kafka and Zookeeper Openshift applications.

**Kafka, in its own official distribution, has a built-in Zookeeper server, which is used instead of a solitary Zookeeper server due to an optimization.**

The complete Kafka template in the YAML form can be found in Section B.2.1. Here is the listing, which describes only the most important parts of the Kafka template:

- `Metadata` provides a general description and contains the name of the Kafka template.
- `Parameters` provides information about Kafka Openshift registry image and the prefix for the each of Kafka object created by the Kafka template.
- `Objects` is the listing of all objects, which are created from the Kafka template, their deployment configuration, names, number of replicas, and containers. There are two containers in the Kafka template: Zookeeper and Kafka container. In the Kafka template are specified container's names, commands for starting servers with their arguments, and ports, through which are containers exposed to other containers or pods in the project.

### 7.6.2 Kafka service with Zookeeper

Services are assigned an IP address and a port pair, when they are accessed. Due to an optimization, Kafka and Zookeeper containers are in the same pod and have their own service.

[13] https://github.com/infinispan/infispector

The complete Kafka service in the YAML form can be found in Section B.3.1. There is the listing describing Kafka service:

- `Metadata` provides a general description and contains the name of the Kafka service.
- `Spec` describes information about the ports, through which are containers exposed to other containers or pods in InfiSpector. The port for the Kafka container has value 9092 and for the Zookeeper container has value 2181.
- `Selector` specifies, which Kafka service is linked with the Kafka deployment, which is created from Kafka template.

### 7.6.3 Druid template

Druid template stores every setting which is needed to create Druid Openshift application.

The complete Druid template in the YAML form can be found in Section B.2.2. There is the listing describing only the most important parts of the Druid template:

- `Metadata` provides a general description and and contains the name of Druid template.
- `Parameters` provides information about the Druid Openshift registry image and the prefix for the each of object created by the Druid template.
- `Objects` is the listing of all objects, which are created from the Druid template, their deployment configuration, names, number of replicas, and containers. There is one Druid container generated from the Druid template. In the Druid template are specified container's name, commands for starting the standalone realtime Druid server with its arguments, and ports, through which is container exposed to other containers or pods in the project.

### 7.6.4 Druid service

The complete Druid service in the YAML form can be found in Section B.3.2. There is the listing describing the Druid service:

- `Metadata` provides a general description and contains the name of the Druid service.
- `Spec` describes information about the ports, through which are containers exposed to other containers or pods in InfiSpector. The port for Druid container has value 9092 and for Zookeeper container has the value 2181.
- `Selector` specifies that Druid service is linked with Druid deployment, which is created from the Druid template.

### 7.6.5 InfiSpector template and service

The InfiSpector template differs from other templates, because the InfiSpector template was created in Openshift web console from the built-in Javascript Node.js Openshift template by the following: Firstly ,the Openshift web console has to be set up and user should be navigated through creation of a new Openshift project. Clicking the "Add to project" button in the top of the Openshift web console page, the catalog of Openshift built-in templates is opened. Then the *Javascript* template should be picked and user should

continued through the *Node.js builds source code* by clicking the "select" button. Lastly, a name of the template and also application name should be filled and added link to GitHub repository with application source code. In InfiSpector case is the GitHub link InfiSpector GitHub repository[14].

Using the Openshift web console is easier solution when an application source is a source code, not a Docker image. InfiSpector service was created automatically during the process of creating the InfiSpector template.

#### InfiSpector route

The InfiSpector route exposes the provisional non–secured *InfiSpector domain*[15] *hostname* `InfiSpector4.InfiSpector.44fs.preview.openshiftapps.com`, which is reachable from outside of Openshift by external clients. **The provisional domain hostname will be different in every deployment update**.

The complete InfiSpector route in the YAML form can be found in Section B.4.1. There is the listing describing the most important parts of the InfiSpector route:

- `Metadata` provides a general description, namespace, labels, and the name of InfiSpector route.
- `Spec` is showing InfiSpector domain hostname and service, which are exposed by the InfiSpector domain hostname and the port, which is paired with the InfiSpector domain hostname with the value 8080.

## 7.7 Running InfiSpector backend applications

The current situation in InfiSpector is that there are pushed Docker images of the InfiSpector backend applications into the Openshift registry, InfiSpector is set up in Openshift and all Openshift configuration and template files in the YAML form for InfiSpector backend applications are already prepared and pushed into the InfiSpector GitHub repository.

The next step is to add YAML files into Openshift. For easier manipulation with templates, services, and potential routes YAML files were merged into the one configuration YAML file each for Druid, Kafka, and InfiSpector. The final merged configuration file for Kafka and Zookeeper tool is shown in Section B.5.

For adding the final merged configuration file from InfiSpector GitHub repository to the Openshift InfiSpector, the specific CLI command should be used:

```
$ oc create -f https://raw.githubusercontent.com/InfiSpector/\
path/to/merged/configuration/file/kafka-config.yaml

$ oc create -f https://raw.githubusercontent.com/InfiSpector/\
path/to/merged/configuration/file/druid-config.yaml

$ oc create -f https://raw.githubusercontent.com/InfiSpector/\
path/to/route/configuration/file/InfiSpector-route.yaml
```

[14] https://github.com/infinispan/infispector
[15] https://tools.ietf.org/html/rfc1034

Because the InfiSpector Node.js application template was created from a built-in Node.js Openshift template, which created also InfiSpector service, so only the InfiSpector route was added from the GitHub to Openshift InfiSpector project. Additionally, InfiSpector Node.js application is also deployed and already running which was arranged by Openshift automatically after InfiSpector Node.js template creation in Openshift web console.

The next step is to build and deploy Kafka with Zookeeper and Druid templates into running Openshift applications. Everything is prepared, so only execution of the specific CLI commands for the new applications creation remains:

```
$ oc new-app kafka

$ oc new-app druid
```

The progress of deploying both Kafka and Druid applications can be tracked by the Openshift CLI commands:

```
# to show the list of the pods in Openshift
$ oc get pods

NAME                    READY  STATUS      RESTARTS
kafka-1-bdd89           2/2    Running     3
druid-3-9bkld           1/1    Running     1
InfiSpector4-2-4mm9m    1/1    Running     1
InfiSpector4-2-build    0/1    Completed   0

# to look deeply in the specific pod
$ oc describe pod kafka-1-bdd89
```

After checking the listing above, it is obvious that the Kafka pod with Zookeeper, the Druid pod, and the InfiSpector pod are running without errors. The Remote deployment of InfiSpector to Openshift Online is completed successfully.

# Chapter 8

# Optimal working Openshift solution for InfiSpector

*Openshift Online NextGen Preview account* (called account), which was granted as a free account for a one year duration by Red Hat, for the Remote Deployment of InfiSpector to Openshift Online purposes, has limited resources.

The account is allowed to have only one project at the same time, in addition, the account has only limited resources that are limited to four cores of CPU and two GiB of a memory. Each of four containers in the InfiSpector deployment occupies one CPU core, therefore InfiSpector is using every CPU core that Openshift offers. The Zookeeper container is using maximally 307 MiB, the Kafka container is using maximally 307 MiB, the Druid container is using up to 512 MiB, and the InfiSpector container is using up to 512 MiB. All containers in total are using up to 1638 MiB of a memory. Loading InfiSpector Node.js web page is slow, because of the low memory resources assigned to the container with InfiSpector. Despite the fact that there are still 362 MiB of a free memory in the Openshift that free memory cannot be assigned to anything. If the 362 MiB of a free memory are being used, whole Openshift project collapses in the next re–deployment. New supporting deployment nodes, which are automatically created in every deployment, demands at least 350 MiB for their running.

In the first project plan, there was the solitary Zookeeper pod, to enable Zookeeper to scale. Because of a lack of a memory, the Zookeeper solitary pod was terminated. Instead of the Zookeeper solitary pod was used a container with Kafka's built–in Zookeeper server inside the Kafka pod.

## 8.1 Future directions

There are the following issues and implements in the InfiSpector remote deployment in Openshift that can be fixed in the near future:

- Implement environmental variables to the Dockerfiles, to easily change versions of tools in the future.
- The input data to InfiSpector should flow through Kafka, then be transported to Druid, and finally be delivered to InfiSpector Node.js server which is communicating with the front–end. Because of the issue with setting up secured binary route from

Openshift, data are injected to InfiSpector in the JSON file right into Druid database. So one of the future directions is to put secured Kafka's route into operation.

- Change the domain InfiSpector generated name, because the hostname `InfiSpector4.InfiSpector.44fs.preview.openshiftapps.com` generated by Openshift[1] is not practical, and this name is changing after every deployment.
- InfiSpector should be able to process a large amount of data, e.g. 18 TiB of Infinispan cluster logs, but with current computational resources, the scaling, and therefore faster data processing, is almost impossible, so a goal is to get more computational resources in Openshift Online, switch from Openshift Online to Openshift Online Enterprise, or install and run Openshift locally on a server or a computer. Openshift running locally has the same functionalities as Openshift Online, but the computational resources can be increased.

[1] https://tools.ietf.org/html/rfc1034

# Chapter 9

# Conclusion

The aim of this Bachelor's thesis is to deploy InfiSpector tool to a cloud environment and provide easy deployment of new versions of InfiSpector during its development. The secondary goal of this thesis is to familiarize Infinispan community, especially developers with InfiSpector tool, how to set up InfiSpector backend, deploy InfiSpector on a cloud service, and make a guide how to deploy complex application on Openshift cloud service.

After the research and comparison of cloud environments, Openshift Online containers application combined with Docker tool were chosen as the most suitable way for running and developing InfiSpector tool cloud infrastructure.

The theoretical background of Infinispan cache cluster, Docker with Dockerfiles, and Openshift has been studied and applied in practice.

The way how to deploy InfiSpector to Openshift was discussed with the InfiSpector community. InfiSpector deployment to Openshift was successfully done. Working Dockerfiles, Docker images and their application setting of InfiSpector backend applications has been created and successfully pushed into Openshift registry. Openshift templates, services and routes for InfiSpector backend has been created, deployed and are running.

Despite the fact that a working InfiSpector deployment to Openshift was successfully performed, InfiSpector backend functionality is not complete. Openshift Online did not offer as much resources and admin privileges as was needed for putting InfiSpector fully into operation which caused two issues. The first issue is that Openshift cannot process big data logs, because of the Openshift Online computational limitation that does not allow InfiSpector to scale accordingly. Second issue is that the secured binary Kafka route which allows external sources to send data into InfiSpector, is not functional due to Openshift Online admin privileges restriction. Now, InfiSpector data input is a JSON format document that is injected into Druid instead of Kafka. Both issues can be solved by installing and running Openshift Online on a local machine, or switching from Openshift Online to Openshift Online Enterprise.

The outcome of this thesis as a part of the official distribution of InfiSpector was introduced at DevConf 2017, and will be presented in Red Hat as a part of the Red Hat Laboratory at FIT conference in May 2017.

# Bibliography


[1] Apache Software Foundation: Kafka. [Online; visited: 25.11.2016].
Retrieved from: https://kafka.apache.org/

[2] Apache Software Foundation: Zookeeper. [Online; visited: 25.11.2016].
Retrieved from: https://zookeeper.apache.org/

[3] Docker community: Docker. [Online; visited: 25.11.2016].
Retrieved from: https://docs.docker.com/

[4] Druid community: Druid.io. [Online; visited: 25.12.2016].
Retrieved from: http://druid.io

[5] Electronit frontier foundation community: HTTPS Everywhere. [Online; visited: 15.4.2017].
Retrieved from: https://www.eff.org/https-everywhere/faq

[6] Git community: Git distributed even if your workflow isn't. [Online; visited: 15.4.2017].
Retrieved from: https://git-scm.com/

[7] Infinispan community: Infinispan. [Online; visited: 25.11.2016].
Retrieved from: http://infinispan.org/

[8] JBoss community: JBoss organization. [Online; visited: 15.4.2017].
Retrieved from: http://www.jboss.org/

[9] JSON community: Introducing JSON. [Online; visited: 15.4.2017].
Retrieved from: http://www.json.org/

[10] Node.js Foundation: Node.js docs. [Online; visited: 15.4.2017].
Retrieved from: https://nodejs.org/en/docs/

[11] Openshift community: Openshift. [Online; visited: 1.1.2017].
Retrieved from: https://docs.openshift.org/latest/welcome/index.html

[12] Aho, A. V.: Algorithms for finding patterns in strings. [Online; visited: 20.4.2017].
Retrieved from: http://infolab.stanford.edu/~ullman/focs/ch10.pdf

[13] Christoff, J.: The Transmission Control Protocol. [Online; visited: 15.4.2017].
Retrieved from: http://condor.depaul.edu/jkristof/technotes/tcp.html

[14] Edlich, S.: NoSQL DEFINITION: Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable. [Online; visited: 15.4.2017].
Retrieved from: http://nosql-database.org/

[15] Fielding, R. T.: Architectural Styles and the Design of Network-based Software Architectures. [Online; visited: 25.12.2016].
Retrieved from: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[16] Johnston, C.: New hard drive write method packs in one terabit per inch. [Online; visited: 15.4.2017].
Retrieved from: https://arstechnica.com/science/2010/05/new-hard-drive-write-method-packs-in-one-terabyte-per-inch/

[17] Odersky, M.: Scala. [Online; visited: 15.4.2017].
Retrieved from: https://www.scala-lang.org/

[18] Oren Ben-Kiki, C. E.: YAML Ain't Markup Language. [Online; visited: 20.4.2017].
Retrieved from: http://www.yaml.org/spec/1.2/spec.html

[19] Sýkora, T.: Infispector. [Online; visited: 25.11.2016].
Retrieved from: http://research.redhat.com/projects/infispector/

[20] Sýkora, T.: Infispector on github. [Online; visited: 25.11.2016].
Retrieved from:
https://github.com/infinispan/infispector/blob/master/README.md

# Appendices

# Appendix A

# Dockefiles

## A.1 Kafka Dockerfile

```
FROM centos
MAINTAINER mciz

ENV DRUID_VERSION   0.8.3
RUN mkdir -p /opt/kafka/ \
    && cd /opt/kafka/ \
    && yum -y install java-1.8.0-openjdk-headless tar \
    && curl -s https://www.mirrorservice.org/\
    sites/ftp.apache.org/ \
    kafka/0.10.1.1/kafka_2.11-0.10.1.1.tgz | \
    tar -xz --strip-components=1 \
    && yum clean all

RUN chmod -R a=u /opt/kafka

WORKDIR /opt/

# 2181 is zookeeper, 9092 is kafka, 3888 is druid,
# 8084 is realtime druid
EXPOSE 2181 2888 3888 9092
```

## A.2 Druid Dockerfile

```
FROM centos
MAINTAINER mciz

ENV DRUID_VERSION   0.8.3
RUN yum -y install java-1.8.0-openjdk-headless tar \
    && yum clean all

RUN mkdir -p /opt/druid/ \
```

```
        && cd /opt/druid/ \
        && curl -s http://static.druid.io/artifacts/ \
        releases/ \
        druid-$DRUID_VERSION-bin.tar.gz | \
        tar -xz --strip-components=1 \
    && yum clean all

COPY infispectorDruid.spec /opt/druid/config/
COPY sampleMessages.json /opt/druid/

RUN chmod -R a=u /opt/druid/
WORKDIR /opt/druid/

# 2181 is zookeeper, 9092 is kafka, 3888 is druid,
# 8084 is realtime druid
EXPOSE 2181 3888 9092 8084
```

# Appendix B

# Openshift

## B.1 Openshift client commands

**Basic commands**

- `$ oc --help`

  Displays useful commands in terminal.

- `$ oc login`

  Logs in to a server or a cluster.

- `$ oc  projects`

  Displays existing projects.

- `$ oc project <project_name>`

  Switches to selected project.

- `$ oc new-project <project_name>`

  Sets up a new project with given name.

- `$ oc  new-app <app_name>`

  Sets up a new application with given name.

- `$ oc status`

  Shows an overview of the current project.

- `$ oc explain <command_or_object>`

  Shows documentation of resources.

- `$ oc types <object>`

  Shows an introduction to concepts and types.

**Build and Deploy commands**

- `$ oc new-build <object>`

  Creates a new build configuration.

- `$ oc tag <image>`

  Tags existing images, including docker images into image streams.

- `$ oc import-image <image_name>`

  Imports images from a Docker registry.

- `$ oc start-build <object>`

  Starts a new build.

- `$ oc cancel-build <object>`

  Cancels running, pending or new builds.

- `$ oc deploy <deployment_name>`

  Starts or cancels deployment.

**Application Management commands**

- `$ oc get <project_or_object>`

  This is a useful command, which displays one or many resources: Pods, projects, services, routes, etc.

- `$ oc describe <objectt>`

  This is also a useful command, which shows details of a specific resource or group of resources. For example, internal substeps in pod deployment could be monitored.

- `$ oc edit <object>`

  Edits a resource on the server. It is used mainly for quick small fixes. Solving big problems is not recommended.

- `$ oc set <object>`

  Helps set specific features on objects 5.2.8.

- `$ oc label <object_or_group_of_objects>`

  Changes label on a group of resources.

- `$ oc delete <object>`

  Deletes one or more resources. For deleting all resources of one type, use delimeter –all.

- `$ oc expose <app>`

  Exposes a replicated application as a service or route, if it is not have been already done separately in Dockerfile, service or route before.

- `$ oc scale <deployment>`
  `Adjust the amount of total replicated pods.`

### Troubleshooting and Debugging commands

- `$ oc logs <object>`

  Prints the object's logs.

### Settings commands

- `$ oc logout`

  Terminates the actual server session.

- `$ oc whoami`

  Echoes name of the actual user. A must have command at uploading images to Openshift registry 5.2.7.

## B.2 Templates

### B.2.1 Kafka and Zookeeper template

```
- apiVersion: v1
  kind: Template
  metadata:
    name: kafka
    annotations:
      description: 1-pod Apache Kafka + ZooKeeper
      tags: messaging,streaming,kafka
  parameters:
  - name: NAME
    description: Name prefix for each object created
    required: true
    value: kafka
  - name: IMAGE
    description: Image with Apache Kafka and Apache ZooKeeper
    required: true
    value: 172.30.47.227:5000/InfiSpector/kafka
  - name: VOLUME_CAPACITY
    description: Persistent volume capacity per pod
    required: true
```

```
  value: 256Mi
objects:
- apiVersion: v1
  kind: DeploymentConfig
  metadata:
    name: ${NAME}
  spec:
    replicas: 1
    selector:
      deploymentconfig: ${NAME}
    template:
      metadata:
        labels:
          deploymentconfig: ${NAME}
      spec:
        containers:
        - name: apache-zookeeper
          image: ${IMAGE}
          command:
          - kafka/bin/zookeeper-server-start.sh
          args:
          - kafka/config/zookeeper.properties
          volumeMounts:
          - mountPath: /kafka/tmp/zookeeper
            name: zookeeper
          ports:
          - containerPort: 2181
        - name: kafka
          image: ${IMAGE}
          command:
          - kafka/bin/kafka-server-start.sh
          args:
          - kafka/config/server.properties
          - --override
          - advertised.host.name=${NAME}
          - --override
          - zookeeper.connect=kafka
          volumeMounts:
          - mountPath: /kafka/tmp/kafka-logs
            name: kafka-logs
          ports:
          - containerPort: 9092
        volumes:
        - name: kafka-logs
          emptyDir: {}
        - name: zookeeper
          emptyDir: {}
```

### B.2.2 Druid template

```
- apiVersion: v1
  kind: Template
  metadata:
    name: druid-solo
    annotations:
      description: 1-pod Druid
      tags: messaging, db, NoSQL, database, druid
  parameters:
  - name: NAME
    description: Name prefix for each object created
    required: true
    value: druid-solo
  - name: IMAGE
    description: Image with Druid
    required: true
    value: 172.30.47.227:5000/InfiSpector/druid-is
  - name: VOLUME_CAPACITY
    description: Persistent volume capacity per pod
    required: true
    value: 256Mi
  objects:
  - apiVersion: v1
    kind: DeploymentConfig
    metadata:
      name: ${NAME}
    spec:
      replicas: 1
      selector:
        deploymentconfig: ${NAME}
      template:
        metadata:
          labels:
            deploymentconfig: ${NAME}
        spec:
          containers:
          - name: druid
            image: ${IMAGE}
            command:
            - java
            args:
            - -Xmx256m
            - -XX:MaxDirectMemorySize=200000000
            - -Ddruid.zk.service.host=kafka
            - -Duser.timezone=UTC
```

```
        - -Dfile.encoding=UTF-8
        - -Ddruid.realtime.specFile=/opt/druid/config/
           infispectorDruid.spec
        - -classpath
        - "/opt/druid/config/_common:/opt/druid/config/
           realtime:/opt/druid/lib/*"
        - io.druid.cli.Main
        - server
        - realtime
        volumeMounts:
        - mountPath: /tmp/druid-logs
          name: druid-logs
        ports:
        - containerPort: 8084
      volumes:
      - name: druid-logs
        emptyDir: {}
```

## B.3 Services

### B.3.1 Kafka and Zookeeper services

```
- apiVersion: v1
  kind: Service
  metadata:
    name: ${NAME}
  spec:
    ports:
    - name: kafka
      port: 9092
    - name: zookeeper
      port: 2181
    selector:
      deploymentconfig: ${NAME}
```

### B.3.2 Druid service

```
- apiVersion: v1
  kind: Service
  metadata:
    name: ${NAME}
  spec:
    ports:
    - name: druid
      port: 8084
    selector:
      deploymentconfig: ${NAME}
```

## B.4 Routes

### B.4.1 InfiSpector route

```
apiVersion: v1
kind: Route
metadata:
  name: InfiSpectorr4
  namespace: InfiSpector
  selfLink: /oapi/v1/namespaces/InfiSpector/routes/\
  InfiSpector4
  uid: a71c297c-1f8e-11e7-b3b6-0e3d364e19a5
  resourceVersion: '1114541183'
  creationTimestamp: '2017-04-12T14:45:20Z'
  labels:
    app: infispector4
  annotations:
    openshift.io/generated-by: OpenShiftWebConsole
    openshift.io/host.generated: 'true'
spec:
  host: infispector4-InfiSpector.44fs.preview.\
  openshiftapps.com
  to:
    kind: Service
    name: infispector4
    weight: 100
  port:
    targetPort: 8080-tcp
  wildcardPolicy: None
status:
  ingress:
    - host: infispector4-InfiSpector.44fs.preview.openshiftapps.com
      routerName: router
      conditions:
        - type: Admitted
          status: 'True'
          lastTransitionTime: '2017-04-12T14:45:20Z'
      wildcardPolicy: None
```

## B.5 Kafka and Zookeeper configuration

```
kind: List
apiVersion: v1
metadata: {}

items:

- apiVersion: v1
```

```
kind: Template
metadata:
  name: kafka
  annotations:
    description: 1-pod Apache Kafka + ZooKeeper
    tags: messaging,streaming,kafka
parameters:
- name: NAME
  description: Name prefix for each object created
  required: true
  value: kafka
- name: IMAGE
  description: Image with Apache Kafka and Apache ZooKeeper
  required: true
  value: 172.30.47.227:5000/InfiSpector/kafka-is
- name: VOLUME_CAPACITY
  description: Persistent volume capacity per pod
  required: true
  value: 256Mi
objects:
- apiVersion: v1
  kind: DeploymentConfig
  metadata:
    name: ${NAME}
  spec:
    replicas: 1
    selector:
      deploymentconfig: ${NAME}
    template:
      metadata:
        labels:
          deploymentconfig: ${NAME}
      spec:
        containers:
        - name: apache-zookeeper
          image: ${IMAGE}
          command:
          - kafka/bin/zookeeper-server-start.sh
          args:
          - kafka/config/zookeeper.properties
          volumeMounts:
          - mountPath: /kafka/tmp/zookeeper
            name: zookeeper
          ports:
          - containerPort: 2181
        - name: kafka
          image: ${IMAGE}
          command:
```

```
          - kafka/bin/kafka-server-start.sh
          args:
          - kafka/config/server.properties
          - --override
          - advertised.host.name=${NAME}
          - --override
          - zookeeper.connect=kafka
          volumeMounts:
          - mountPath: /kafka/tmp/kafka-logs
            name: kafka-logs
          ports:
          - containerPort: 9092
        volumes:
        - name: kafka-logs
          emptyDir: {}
        - name: zookeeper
          emptyDir: {}
- apiVersion: v1
  kind: Service
  metadata:
    name: ${NAME}
  spec:
    ports:
    - name: kafka
      port: 9092
    - name: zookeeper
      port: 2181
    selector:
      deploymentconfig: ${NAME}
```