Distributed Microservice Architecture with Docker

Imanol Urra Ruiz

University Master's Degree in Computer Engineering

Félix Freitag

22 June 2016

To **Olga** and **my family**:
Who remind me who I am,
Who show me the potential I have,
Who believe in me,
Who support me with patience,
Who are always there for me.

To all **NGCS team** at **Arsys**:
For taking part in the progress of my career,
For the help,
For teaching me how to perform a Hasselhoff attack.

To **Félix Freitag**:
For his guidance,
For being my mentor in this project,
For showing me how to take the right decisions.

Index

Index of Figures

Information about Final Master's degree Project

Author

Imanol Urra Ruiz

Consulter

Félix Freitag

Delivery date

22 June 2016

Area

Distributed Systems

Degree

University Master's Degree in Computer Engineering

Abstract

In a globalized world where big opportunities come through the Internet, where people is accustomed to be bombarded with a big quantity of information and they want this information to be available as quick as possible. How can an organization expand its business all over the world? How can they build an application or product to collect information and deploy it in a distributed system? A problem to solve comes out. A distributed data scraper designed under a distributed architectural style represented called *microservice architecture* implemented with *Docker* technologies is a possible solution. A system under a *Docker Swarm* cluster that automatically deploy and distributes data scrapers and other services involved among the different nodes in the cluster, where each service is independently working for providing highly available collections of information to an endpoint for being consumed by and application.

Keywords

Distributed system, Microservice architecture, Docker, Data scraper

1. Introduction

1.1. Context and justification of the project

Imagine a world where people are hyperconnected to the Internet, where information is on every corner spread at every device connected to the net. Imagine a globalized culture where each business tries to take advantages of the Internet to expand their products around the globe. Then, how is possible to an application or system from a company like *Skyscanner*, *Momondo* or even *Google* or *Facebook* collect that big quantity of information and serve it to each person as fast as a beam of light? How do they do it?

Here comes the problem to solve. How is possible to a company like *Skyscanner* handles all collected content or information from other sources and offer it to his or her final users?

This Final Master's degree Project is a workaround of an alternate point of view focused on a solution of a distributed data scraper developed using *microservice architecture* and *Docker* platform. Trendy concepts in the field of software development.

Microservice architecture provides a philosophy of how an application should be developed and deployed. It is used to build distributed software systems and is considered as a more concrete and modern interpretation of *Service Oriented Architectures*. *Docker* instead, it can be considered as a platform that provides a solution to containerize or package a deployable, isolated and distributed application with all needed components where is getting more importance every day.

1.2. Project objectives

This Final Master's degree Project is going to be focused on exposed terms of *microservice architecture* and *Docker*. Both suppose a big scope that can evolve the exposed problem. For this reason, the main objectives are:

1. Study of microservice architecture. What it is and how can be applied it in software development.
2. Study of Docker platform. What it is and how can use each tool with microservice architecture.
3. Development and deployment of a distributed data scraper.

The system is going to be designed and implemented with *microservice architecture* and *Docker* under a cluster of three nodes spread over the world. Also, it is going to take advantage of using new languages and technologies like *Node.js*, *MongoDB* and *Redis* as personal objectives to get used to with *MEAN stack*.

1.3. Followed approach and method

After considering the best approach and how it can cover the scope of the project where there are parts of learning about and a part of putting into practise *microservices architecture* and *Docker*. I decided to make a new system complemented with reusable components, which are free to be used on *Docker Hub*. This makes it easier to build a demonstrable and usable system. Also show de benefits of the architecture and technology used for the development of the system.

1.4. Work plan

The resources needed for the realization of the project are the common resources used in a software development. In my case, *PHPStorm* is the selected program for coding the use case because is a complete IDE that offers a proper integration with *MongoDB*, *Node.js* and *Docker*. Also, I installed Docker Toolbox in my *MacBook* for the reason that *Docker* use Linux containerization, and it is essential to provide all necessary components for using it in a local environment. Finally, I decided to use the one-month free trial of *1&1 Cloud Panel* for creation of the cluster.

Related to resources, a work plan is needed for the realization of the project, which fits with the requirement of a three continuous assessments. Each assessment covers different tasks to accomplish the objectives.

1.4.1. Planning

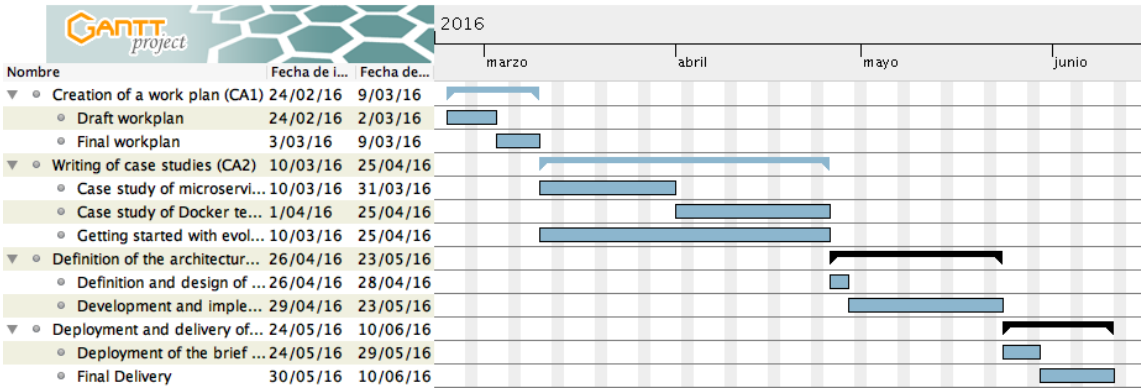

Figure 1. Gantt diagram of work plan

The work plan can be represented with the following Gantt diagram.

Creation of a work plan (CA1)

From 24/02/2016 to 09/03/2016: 15 days

This stage has as result the delivery of a document with a plan on how the project is distributed through the realization periods. Also, it is the moment of getting used to the objectives of the work itself.

Writing of case studies (CA2)

From 10/03/2016 to 25/04/2016: 47 days

This period of time is going to be dedicated for the investigation, analysis and writing of both use case exposed in the chapter 1.4 about *microservice architecture* and *Docker* platform.

Also during this phase, I am going to make the most of my time to get used to evolved technologies and platforms and to set up the development environment for the next period of time to design and build the system for the brief use case. Furthermore, this supposes to test the technologies of *Docker* and the programming language of *Node.js*, bringing an idea about what the system will be like.

Definition of the architecture and development of brief use case (CA3)

From 26/04/2016 to 23/05/2016: 28 days

Thanks to the phase done before, I am going to have an idea of what the distributed data scraper architecture is. Despite defining and documenting the architecture, most of this period of time is going to be dedicated to the development and implementation of the brief use case.

Deployment and delivery of the Final Master's degree Project (Final Delivery)

From 24/05/2016 to 10/05/2016: 18 days

At the last stage, the result is the implementation and deployment of the system in a three-node cluster and finalise the *Final Master's degree Project* report to be delivered.

1.5. Brief summary of obtained products

At the end, the obtained product is a distributed data scraped system developed under a *microservice architectural style* with *Docker* platform. Thus, this system is able to deploy scraper containers on demand when a new configuration is added.

1.6. Brief description of the other chapters from memory

Finally, next chapters are going to talk about the terms explained before. The second chapter is related with case studies about *microservices* and *Docker*, both studies are the backbone of this Final Master's degree Project. Third chapter is about the development, implementation and deployment of the brief use case taking into account learned terms in the chapter before. Finally, I am going to conclude this report with a global conclusion about the project.

2. Case studies

2.1. Microservice architecture

This first of two case studies exposed in this *Final Master's degree Project*, is focused on a new-fashioned way of designing software applications named as *microservice architecture*.

There is not a concrete definition for *microservice architecture*, but as one of the gurus about this term Martin Fowler says in his article, the *microservice architecture* is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms like HTTP resource API [1].

Also, the philosophy of *microservice architecture* in essence is equals to the Unix philosophy of *Do one thing and do it well*, being described as small and fine-grained services to perform a single function [5]. The organization culture should embrace automation of deployment and testing. This eases the burden on management and operations. The culture and design principles should embrace failure and faults, similar to anti-fragile systems. Each service is elastic, resilient, composable, minimal, and complete.

The popularity of this architectural style has increased due to different factors strongly associated with the arrival of distributed systems and the inconvenience and difficulties of developing and deploying monolithic applications in the cloud. Additionally, *microservice architecture* has been getting more importance inside *Service Oriented Architecture* or *SOA* paradigm

For this reason, it is essential to divide this case study for comparing *microservice architecture* to *monolithic architecture* and *SOA*.

2.1.1. Monolithic and Microservices Architectures

It is said that an application built as a monolithic, is built as a single unit or single logical executable that normally handles HTTP requests, executes domain logic

retrieving or updating data from database which populate a view for sending to the browser.

Even though a monolithic application can be successful, the nature of this architecture makes some inconvenience arise. Any changes and new implementations that occur during development involve building and deploying a new version of the application ending in an increase of possibilities of failures in the application due to its complexity. The scalability also shows inconveniences, you can horizontally scale the application by running many instances of the entire application in different servers.

In contrast, in addition to the description about microservices at the beginning of this case study, *microservice architecture* lends to services deploy and scale independently providing a firm module boundary and the possibility of write services in different programming languages.

Thus, the *Figure 2* describes both architectures.

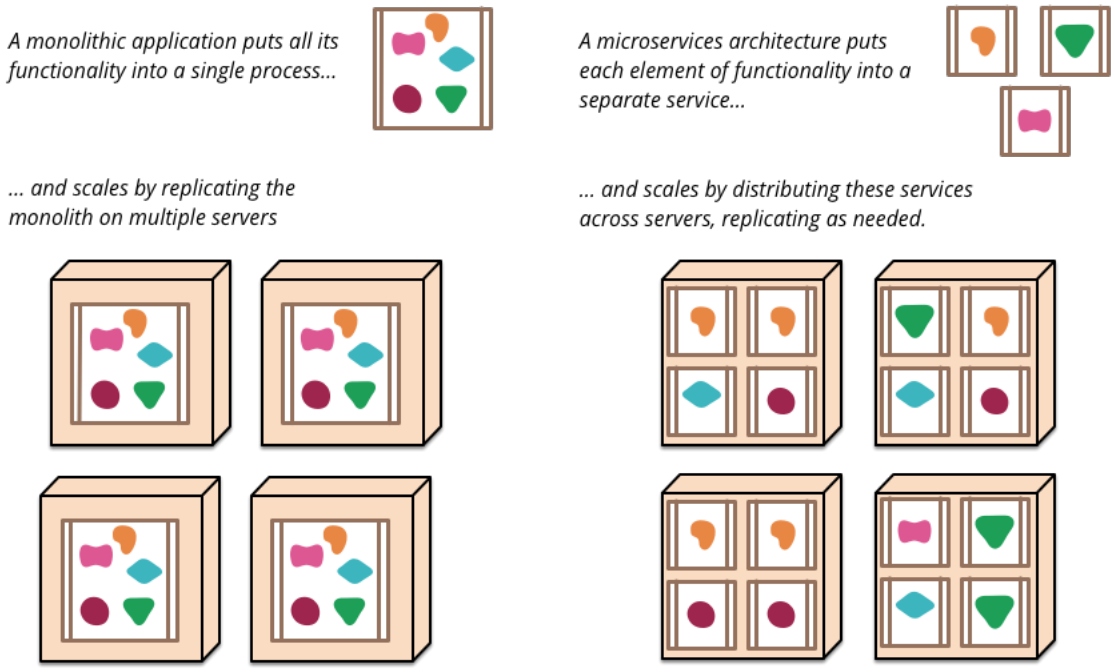

Figure 2. Monolithic architecture versus microservice architecture [Source: http://martinfowler.com]

Moreover, *microservice architecture* shows common characteristics that can suppose a great benefit in its implementation in order to use *monolithic architecture*:

- Componentization via Services
- Organized around Capabilities and Business functionalities
- Products instead of Projects
- Integration of smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation and Continuous Delivery
- Design for failures

Componentization via Services

For a long time, with the desire of software developers to build reusable code, it has been developing common libraries as part of software, which are components linked into a program called using in-memory functions calls.

Components can be defined as a unit of software that encapsulates a set of related functions, emphasizes the separation of concerns by loosely coupled and independently reusable, replaceable and upgradeable.

Regarding that, *monolithic* and *microservice architecture* use libraries, but *microservices* componentize the application by breaking down into services, that is, playing the role of converting components into services. Services as components will provide the architecture with the capability of being independently deployable rather than libraries, so you need to redeploy the entire application when a change is made. Also, componentization via services fit better in a distributed era.

Organized around Capabilities and Business functionalities

When trying to divide a large application into parts it tends to split it depending on the organization structure following unknowingly *Conway's Law*.

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." Melvyn Conway, 1967

This law can be illustrated with a common organization where there are different teams for different technology layers, UI, middleware or business logic and database specialist teams. This approach is common when a monolithic application is developed.

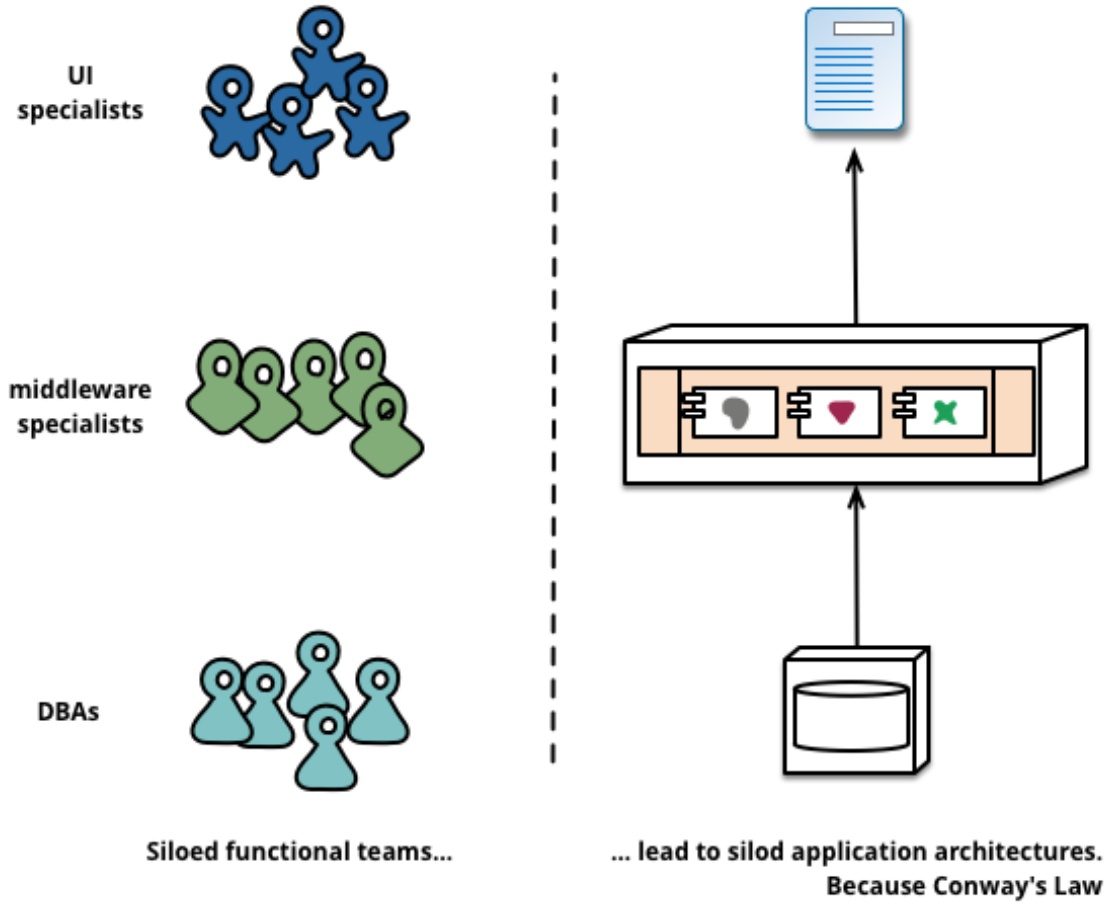

Figure 3. Conway's Law: Common organization in a monolithic application [Source: http://martinfowler.com]

With *microservices architecture* instead, the organization is different and tend to split the large application into cross-functional teams by service or business capabilities where each team needs to implement a complete software stack.

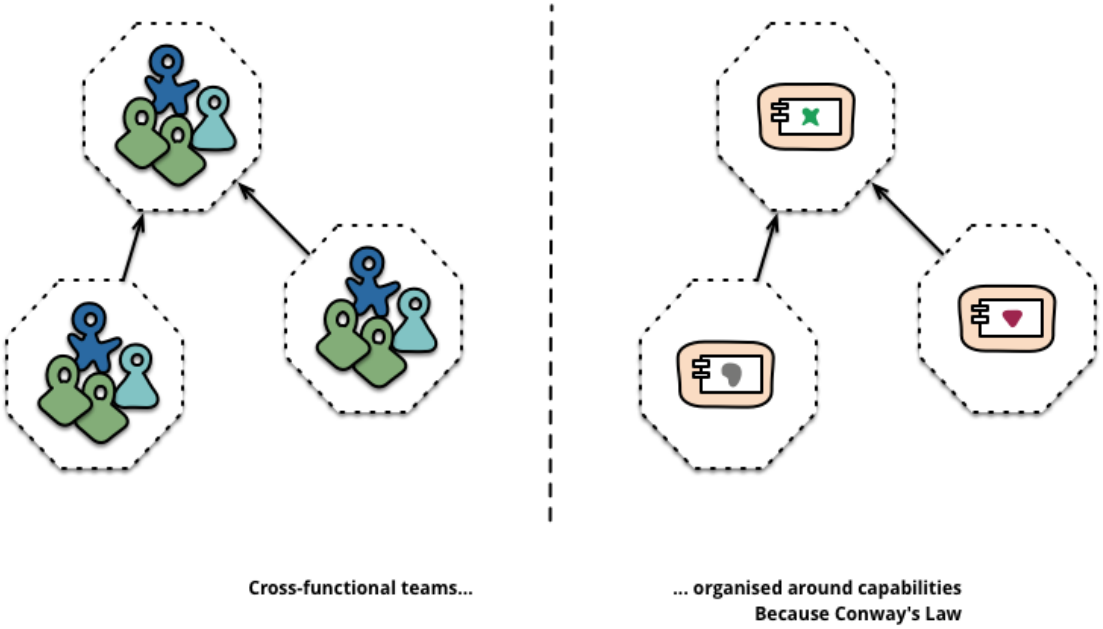

Figure 4. Conway's Law: Common organization in a microservices application [Source: http://martinfowler.com]

Products instead of Projects

Other common characteristic of microservices is the way of an application is developed.

Instead of consider the application as a project and develop it under this model like in most cases, where the purpose is to deploy or deliver a piece of software, which is considered to be finished and handed over to a maintenance organization, *microservice architecture* tends to follow the model of product, which in comparison to project model, the team should own the product over its full lifetime.

Integration of smart endpoints and dumb pipes

Related to monolithic application, the components are executing in-process and communication between them is via either method invocation or function call, in applications build from *microservices* the aim is to be as decoupled and as cohesive as possible following the approach named *Smart endpoints and dumb pipes*. This means that each service is own of their own domain logic, applying logic and producing a response when a request is received. These responses are choreographed using simple *RESTful API*.

Decentralized Governance

Other inconvenience of *monolithic architecture* comes as a consequence of tending to standardize on single technology platform due to governance centralization. Thus, there are no possibilities to fit a proper solution for a concrete problem.

For this reason, *microservice architecture* gives the possibility to choose the better solution for each service and each problem, decentralising the governance and being favourable for avoiding *YAGNI (You aren't gonna need it)* dilemma, due to the production of useful solutions.

Decentralized Data Management

To summarise, decentralization of data management means that the data model is differing between systems, that is, each supplied data from the same data model can be displayed in different forms for each system.

In addition to data model decentralizing decisions, *microservice architecture* also decentralises data storage decisions in favour of managing a database per service or an instance of the same database technology. In contrast, *monolithic architecture* prefers a single logical database for data persistence.

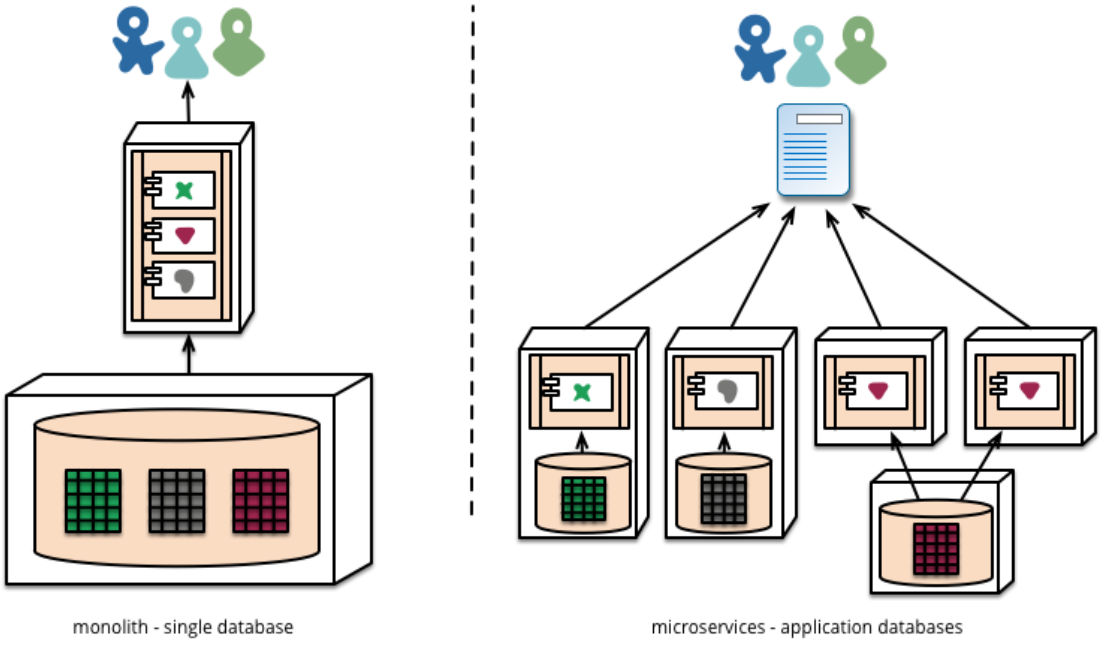

Figure 5. Data management comparison [Source: http://martinfowler.com]

As a result of decentralization of data, microservices should manage with eventual consistency by transactionless coordination between services in comparison to a strong consistency using transactions, which is commonly used in monolithic applications.

Infrastructure Automation and Continuous Delivery

Infrastructure automation and Continuous Delivery also benefit *microservice architecture* by facilitating the deployment of an application in production by delivering small batches of changes.

Also, with the implementation of infrastructure automations and *Continuous Delivery*, the cost of pushing changes to production environments is less than traditional methods like iterative deliveries.

Design for failures

Finally, as a consequence and inconvenience of using *microservice architecture* in contrast to monolithic architecture, the implication of splitting an application as service and as components, the piece of software should be designed to tolerate the failure of service.

For this reason, it is important to implement the ability of detecting a failure as soon as possible and if the situation permits, automatically restore the service. *Microservices* put a lot of emphasis on real-time monitoring, checking both architectural elements and business relevant metrics. During the design of the application, it is also important to take into account stability patterns to improve the stability of the application, using patterns like *Timeouts, Circuit Breakers* and *Bulkheads*.

2.1.2. Service Oriented Architecture (SOA) and Microservices

Service Oriented Architecture is described by OASIS group as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations [11].

Also, Open Group defines *SOA* as an architectural style that supports service-orientation, being a way of thinking in terms of service and service-based development and the outcomes of services [12].

When trying to relate *microservices architecture* to *Service Oriented Architecture* or *SOA*, different controversial ideas come out. For this reason, it is being considered to include a brief point of view about this.

For instance, Martin Fowler says that both are very similar or service orientation done right, but the problems arrive when *SOA* means too many different things and it comes with the focus of ESBs used to integrate monolithic applications.

Such as Martin Fowler, Sergio Maurenzi, is other author who is in the same way and states that *SOA* and *microservices* are similar by definition but *Service Oriented Architecture* has been obfuscated by ESB.

Both opinions help to understand that *microservice architecture* is distinct from *Service Oriented Architecture* in the form of integrated various services. In the first case, services are independent instead of integrating services in one application like the second case.

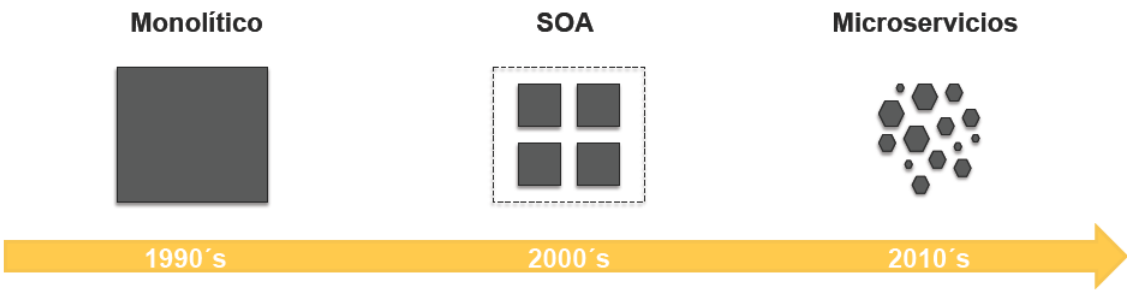

Figure 6. Evolution of architectures [Source: http://sergiomaurenzi.blogspot.com.es]

2.1.3. Conclusions

With the aim of finishing this case study, I like to leave a brief opinion and point of view about the terms exposed here.

First of all, I see *microservices* as architecture for present and future solution in really big applications, and there are really big examples of companies, which decide to use it like Amazon, Netflix and Microsoft. I mean, probably for a small business is not necessary to follow this style of architecture and a monolithic solution is the best way to follow, but when someone is thinking to develop a solution in which the application is the important part of your business, probably the *microservice architecture* is a good solution due to the big advantages and characteristics exposed before.

In addition to this, at the point where the application is growing, I think that it can be beneficial a possible implantation of *microservices* due principally, to its own granularity. A service is a small piece of software in which the development is less complex compared with a monolithic solution. Also, having a highly distributed systems like Internet, you can deploy a service as you need it, supposing a benefit in economical factors, due to the possibility of deploying a small service instead of entire applications, as happened in a monolithic application.

Nevertheless, *microservice architecture* is not as bright as its followers want to expose. There is some criticism for this architecture and one of them is that architecture introduces additional complexity and new problems to deal with, such as network latency, load balancing and fault tolerance ignoring that one of these belongs to the *fallacies of distributed computing*.

2.2. Docker

This second of two case studies is focused on the trendy technology called *Docker*. *Docker* and *Microservices*, the main term studied in the first case study, are highly related.

Docker is defined in the official web page as an open platform for developing, shipping, and running applications providing the possibility of separating applications from infrastructure and treats your infrastructure like a managed application, helping to ship the code faster, test faster, deploy faster, and shorten the cycle between writing code and running code [13].

This is done by the combination of kernel containerization features with workflows and tooling that helps manage and deploy applications. Also, it provides a way to run almost any application securely isolated in a container. The isolation and security allow running many containers simultaneously on a *Docker host*. The lightweight nature of containers, which run without the extra load of a hypervisor, means you can get more out of hardware.

In addition, *Docker* can help in different ways, for example, getting an application as components into *Docker* containers, this can aim following *microservices architecture*, topic that will be discussed at the end of this case study. Moreover, this technology also can help distributing and shipping containers whether local datacentre or the Cloud.

2.2.1. Dedicated Servers, Virtual Machines and Containers

As a brief lesson of history, it is possible to follow the evolution of application development and deployment until nowadays. Traditionally someone who wanted to deploy an application was practically obligated to pay for an entire *dedicated server* or VPS, normally with some vendors lock in. That supposes one application on one physical server, which means slow deployment times, huge costs and wasted resources with the addition of difficulties to scale and migrate.

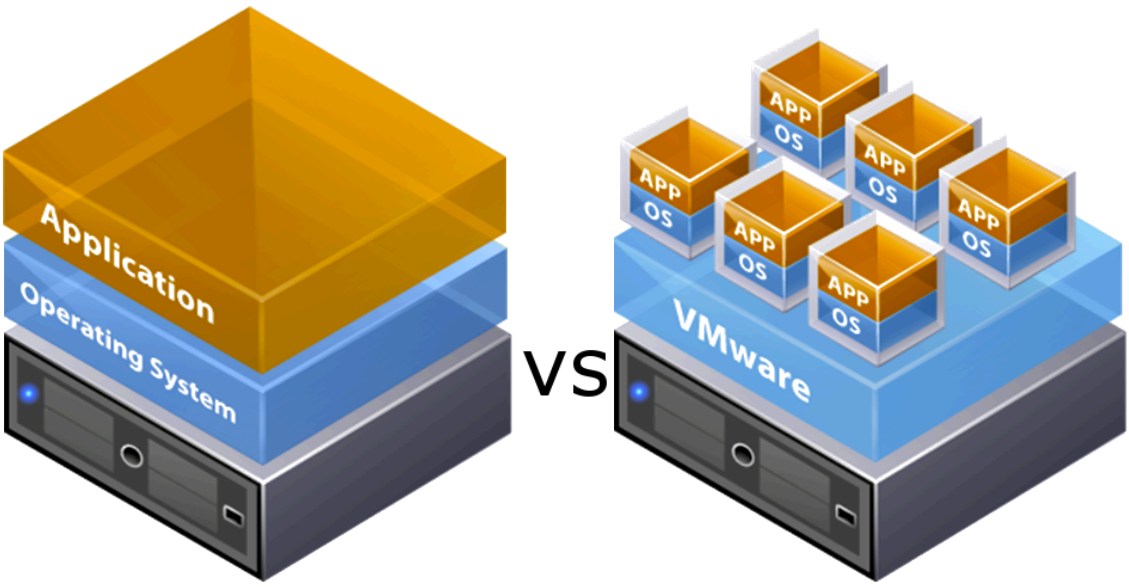

Figure 7. Dedicated server versus Virtual Machine [Source: http://www.ophtek.com]

The natural evolution was the well known *Hypervision-based Virtualizations* technologies or *Virtual Machines*. Nowadays is a consolidated technology in al mayor vendors due to the possibilities of containing multiple applications in one physical server where each run in a VM, where an user can provision as machines as they like and pay as they go.

A VM come with big benefits comparing with dedicated servers. There is a better resource pooling and as commented before, one physical machine is divided into multiple VM, being easy to scale due to the rapid elasticity of *Hypervision-based Virtualizations*.

However, virtualization technologies have some limitations. Each VM, still requires CPU allocation, storage, RAM and entire guest operating system, which mean wasted resource. Also, the more *Virtual Machines*, the more resources are needed and application portability is not guaranteed.

Finally, there is a new technology as a part of an evolution for deploying applications. This technology is the containerization that offers *Docker*. A containerization or *container-based virtualization* uses the host operating system kernel to run multiple guest instances or containers.

Each container is isolated and has its own root file system, processes, memory, devices and networks ports. Besides these characteristics, the containerization gives the possibility of using different versions of the same software or application.

In contrast to VMs, containers are more lightweight and they do not need to install a guest operating system, supposing less CPU, RAM and space requirements. As a result, it is possible to deploy more containers per machine than *Hypervision-based Virtualizations* with greater portability possibilities.

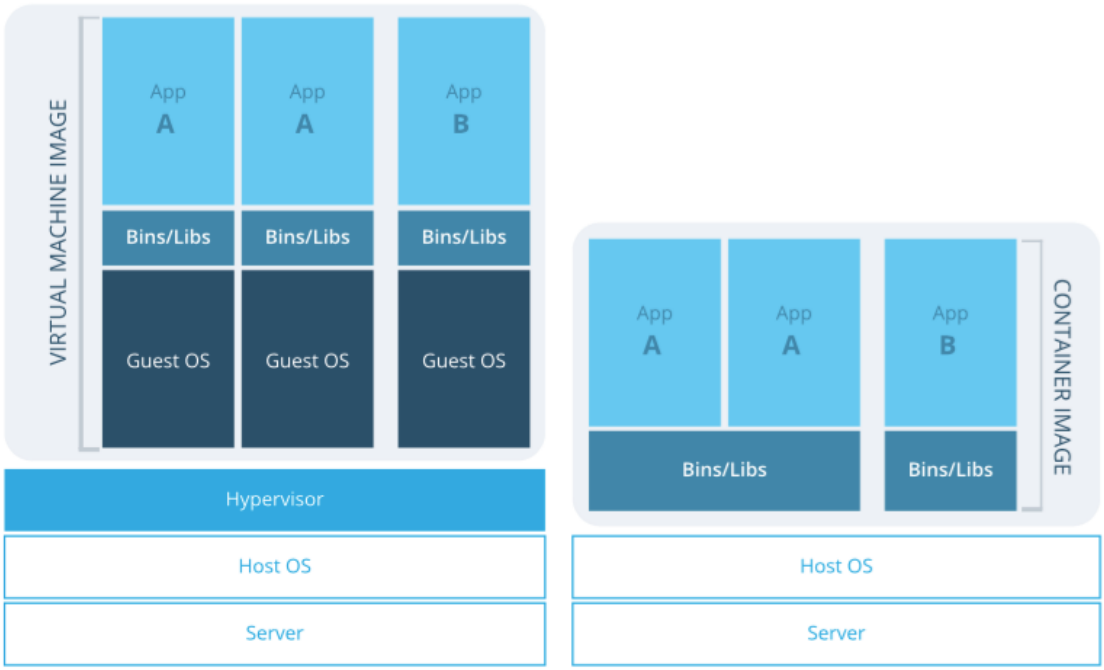

Figure 8. VM versus Containerization [Source: https://www.openstack.org]

This supposes that containerization with *Docker* reveals some benefits. First, it helps developer to focus on building their applications and system administrator's focusing on deployment due to *separation of concerns*. Second, development cycles are faster than in VMs or traditional ways. Third, the application portability gives the possibility of building in one environment and ship to another and finally, it is easier to scale and spin up new containers if needed.

2.2.2. Docker Orchestration tools and components

Docker Orchestration tools, as the name implies, are tools for orchestrating distributed applications with Docker:

- Docker Engine
- Docker Machine
- Docker Swarm
- Docker Compose
- Docker Registry and Docker Hub

Docker Engine

Docker Engine is a fundamental component, lightweight runtime and containerization platform that uses Linux kernel namespaces, which gives the

isolated workspace, and control group from a *Docker host*, enabling containers to be built, shipped and run.

Container is an isolated and secure application platform that holds everything that is needed for an application to run being built from one or multiple Docker images, a read-only template created by a user. A *Docker image* can contain for example an operating system or a web application, even though, a game server like Minecraft can be hold in a *Docker Registry* like Docker Hub.

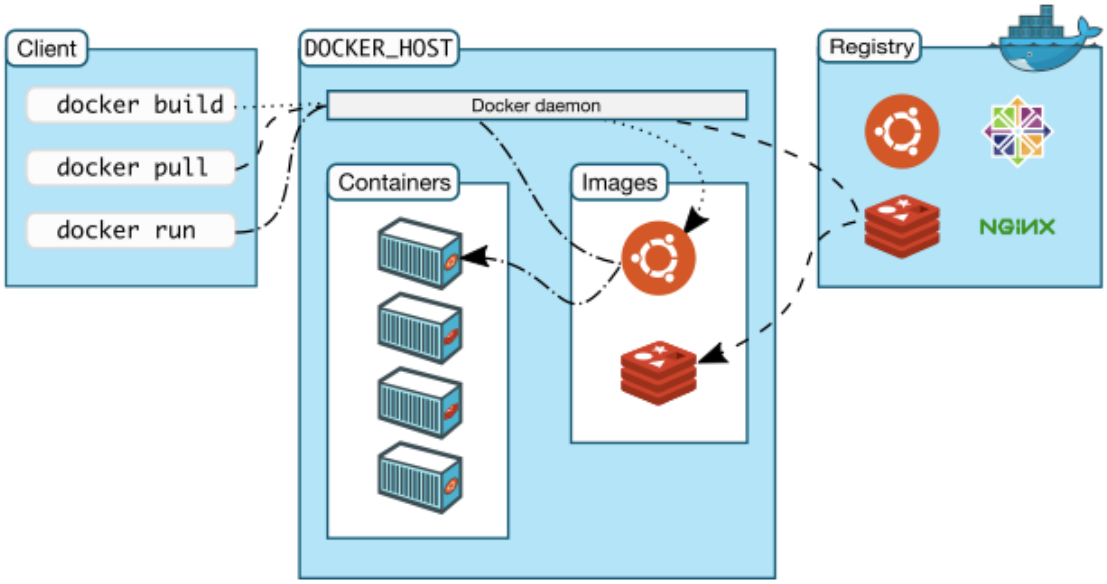

Figure 9. Docker platform workflow [Source: https://www.docker.com]

Docker Registry and Docker Hub

Docker Registry is defined by Docker official documentation as stateless and highly scalable server side applications that stores and lets distribute Docker images. These are public or private stores from which upload or download images. Docker registries are the distribution component of Docker.

It is recommended using the Registry for tightly controlling where your images are being stored, fully own images distribution pipeline and integrate image storage and distribution tightly into in-house development workflow.

Docker Hub instead, is the public *Docker Registry*. It serves a huge collection of existing images ready for use created by users.

Docker Machine

Docker Machine is a tool that automatically provisioning *Docker hosts* and install *Docker Engine* on them providing more efficient processes.

Docker Swarm

Docker Swarm is a native clustering tool that clusters *Docker hosts* and schedules containers using scheduling strategies, also turns a pool of host machines into a single virtual host because *Docker Swarm* serves the standard *Docker Remote API* for communicating with a Docker daemons to transparently scale to multiple hosts.

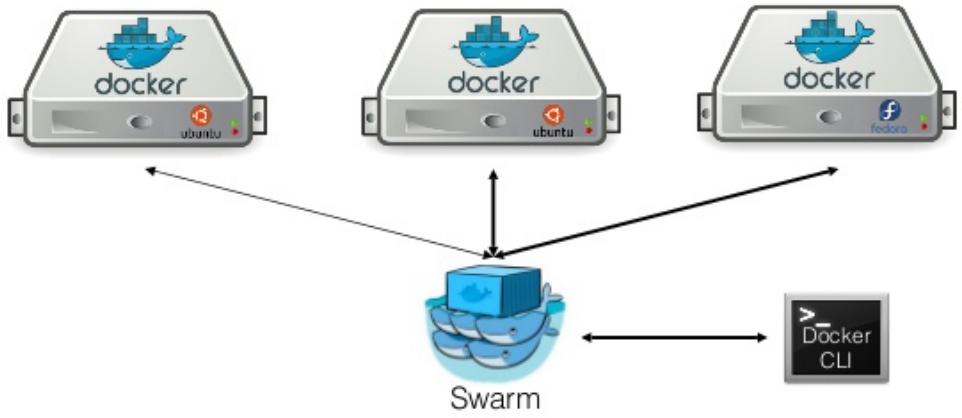

Figure 10. Docker Swarm clustering [Source: https://www.docker.com]

Also, it supports many discovery back-ends like *etcd* from *CoreOS*, *Consul* and *Zookeeper* for managing services in containers.

Docker Compose

Docker Compose is a tool for creating and managing multi-container applications where containers are all defined and linked in a single file spinning up all containers in a single command. Each container runs a particular component or service of an application.

Additionally, *Docker Compose* comes with some benefits, such as a quick recap on linking containers and the possibility of accessing to data from source containers in recipient containers.

2.2.3. Microservice Architecture and Docker

As commented at the beginning of this case study, *microservice architecture* and *Docker* are closely related.

The context come when the *microservice architectural style* explained in the correspondent case study is applied and architected the system as a set of services or small building blocks. Each service is independently deployed as a set of isolated service instances for throughput and availability, which are scalable and written using a variety of languages, frameworks and framework versions. Also, each service must be able to build quickly and deploy it as cost-effectively as possible.

Due to the reasons explained above, the solution comes through to package the service as a *Docker image* and deploy each service instance as a Docker container.

A common *microservice architecture* deployed with *Docker* can be seen as following diagram in the *Figure 11*.

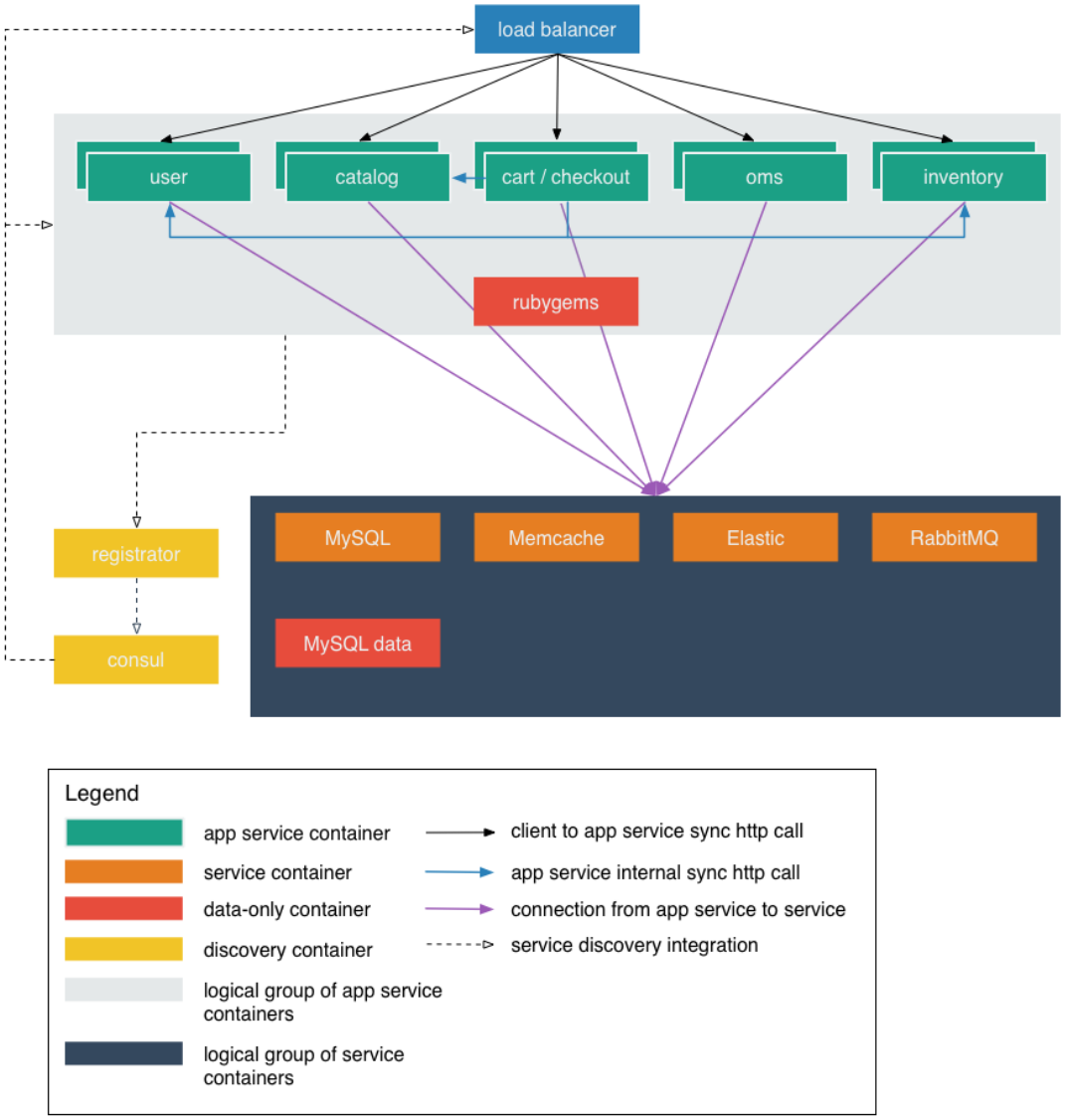

Figure 11. Application developed under microservices and Docker [Source: http://howtocookmicroservices.com]

At this diagram, it is possible to see a logical group of app service containers that provides all logic from a business as a user service, catalogue service and others. All of them are linked to a logical group of infrastructure service containers. Finally, each app service container is integrated with a service discovery like Consul for discovering each other and a load-balancer to guide the client http calls to the correspondent app service registered by the service discovery. Also, in a distributed system where scalability is needed, a host discovery or orchestration tool like *Docker Swarm* for scheduling containers is fundamental for a correct host clustering.

As consequence, this approach includes some benefits like a straightforward way to scale up and down a service by changing the number of container instances, where each container encapsulates the technology used to build the service and imposes limits on the CPU and memory consumed. Likewise, containers are extremely fast to build and start in comparison to other deployment solutions like VM, which need an entire operating system and are slower to start running.

Furthermore, it is easier for developers to understand, because they only have to focus on their services and the whole app does not have to be committed to one technology stack. If one service goes down, application should still run, albeit with reduces functions.

2.2.4. Conclusions

In conclusion, *Docker* is becoming an extremely popular way of packaging and deploying services and it goes hand in hand with *microservice architecture*. Each service from an application is packaged as a Docker image and each service from an instance is a Docker container. Thanks to *Docker* and *microservices*, *distributed system* are easier to deploy and several clustering frameworks and tools like *Kubernetes, Marathon/Mesos, Amazon EC2 Container Service, CoreOS, OpenStack* and others have been coming out.

Moreover, different companies like Amazon or 1&1 are firmly betting on the future of containerization intimating that is a technology to take into account.

In contrast, like *microservice architecture, Docker* comes with an addition of complexity when dealing with it, supposing some disadvantages. The most common disadvantage when working with containers is the necessity of having everything properly controlled and monitored for troubleshooting any problem that can occur.

3. Implementation of a brief use case

After looking at the theory, it comes the moment to use all learned terms in a brief use case. The use case itself, is a small system that is going to implement distributed and containerized microservice architecture to scrap data and serve collected data on different endpoints. This means that each service is implemented as a container, being independent from the other service. For this reason, each service is going to benefit for all of advantages from containerized technology and microservice architectural style.

3.1. Design and definition of the architecture

First of all, it is to define some essential considerations that the system must have. The application must be simple to be focused on what each service is going to do. Also, a service must have a high cohesion and a low coupling for helping in the scalability of the application. In addition, communication between services is made with a *RESTful API* using *JSON* as data type for a simple connection.

Once taking the essential considerations as starting point, the *Figure 12* shows the global view of the infrastructure architecture.

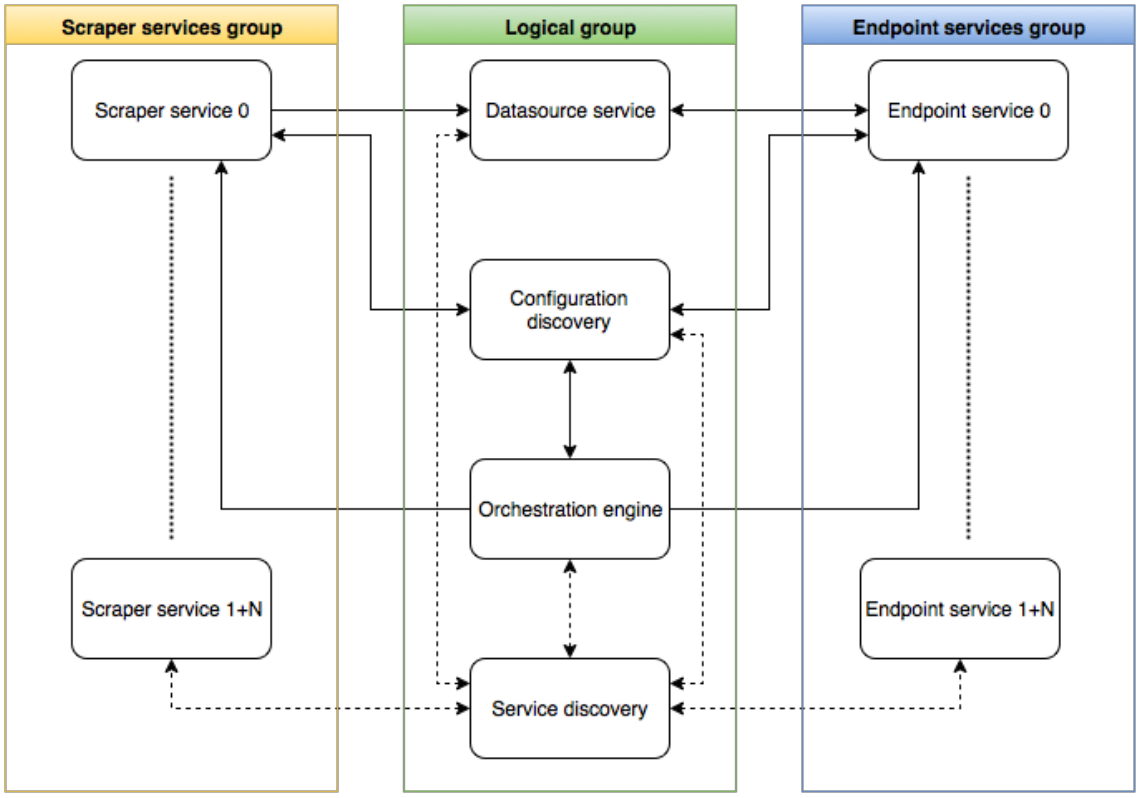

Figure 12. Architecture of distributed data scraper system

As well as shown above, it is possible to distinguish three logical groups where every box is a container.

- **Scraper services group** is a finite group of containers of scrapers deployed in the distributed system. Each scraper will collect a concrete data from an external data source like an API or a web page. The collected data is stored in the internal data source.
- **Logical group** that include the main services for the correct running of the entire system where it can be differentiated different services:
    - *Datasource service* is for storing data collected by deployed scrapers and processed by a Map-Reduce function for transforming each data in compressible and usable data collections.
    - *Configuration-discovery service* is the responsible for managing configurations for each scraper or endpoint services, which is needed due to the automated deployment of scrapers and endpoints. Each new deployed service is going to ask the configuration discovery API for a free configuration when service is started.
    - *Orchestration-engine service* in contrast, is a service on top of the service discovery container and it is responsible for deploying a new scraper or endpoint service when a new configuration is detected.
    - *Service discovery* is and important service for having a control of deployed containers or services in the system.
- **Endpoint services group** like *Scraper services group*, is a finite group that depends on the number of collections processed by the *Datasource service*. One collection, one endpoint. Each container instance for this group has a public API for requesting collected data.

3.2. Development of the distributed data scraper system

First of all, before the explanation of each service evolved in the system, I am obliged to explain the considerations taken during the development.

The main programming language I choose is *Node.js* for the simple reason that *Node.js* is *JavaScript* in the server and it is natively part of web technologies. This is

going to make the development of the system easier, in particular the *Scraper service*.

In addition to *Node.js*, *MongoDB* is going to be used for data storage and manipulation for the reason that uses *JSON*, a native *JavaScript* notation for data transmission. At last, *Redis* is other data store that I have implemented as a cache for maintaining *scraper service* and *Endpoint service* independent of other services in case of failure.

Also, I decided to store all the code on *BitBucket*, a *Git* repository, which give all advantages of a distributed version control system and also helps to automatize creations of *Docker images* for a posterior use in the deployment phase.

Furthermore, for each container or service, it is necessary to define a stack of technologies and the interaction between them. An image is worth thousand words and the diagram at *Figure 13* shows it in more depth.

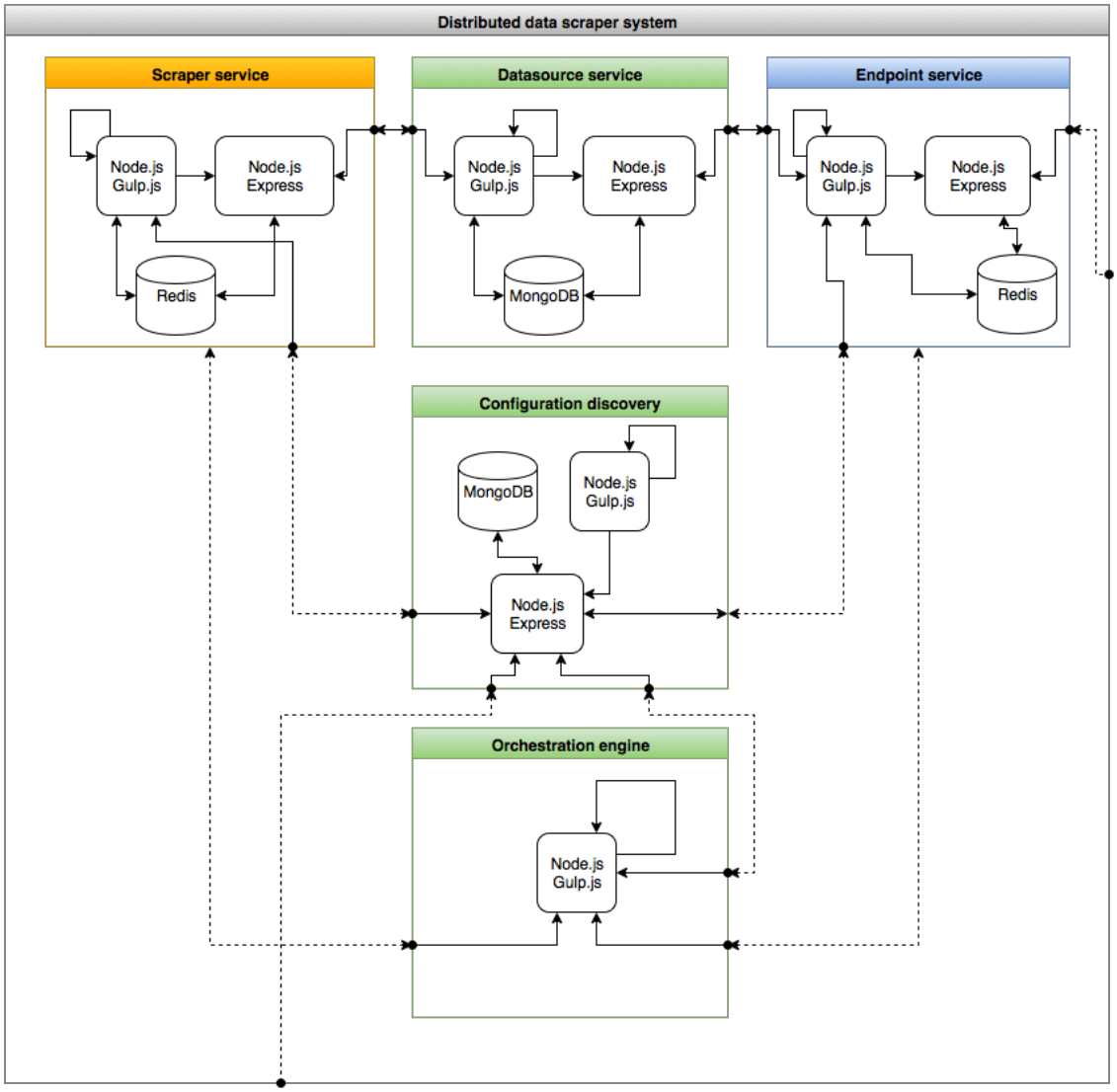

Figure 13. Distributed data scraper in detail

3.2.1. Scraper service

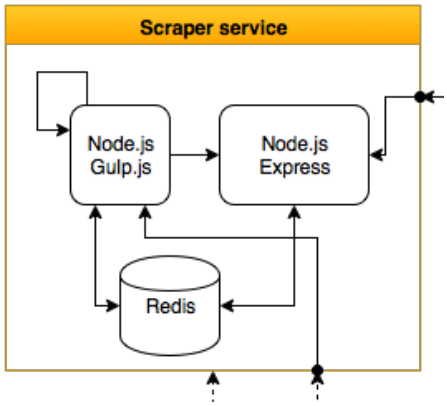

Figure 14. Scraper service

Scraper service is a service of the system developed with *Node.js* that scraps data from an external datasource for storing this new data in the *Datasource service*. For

this labour, a *Gulp* task is programmed for scraping configured data every specified time in the configuration. *Gulp* is a *Node.js*-based task runner.

Configuration is requested from *Configuration-discovery service* when the container is started. Once the configuration is received, this is stored in an in-memory data structure store called *Redis*.

The process of scraping the new data is done by the implementation of *X-Ray,* a Node.js package that fits with the aim of scraping from a web page. Obtained data from this process, is mapped and sent to *Datasource service* using a *RESTful API* implemented with *Express* by *Datasource service*.

Furthermore, the best way to understand what this service does, is to examine the written code.

```
var gulp = require('gulp');
var nodemon = require('gulp-nodemon');
var request = require('request-promise');

// Configuration
var containerHostname = process.env.HOSTNAME || false;
var consulUrl = 'http://217.160.15.75:8500/v1';
var redisHost = process.env.REDIS_PORT_6379_TCP_ADDR || '127.0.0.1';
var redisPort = process.env.REDIS_PORT_6379_TCP_PORT || 6379;

// Components
var configuration = require('./lib/config')(redisPort, redisHost);
var scraper = require('./lib/scraper');

/**
 * Set configuration obtained from configuration discovery service
 */
gulp.task('set-config', function () {
    return request({
        "uri": consulUrl + '/catalog/service/configuration-discovery',
        json: true // Automatically stringifies the body to JSON
    }).then(function (services) {
        var configurationDiscoveryAddr = services[services.length -
1].ServiceAddress;
        var configurationDiscoveryPort = services[services.length -
1].ServicePort;
```

```
        var query = 'limit=1';
        if (containerHostname) {
            query += '&container=' + containerHostname;
        }

        return request({
            "uri": 'http://' + configurationDiscoveryAddr + ':' +
configurationDiscoveryPort + '/configurations?' + query,
            "json": true
        });
    }).then(function (config) {
        return configuration.set(config[0]);
    });
});

/**
 * Scrap data
 */
gulp.task('scrap-data', ['set-config'], function () {
    configuration.get().then(function (config) {
        var config = JSON.parse(config);
        var scrap = (function scrap(config) {
            scraper(config).get().then(function (scrapedData) {
                console.info('[' + new Date().toISOString() + '] Data properly
scraped...');
                console.info('[' + new Date().toISOString() + '] Data sent to
datasource...');

                return request({
                    "uri": consulUrl + '/catalog/service/datasource',
                    json: true // Automatically stringifies the body to JSON
                }).then(function (datasources) {
                    var datasourceAddr = datasources[datasources.length -
1].ServiceAddress;
                    var datasourcePort = datasources[datasources.length -
1].ServicePort;

                    return request({
                        "method": 'POST',
                        "uri": 'http://' + datasourceAddr + ':' +
datasourcePort + '/items',
                        "body": {
                            "collection": config.collection,
```

```
                            "data": scrapedData
                        },
                        json: true // Automatically stringifies the body to
JSON
                    })
                }).then(function (response) {
                    if (response.ok) {
                        console.info('[' + new Date().toISOString() + ']
Sent!');
                    }
                    else {
                        console.warn('[' + new Date().toISOString() + ']
Something wrong happens...');
                    }
                }).catch(function (e) {
                    console.error('[' + new Date().toISOString() + '] ' +
e.message);
                });
            }).catch(function (e) {
                console.error('[' + new Date().toISOString() + '] ' +
e.message);
            });

            // Set timeout for scraping again.
            console.info('[' + new Date().toISOString() + '] Scrap timeout
setted...');
            setTimeout(function () {
                scrap(config);
            }, parseInt(config.ttl, 10) * 1000);
        })(config);
    });
});

/**
 * Start RESTful API
 */
gulp.task('start', function () {
    nodemon({script: 'app.js', legacyWatch: true});
});

gulp.task('default', ['set-config', 'scrap-data', 'start']);
```

In addition to this, for helping the implementation of the *gulpfile*, the service has two components that I developed to wrap *Redis* and *X-Ray* packages.

Respectively, the first one looks like.

```
/**
 * Auto-config module
 */
var Config = module.exports = function (port, host) {
    this.promise = require('bluebird');
    this.redis = this.promise.promisifyAll(require("redis"));
    this.host = host || '127.0.0.1';
    this.port = port || 6379;
    this.client = this.redis.createClient(port, host);

    if (!(this instanceof Config)) return new Config(this.port, this.host);
};

/**
 * Get configuration from redis
 */
Config.prototype.get = function () {
    return this.client.getAsync('config');
};

/**
 * Set configuration in redis
 *
 * @param config
 * @returns {*}
 */
Config.prototype.set = function (config) {
    return this.client.setAsync('config', JSON.stringify(config));
};

/**
 * Get Redis client
 */
Config.prototype.del = function () {
    return this.client.del('config');
};
```

And the second one instead.

```
/**
 * Scraper module
 * Wrapper for better usage of the X-ray module
 *
 * @author Imanol Urra Ruiz
 */
var Scraper = module.exports = function (config) {
    this.promise = require('bluebird');
    this.config = config;
    this.xray = require('x-ray')();

    if (!(this instanceof Scraper)) return new Scraper(config);
};

Scraper.prototype.get = function () {
    var config = this.config;

    return new this.promise(function (resolve, reject) {
        this.xray(config.scraper.source, config.scraper.scope,
config.scraper.selector)
            .paginate(config.scraper.paginate)
            .limit(config.scraper.limit)(function (err, data) {
                if (err) {
                    reject(err);
                }
                resolve(data);
            });
    });
};
```

Once the service is developed, a *Dockerfile* is created and *Docker Hub* automates *Docker image* creation when code is pushed to *Git* repository.

```
FROM node:latest

RUN mkdir /usr/src/app/ && npm install gulp -g

WORKDIR /usr/src/app
COPY app/package.json /usr/src/app/package.json

RUN npm install

COPY app/ /usr/src/app
```

```
EXPOSE 3000

CMD [ "npm", "start" ]
```

3.2.2. Datasource service

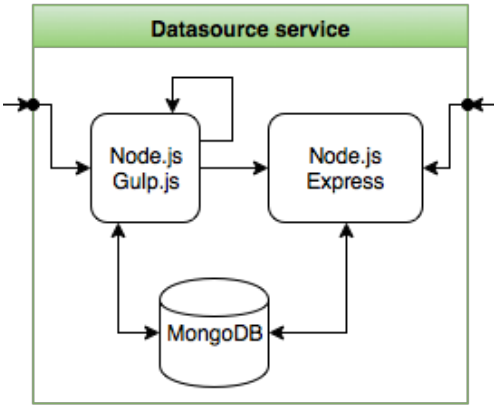

Figure 15. Datasource service

Datasource service also implements a *RESTful API* with *Express,* where data sent by a *Scraper service* is stored in a capped collection in *MongoDB*.

On the other hand, a *gulpfile* is implemented as well, where some tasks are developed. Then, one of these tasks has as an objective to wait with an opened *MongoDB tailable cursor* for new data to be inserted in the capped collection and when data is detected, is copied in a temporal collection that is mapped and reduced with the aggregation framework from *MongoDB* every specific time for another task. As a result of this reduction, a new final collection is created ready for Endpoint service.

Again, it is a good way to analyse the written code for a better understanding.

```
var gulp = require('gulp');
var nodemon = require('gulp-nodemon');

var mongoUrl = process.env.MONGO_PORT ? process.env.MONGO_PORT.replace('tcp',
'mongodb') : 'mongodb://localhost:27017';
process.env.CD_CONFIG_MONGODB_URL = mongoUrl;

var Promise = require('bluebird');
```

```
/**
 * Check database.
 */
gulp.task('database-check', function () {
    // Client configuration
    var client = require('mongodb').MongoClient.connect(mongoUrl +
'/scrapr_pro');

    client.then(function (db) {
        console.info('[' + new Date().toISOString() + '] Database connection
checked');

        db.close();
    }).catch(function (e) {
        console.error('[' + new Date().toISOString() + ']' + e.message);
    });
});

gulp.task('datasources', function () {
    var client = require('mongodb').MongoClient.connect(mongoUrl +
'/scrapr_pro');

    client.then(function (db) {
        db.dropCollection('capped_datasources');
        db.createCollection('capped_datasources', {capped: true, size:
100000}).then(function (cappedCollection) {
            var cursorStream = (function cursorStream(cappedCollection) {
                cappedCollection.find().sort({$natural: -
1}).limit(1).hasNext().then(function (hasNext) {
                    if (hasNext) {
                        console.info('[' + new Date().toISOString() + ']
Tailable cursor started');

                        var stream = cappedCollection.find()
                            .addCursorFlag('tailable', true)
                            .addCursorFlag('awaitData', true)
                            .setCursorOption('numberOfRetries', -1)
                            .stream();

                        // If there is a stream, then insert received data to
temporal data
                        stream.on('data', function (data) {
                            db.collection('tmp_' + data.collection)
                                .insertMany(data.data)
```

```
                                .then(function (result) {
                                    console.info('[' + new
Date().toISOString() + '] New data inserted to tmp_' + data.collection + '!');
                                })
                                .catch(function (e) {
                                    console.error(e.message);

                                    // Create cursor again if there is an
error
                                    setTimeout(function () {
                                        cursorStream(cappedCollection);
                                    }, 5000);
                                });
                        }).on('error', function (error) {
                            console.error(error);
                        }).on('close', function () {
                            console.info('[' + new Date().toISOString() + ']
Tailable cursor closed');

                            // Create cursor again if there is an error
                            setTimeout(function () {
                                cursorStream(cappedCollection);
                            }, 5000);
                        });
                    }
                    else {
                        console.info('[' + new Date().toISOString() + '] There
is no data in datasource stream already');

                        setTimeout(function () {
                            cursorStream(cappedCollection);
                        }, 5000);
                    }
                }).catch(function (e) {
                    console.error('[' + new Date().toISOString() + ']' +
e.message);

                    setTimeout(function () {
                        cursorStream(cappedCollection);
                    }, 5000);
                });
            })(cappedCollection);
        }).catch(function (e) {
            console.error('[' + new Date().toISOString() + ']' + e.message);
```

```
        });
    }).catch(function (e) {
        console.error('[' + new Date().toISOString() + ']' + e.message);
    });
});

/**
 * Set up cron for mapReduce.
 */
gulp.task('aggregation', function () {
    var client = require('mongodb').MongoClient.connect(mongoUrl +
'/scrapr_pro');

    client.then(function (db) {
        var runAggregate = (function runAggregate(db) {
            db.listCollections({name: {$regex:
/(tmp_)/i}}).toArray().then(function (tmpCollections) {
                if (tmpCollections.length > 0) {
                    console.log('[' + new Date().toISOString() + ']
Aggregation procedure have been started...');

                    tmpCollections.forEach(function (tmpCollection) {
                        var endCollection =
tmpCollection.name.split('_').pop();

                        db.collection(tmpCollection.name).aggregate([
                            {
                                "$group": {
                                    "_id": "$title",
                                    "items": {
                                        "$addToSet": "$item"
                                    }
                                }
                            },
                            {
                                "$project": {
                                    "_id": "$_id",
                                    "title": "$_id",
                                    "items": "$items",
                                    "timestamp": {"$add": [new Date()]}
                                }
                            },
                            {
                                "$out": endCollection
```

```
                            }
                        ]).toArray().then(function (result) {
                            console.info('[' + new Date().toISOString() + ']
Aggregation have been finished and output to: ' + endCollection);

                            setTimeout(function () {
                                runAggregate(db);
                            }, 2 * 60 * 1000);
                        }).catch(function (e) {
                            console.error('[' + new Date().toISOString() + ']
' + e.message);

                            setTimeout(function () {
                                runAggregate(db);
                            }, 2 * 60 * 1000);
                        });
                    }
                    else {
                        console.info('[' + new Date().toISOString() + '] Nothing
to aggregate...');

                        setTimeout(function () {
                            runAggregate(db);
                        }, 2 * 60 * 1000);
                    }
                });
            })(db);
        }).catch(function (e) {
            console.error('[' + new Date().toISOString() + '] ' + e.message);
        });
});

gulp.task('start', function () {
    nodemon({script: 'app.js', legacyWatch: true});
});

gulp.task('default', ['database-check', 'datasources', 'aggregation',
'start']);
```

Also, a *Dockerfile* must be created for a posterior usage of this service.

```
FROM node:latest

RUN mkdir /usr/src/app/ && npm install gulp -g
```

```
WORKDIR /usr/src/app
COPY app/package.json /usr/src/app/package.json

RUN npm install

COPY app/ /usr/src/app

EXPOSE 3001

CMD [ "npm", "start" ]
```

3.2.3. Endpoint service

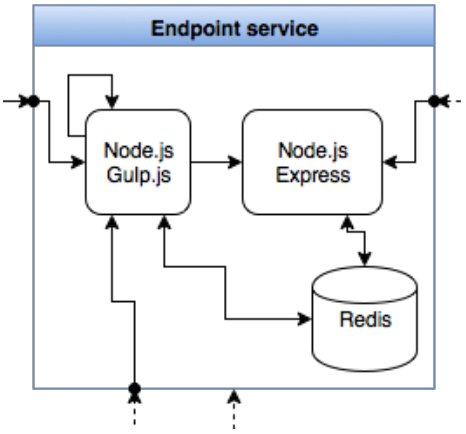

Figure 16. Endpoint service.

Endpoint service is technically similar to *Scraper service*, instead of getting data from an external data source, gets processed and reduced data from a collection stored in *MongoDB* on *Datasource service*. Retrieved data is exposed to the public via *RESTful API* build with *Express*.

Like *Scraper service*, *Redis* is implemented to save service configuration and to cache data obtained from *Datasource service*. This decision is to maintain a *high availability* and *eventual consistency* if the *Datasource service* is down or not.

As well as the rest of services, *Endpoint service* also implements a *gulpfile*, which is executed when service starts, to get configuration from *Configuration-discovery service*, and also retrieve data from *Datasource service* to be served to the public via *RESTful API*.

```
// Requires
var gulp = require('gulp');
var nodemon = require('gulp-nodemon');

// Configuration
var containerHostname = process.env.HOSTNAME || false;
var consulUrl = 'http://217.160.15.75:8500/v1';
var redisHost = process.env.REDIS_PORT_6379_TCP_ADDR || '127.0.0.1';
var redisPort = process.env.REDIS_PORT_6379_TCP_PORT || 6379;
var configuration = require('./lib/config')(redisPort, redisHost);
var datasource = require('./lib/datasource')(redisPort, redisHost, consulUrl);

/**
 * Retrieve configuration form Configuration-disvocery service
 */
gulp.task('set-config', function () {
    return request({
        "uri": consulUrl + '/catalog/service/configuration-discovery',
        json: true // Automatically stringifies the body to JSON
    }).then(function (services) {
        var configurationDiscoveryAddr = services[services.length -
1].ServiceAddress;
        var configurationDiscoveryPort = services[services.length -
1].ServicePort;

        var query = 'limit=1';
        if (containerHostname) {
            query += '&container=' + containerHostname;
        }

        return request({
            "uri": 'http://' + configurationDiscoveryAddr + ':' +
configurationDiscoveryPort + '/configurations?' + query,
            "json": true
        });
    }).then(function (config) {
        return configuration.set(config[0]);
    });
});

/**
 * Get and update scraped data from datasource service
 */
gulp.task('update-datasource', ['set-config'], function () {
```

```
    configuration.get().then(function (config) {
        var config = JSON.parse(config);
        var updateDatasource = (function updateDatasource(config) {
            datasource.getFromDatasource(config.collection).then(function
(response) {
                console.info('[' + new Date().toISOString() + '] Updating
datasource...');

                return datasource.set(response);
            }).then(function (result) {
                if (result === 'OK') {
                    console.info('[' + new Date().toISOString() + ']
Updated!');
                } else {
                    console.warn('[' + new Date().toISOString() + '] Something
wrong updating endpoint...');
                }
            }).catch(function (e) {
                console.error('[' + new Date().toISOString() + '] ' +
e.message);
            });

            // Set timeout for update data
            console.info('[' + new Date().toISOString() + '] Update timeout
setted...');
            setTimeout(function () {
                updateDatasource(config);
            }, parseInt(config.ttl, 10) * 1000);
        })(config);
    }).catch(function (e) {
        console.error('[' + new Date().toISOString() + '] ' + e.message);
    });
});

/**
 * Start RESTful API
 */
gulp.task('start', function () {
    nodemon({script: 'app.js', legacyWatch: true});
});

gulp.task('default', ['set-config', 'update-datasource', 'start']);
```

Like other services developed in the system, *gulpfile* uses a component that I implement to wrap storage functionalities for data retrieved from *Datasource service*.

```
/**
 * Datasource component
 */
var Datasource = module.exports = function (port, host, consulUrl) {
    this.promise = require('bluebird');
    this.redis = this.promise.promisifyAll(require('redis'));
    this.host = host || '127.0.0.1';
    this.port = port || 6379;
    this.client = this.redis.createClient(port, host);
    this.request = require('request-promise');
    this.consulUrl = consulUrl || '';

    if (!(this instanceof Datasource)) return new Datasource(this.port,
this.host, this.consulUrl);
};

/**
 * Get configuration from redis
 */
Datasource.prototype.get = function () {
    return this.client.getAsync('datasource');
};

/**
 * Set configuration in redis
 *
 * @param datasource
 * @returns {*}
 */
Datasource.prototype.set = function setDatasource(datasource) {
    return this.client.setAsync('datasource', JSON.stringify(datasource));
};

/**
 * Get Redis client
 */
Datasource.prototype.del = function delDatasource() {
    return this.client.del('datasource');
};
```

```
/**
 * Update datasource from a given collection
 *
 * @param collection
 */
Datasource.prototype.getFromDatasource = function
getFromDatasource(collection) {
    var request = this.request;
    var consulUrl = this.consulUrl;

    return request({
        "uri": consulUrl + '/catalog/service/datasource',
        json: true // Automatically stringifies the body to JSON
    }).then(function (datasources) {
        var datasourceAddr = datasources[datasources.length -
1].ServiceAddress;
        var datasourcePort = datasources[datasources.length - 1].ServicePort;

        return request({
            "uri": 'http://' + datasourceAddr + ':' + datasourcePort +
'/collections/' + collection,
            "json": true
        });
    });
};
```

A *Dockerfile* is written to containerize the service too.

```
FROM node:latest

RUN mkdir /usr/src/app/ && npm install gulp -g

WORKDIR /usr/src/app
COPY app/package.json /usr/src/app/package.json

RUN npm install

COPY app/ /usr/src/app

EXPOSE 3000

CMD [ "npm", "start" ]
```

3.2.4. Configuration discovery

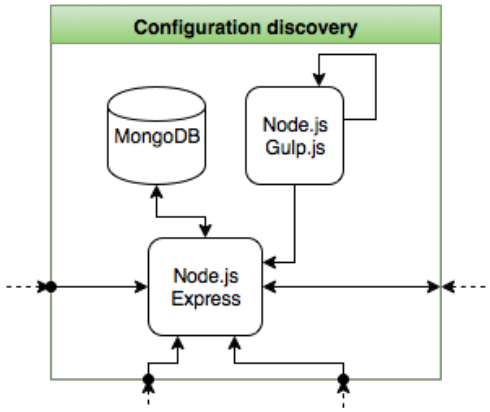

Figure 17. Configuration-discovery service

Configuration-discovery service is based on the pattern called *External configuration store*. Microsoft introduces this pattern as a pattern to move configuration information out of the application deployment package to a centralized location [54].

This pattern can provide opportunities for easier management and control of configuration data also to share configuration data across applications and application instances. This is implemented with a *MongoDB* instance to store configuration in a persistent database where data is sent to services when is requested via *RESTful API*.

Just as *Datasource service*, *Configuration-discovery service* also implements a *MongoDB* capped collection with a tailable cursor that never closes. As a result, the cursor is going to be listening for a new configuration and when a new one is detected, the service adds a scraper configuration and an endpoint configuration based on inserted data from *RESTful API*.

```
var gulp = require('gulp');
var nodemon = require('gulp-nodemon');

var mongoUrl = process.env.MONGO_PORT ? process.env.MONGO_PORT.replace('tcp',
'mongodb') : 'mongodb://localhost:27017';
process.env.CD_CONFIG_MONGODB_URL = mongoUrl;

/**
 * Set up some db configurations.
```

```
 */
gulp.task('database-check', function () {
    // Client configuration
    var client = require('mongodb').MongoClient.connect(mongoUrl +
'/scrapr_pro');

    client.then(function (db) {
        console.info('[' + new Date().toISOString() + '] Database connection
checked');

        db.close();
    }).catch(function (e) {
        console.error(e.stack);
    });
});

/**
 * Create a tailable curos
 */
gulp.task('create-tailableCursor', function () {
    var client = require('mongodb').MongoClient.connect(mongoUrl +
'/scrapr_pro');

    client.then(function (db) {
        db.dropCollection('capped-configurations');
        db.createCollection('capped-configurations', {capped: true, size:
100000}).then(function (cappedCollection) {
            var intervalUntilDocumentFirstDocument = setInterval(function () {
                cappedCollection.find().sort({$natural: -
1}).limit(1).hasNext().then(function (hasNext) {
                    if (hasNext) {
                        clearInterval(intervalUntilDocumentFirstDocument);

                        var stream = cappedCollection.find()
                            .addCursorFlag('tailable', true)
                            .addCursorFlag('awaitData', true)
                            .setCursorOption('numberOfRetries', -1)
                            .stream();

                        stream.on('data', function (config) {
                            console.info('[' + new Date().toISOString() + ']
New scraper configuration detected!');

                            db.collection('configurations')
```

```
                                .insertOne(config)
                                .then(function (result) {
                                    console.info('[' + new
Date().toISOString() + '] New scraper configuration inserted!');
                                })
                                .catch(function (e) {
                                    console.error(e.stack);
                                });

                            db.collection('configurations').find({type:
'endpoint', collection: config.collection})
                                .count()
                                .then(function (count) {
                                    if (count === 0) {
                                        db.collection('configurations')
                                            .insertOne({
                                                type: 'endpoint',
                                                collection: config.collection,
                                                ttl: config.ttl,
                                                container: null
                                            })
                                            .then(function (result) {
                                                console.info('[' + new
Date().toISOString() + '] New endpoint configuration inserted!');
                                            })
                                            .catch(function (e) {
                                                console.error(e.stack);
                                            });
                                    }
                                });
                        }).on('error', function (error) {
                            console.error(error);
                        }).on('close', function () {
                            console.info('[' + new Date().toISOString() + ']
Tailable cursor closed');
                        });

                        console.info('[' + new Date().toISOString() + ']
Tailable cursor started');
                    }
                });
            }, 5000);
        }).catch(function (e) {
            console.error(e.stack);
```

```
        });
    }).catch(function (e) {
        console.error(e.stack);
    });
});

/**
 * Start RESTful API
 */
gulp.task('start', function () {
    nodemon({script: 'app.js', legacyWatch: true});
});

gulp.task('default', ['database-check', 'create-tailableCursor', 'start']);
```

Again, a *Dockerfile* complements the service to allow the service containerization.

```
FROM node:latest

RUN mkdir /usr/src/app/ && npm install gulp -g

WORKDIR /usr/src/app
COPY app/package.json /usr/src/app/package.json

RUN npm install

COPY app/ /usr/src/app

EXPOSE 3003

CMD [ "npm", "start" ]
```

3.2.5. Orchestration engine

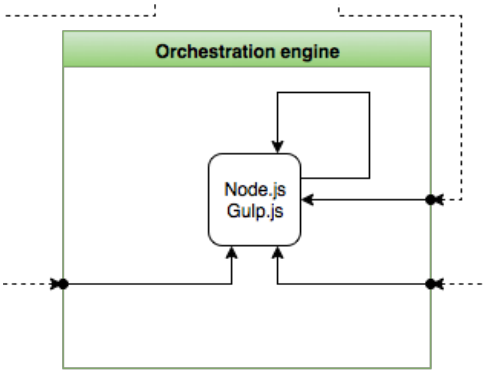

Figure 18. Orchestration-engine service

Finally, *Orchestration engine* is a service that provides orchestration logic for deploying new *Scraper service* and *Endpoint services* on demand when a new configuration from *Configuration-discovery service* is added. For this job, a task from *gulpfile* is checking every few seconds if a new configuration is added and in case of detecting one, a new container is deployed.

```
var gulp = require('gulp');
var nodemon = require('gulp-nodemon');
var fs = require('fs');
var request = require('request-promise');

var consulUrl = 'http://217.160.15.75:8500/v1';

gulp.task('orchestrate', function () {
    request({
        "uri": consulUrl + '/catalog/service/swarm-3376',
        json: true // Automatically stringifies the body to JSON
    }).then(function (swarm) {
        var orchestration = require('./lib/orchestration')({
            "host": swarm[swarm.length - 1].ServiceAddress,
            "port": swarm[swarm.length - 1].ServicePort,
            "protocol": "https",
            "ca": fs.readFileSync('/etc/docker/ca.pem'),
            "cert": fs.readFileSync('/etc/docker/server.pem'),
            "key": fs.readFileSync('/etc/docker/server-key.pem')
        });

        // Start orchestration
        var orchestrate = (function orchestrate(orchestration) {
            var configurationDiscoveryAddr;
            var configurationDiscoveryPort;
```

```
            return request({
                "uri": consulUrl + '/catalog/service/configuration-discovery',
                json: true // Automatically stringifies the body to JSON
            }).then(function (services) {
                configurationDiscoveryAddr = services[services.length -
1].ServiceAddress;
                configurationDiscoveryPort = services[services.length -
1].ServicePort;

                return request({
                    "uri": 'http://' + configurationDiscoveryAddr + ':' +
configurationDiscoveryPort + '/configurations?container=null&limit=1',
                    "json": true
                });
            }).then(function (configurations) {
                if (configurations.length) {
                    configurations.forEach(function (configuration) {
                        orchestration.up(configuration.type,
configuration.collection + '_' + configuration._id).then(function
(containerInfo) {
                            console.info('[' + new Date().toISOString() + '] '
+ containerInfo.Name + ' ' + containerInfo.Config.Hostname);

                            // update configuration
                            console.info('[' + new Date().toISOString() + ']
Updating container field from configuration...');

                            var configurationId = configuration._id;
                            delete configuration._id;
                            configuration.container =
containerInfo.Config.Hostname;

                            request({
                                "method": 'PUT',
                                "uri": 'http://' + configurationDiscoveryAddr
+ ':' + configurationDiscoveryPort + '/configurations/' + configurationId,
                                "body": configuration,
                                "json": true // Automatically stringifies the
body to JSON
                            }).then(function (response) {
                                console.info('[' + new Date().toISOString() +
'] Configuration updated!');
                            }).catch(function (e) {
```

```
                            console.error(e.message);
                        });

                        setTimeout(function () {
                            orchestrate(orchestration);
                        }, 10000);
                    }).catch(function (e) {
                        console.error(e.message);

                        setTimeout(function () {
                            orchestrate(orchestration);
                        }, 10000);
                    });
                }
                else {
                    setTimeout(function () {
                        orchestrate(orchestration);
                    }, 10000);
                }
            }).catch(function (e) {
                console.log(e.message);

                setTimeout(function () {
                    orchestrate(orchestration);
                }, 10000);
            });
        })(orchestration);
    }).catch(function (e) {
        console.error(e.message);
    });
});

gulp.task('start', function () {
    nodemon({script: 'app.js', legacyWatch: true});
});

gulp.task('default', ['orchestrate', 'start']);
```

For this task, *Dockerode*, a Node.js package that implements *Docker Remote API* is used by a component that I implemented for this service.

```
var Orchestration = module.exports = function (dockerConfig) {
    // Requires
```

```
    this.Promise = require('bluebird');
    this.fs = require('fs');
    this.request = require('request-promise');
    this.Docker = require('dockerode');

    // Default values
    this.dockerConfig = dockerConfig || {
            "host": "192.168.99.100",
            "port": 2376,
            "protocol": "https",
            "ca": this.fs.readFileSync('/ca.pem'),
            "cert": this.fs.readFileSync('/cert.pem'),
            "key": this.fs.readFileSync('/key.pem')
        };
    this.dockerInstance = this.Promise.promisifyAll(new
this.Docker(this.dockerConfig));

    // return instance in case of function called
    if (!(this instanceof Orchestration)) return new
Orchestration(dockerConfig);
};

Orchestration.prototype.getDockerInstance = function getDockerInstance() {
    return this.dockerInstance;
};

Orchestration.prototype.up = function runScraper(service, name) {
    var _up = this._up;

    return _up({
        "Image": "redis:latest",
        "name": name + '_' + service + "_redis"
    }).then(function (data) {
        return _up({
            "Image": "ind3x/" + service,
            "name": name + "_" + service + "_" + service,
            "Env": [
                "REDIS_PORT_6379_TCP_ADDR=" + data.NetworkSettings.IPAddress,
                "REDIS_PORT_6379_TCP_PORT=" +
Object.keys(data.NetworkSettings.Ports).shift().split('/')[0]
            ],
            "ExposedPorts": {
                "3000/tcp": {}
            },
```

```
            "HostConfig": {
                "Links": [name + '_' + service + "_redis:db"],
                "PortBindings": {
                    "3000/tcp": [{"HostPort": ""}]
                }
            }
        })
    }).catch(function (e) {
        console.error(e.message);
    });
};

Orchestration.prototype._up = function _run(params) {
    var docker = this.dockerInstance;

    return new this.Promise(function (resolve, reject) {
        docker.infoAsync().then(function (info) {
            if (info) {
                console.info('[' + new Date().toISOString() + '] Pulling ' +
params.Image + '...');
                return docker.pullAsync(params.Image);
            }
        }).then(function (stream) {
            stream.on('error', function (e) {
                reject(e);
            });

            stream.on('data', function (data) {
            });

            stream.on('end', function () {
                console.info('[' + new Date().toISOString() + '] Creating
container from image ' + params.Image + '...');

                docker.createContainerAsync(params).then(function (container)
{
                    var container = Promise.promisifyAll(container);

                    container.startAsync().then(function () {
                        return container.inspectAsync();
                    }).then(function (data) {
                        resolve(data);
                    }).catch(function (e) {
                        reject(e);
```

```
                    });
                }).catch(function (e) {
                    // delete container
                    console.error('[' + new Date().toISOString() + '] Cannot
create container from ' + params.Image + ', trying to delete if one
exist...');

                    docker.getContainer(params.name).removeAsync({
                        "force": true
                    }).then(function (data) {
                        console.info('[' + new Date().toISOString() + ']
Removed container with name ' + params.name + '...');
                    });

                    reject(e);
                });
            });
        }).catch(function (e) {
            reject(e);
        });
    });
};
```

Finally, this service is dockerized with the correspondent *Dockerfile*.

```
FROM node:latest

RUN mkdir /usr/src/app/ && npm install gulp -g

WORKDIR /usr/src/app
COPY app/package.json /usr/src/app/package.json

RUN npm install

COPY app/ /usr/src/app

EXPOSE 3004

CMD [ "npm", "start" ]
```

3.2.6. Consul

Otherwise, the system must need other services for a proper functioning. The main reason for implementing other components as services, is that the system is

focused on a distributed environment where services are deployed on different nodes inside a infrastructure like a cluster, needing to know in which node is a particular service.

Consul is the solution. Consul is defined by its owners *HashiCorp*, as a distributed, highly available and datacentre-aware tool for discovering and configuring services in an infrastructure.

That is, a service discovery backend like *Consul* is ideal to store information about where each service is and lets each service know where is another one with a request to a *RESTful API* provided by *Consul*.

Also, I complemented Consul service with *Registrator*. *Registrator* automatically registers and deregisters services for any Docker container by inspecting containers as they come online.

3.3. Deployment of the distributed data scraper system

Once the development of the brief use case is done, it is time for the deployment of the system in a production environment. The environment I have chosen is the *1&1 Cloud Panel,* which provides all necessary tools to deploy every virtual machine that I want in different datacentres with a very user-friendly panel or interface. Also they provide a *Docker Machine driver* too.

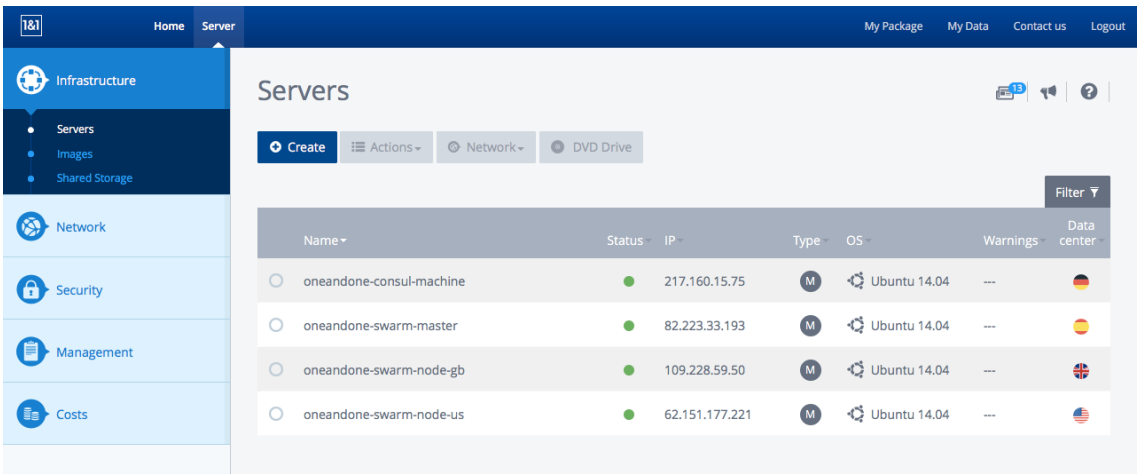

Figure 19. 1&1 Cloud Panel

Also I have decided to use *Ubuntu* as Linux distribution for the reason that it is one of the recommended by *Docker* and also the most used.

3.3.1. Provisioning Docker Swarm cluster with Consul

Moreover, for the system deployment, I have deployed four virtual machines, one on each datacentre offered by 1&1 and for this labour I have written a bash script to provision easily and configure the *Docker Swarm* cluster through *Docker Machine.*

```
#!/bin/bash

echo "Creating 1&1 Docker Swarm cluster"
echo "Creating oneandone-consul-machine..."
  docker-machine create -d oneandone \
  --oneandone-api-key              a85313038fabaff02c086dff8e9e885e \
  --oneandone-datacenter           DE \
  --oneandone-size                 M \
  --oneandone-os                   ubuntu1404-64std\
  oneandone-consul-machine
wait

echo "Running Consul Docker image in oneandone-consul-machine..."
{
  eval $(docker-machine env oneandone-consul-machine)
  wait
  docker run -d -p 8500:8500 progrium/consul -server -bootstrap
} &> /dev/null
wait

echo "Creating oneandone-swarm-node-us..."
  docker-machine create -d oneandone \
  --oneandone-api-key              a85313038fabaff02c086dff8e9e885e \
  --oneandone-datacenter           US \
  --oneandone-size                 M \
  --oneandone-os                   ubuntu1404-64std \
  --swarm \
  --swarm-discovery="consul://$(docker-machine ip oneandone-consul-
machine):8500" \
  --engine-opt="cluster-store=consul://$(docker-machine ip oneandone-consul-
machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
  oneandone-swarm-node-us
wait

echo "Creating oneandone-swarm-node-gb..."
  docker-machine create -d oneandone \
  --oneandone-api-key              a85313038fabaff02c086dff8e9e885e \
```

```
  --oneandone-datacenter           GB \
  --oneandone-size                 M \
  --oneandone-os                   ubuntu1404-64std \
  --swarm \
  --swarm-discovery="consul://$(docker-machine ip oneandone-consul-
machine):8500" \
  --engine-opt="cluster-store=consul://$(docker-machine ip oneandone-consul-
machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
  oneandone-swarm-node-gb
wait

echo "Creating oneandone-swarm-master..."
  docker-machine create -d oneandone \
  --oneandone-api-key              a85313038fabaff02c086dff8e9e885e \
  --oneandone-datacenter           ES \
  --oneandone-size                 M \
  --oneandone-os                   ubuntu1404-64std \
  --swarm \
  --swarm-master \
  --swarm-discovery="consul://$(docker-machine ip oneandone-consul-
machine):8500" \
  --engine-opt="cluster-store=consul://$(docker-machine ip oneandone-consul-
machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
  oneandone-swarm-master
wait

echo "Running Registrator Docker image in oneandone-swarm-master..."
{
  eval $(docker-machine env --swarm oneandone-swarm-master)
  wait
  docker -H $(docker-machine ip oneandone-swarm-master):2376 run -d \
  --name=registrator \
  --net=swarm-network \
  --volume=/var/run/docker.sock:/tmp/docker.sock \
  gliderlabs/registrator:latest -ip $(docker-machine ip oneandone-swarm-
master) \
  consul://$(docker-machine ip oneandone-consul-machine):8500
} &> /dev/null
wait

echo "Running Registrator Docker image in oneandone-swarm-node-us..."
{
```

```
  eval $(docker-machine env --swarm oneandone-swarm-master)
  wait
  docker -H $(docker-machine ip oneandone-swarm-node-us):2376 run -d \
  --name=registrator-us \
  --net=swarm-network \
  --volume=/var/run/docker.sock:/tmp/docker.sock \
  gliderlabs/registrator:latest -ip $(docker-machine ip oneandone-swarm-node-
us) \
  consul://$(docker-machine ip oneandone-consul-machine):8500
} &> /dev/null
wait

echo "Running Registrator Docker image in oneandone-swarm-node-gb..."
{
  eval $(docker-machine env --swarm oneandone-swarm-master)
  wait
  docker -H $(docker-machine ip oneandone-swarm-node-gb):2376 run -d \
  --name=registrator-gb \
  --net=swarm-network \
  --volume=/var/run/docker.sock:/tmp/docker.sock \
  gliderlabs/registrator:latest -ip $(docker-machine ip oneandone-swarm-node-
gb) \
  consul://$(docker-machine ip oneandone-consul-machine):8500
} &> /dev/null
wait

echo "Running SwarmUI image in oneandone-swarm-master..."
{
docker -H $(docker-machine ip oneandone-swarm-master):2376 run -d \
  --name=swarmui \
  --volume=/etc/docker:/etc/docker \
  -p 9000:9000 ptimagos/swarmui \
  -consul http://$(docker-machine ip oneandone-consul-machine):8500 \
  -tls -CA /etc/docker/ca.pem -cert /etc/docker/server.pem -key
/etc/docker/server-key.pem
} &> /dev/null

echo "1&1 Docker Swarm cluster successfully created!"
```

As it can be see on the code above, the initial step is to create the virtual machine inside the datacentre located in Germany. This virtual machine is called *oneandone-consul-machine* and after creation it runs and creates a *Docker*

container from *Consul* image on it. Its objective is to centralize all service registration outside *Docker Swarm* cluster.

The next step is the creation and configuration of Docker Swarm. For this purpose, it creates three virtual machines, each of them in different datacentres located in Spain, United Kingdoms and United States of America. One of them is called *oneandone-swarm-master* and acts as manager of the entire cluster being the responsible for scheduling container creation across the cluster. The rest are nodes joined to the cluster.

When the creation of virtual machines is finished, it is time for running and creating new Docker containers from *Registrator* image in each node. This is going to register new Docker containers in Consul when they are created. As a condition, it is mandatory to force the use of public IP address for registering services because if not, it will be impossible to discover the public IP address for using other *RESTful APIs* from services.

The last step, and just for a better viewing of the cluster, it creates a container from *SwarmUI* image. *SwarmUI* as the owners say, is a web interface for the *Docker* and *Swarm Remote API* being the goal to provide a pure client side implementation so it is effortless to connect and manage *Docker* [55]. It is necessary to take into account that *Docker Machine* provision virtual machines configured with *TLS* certificates and for this reason, *SwarmUI* must be running and configured in the same node where Swarm manager is configured, in this case, in *oneandone-swarm-master node*.

As a result, the infrastructure is now successfully prepared and ready for the system deployment. This is possible to check by connecting with the *Swarm cluster*.

```
eval $(docker-machine env --swarm oneandone-swarm-master)
docker info
```

This shows and outputs with the information about the cluster.

```
Containers: 25
 Running: 21
 Paused: 0
 Stopped: 4
Images: 35
Server Version: swarm/1.2.3
Role: primary
Strategy: spread
Filters: health, port, containerslots, dependency, affinity, constraint
Nodes: 3
 oneandone-swarm-master: 82.223.33.193:2376
  └ ID: RKB4:NNPX:EAA4:QORF:LXV6:3MCG:4QSI:BE2J:44PL:PEE4:GEWG:7O5H
  └ Status: Healthy
  └ Containers: 9
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.012 GiB
  └ Labels: executiondriver=, kernelversion=3.13.0-86-generic,
operatingsystem=Ubuntu 14.04.4 LTS, provider=oneandone, storagedriver=aufs
  └ UpdatedAt: 2016-06-10T16:45:35Z
  └ ServerVersion: 1.11.1
 oneandone-swarm-node-gb: 109.228.59.50:2376
  └ ID: N7YB:7IJU:TOFG:YADK:NTHJ:6TQU:YSYV:FSWJ:FBDB:NE7K:PSCL:54JF
  └ Status: Healthy
  └ Containers: 7
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.012 GiB
  └ Labels: executiondriver=, kernelversion=3.13.0-86-generic,
operatingsystem=Ubuntu 14.04.4 LTS, provider=oneandone, storagedriver=aufs
  └ UpdatedAt: 2016-06-10T16:45:35Z
  └ ServerVersion: 1.11.1
 oneandone-swarm-node-us: 62.151.177.221:2376
  └ ID: NBET:WM4S:KYKX:Y7AD:LLWZ:4GC4:3FOE:6NSH:IFIO:OWE6:OSNF:47RD
  └ Status: Healthy
  └ Containers: 9
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.012 GiB
  └ Labels: executiondriver=, kernelversion=3.13.0-86-generic,
operatingsystem=Ubuntu 14.04.4 LTS, provider=oneandone, storagedriver=aufs
  └ UpdatedAt: 2016-06-10T16:45:04Z
  └ ServerVersion: 1.11.1
Plugins:
 Volume:
 Network:
Kernel Version: 3.13.0-86-generic
```

```
Operating System: linux
Architecture: amd64
CPUs: 3
Total Memory: 3.037 GiB
Name: f1b8081b7f55
```

3.3.2. Deployment of the distributed data scraper system in the Swarm cluster

The following step after the Swarm cluster creation, it is to deploy the distributed data scraper. For this task, I have used *Docker Compose* and as explained in the chapter 2.2.2, it is a tool for creating and managing multi-container applications where containers are all defined and linked in a single *YAML* file spinning up all containers in a single command.

Also, I have taken some considerations. *Scraper service* and *Endpoint service* are not necessary to deploy from respective images because the *Orchestration-engine service* is the responsible for this task. *Orchestration-engine service* needs to be running on swarm manager node for the reason that needs *oneandone-swarm-master TLS certificates* for running new *Scraper service* and *Endpoint service containers* when a new configuration is detected.

Moreover, I have considered letting *Docker* automatically define containers port mapping and containers are going to be processed from *Docker images* that are located in my *Docker Hub* account.

At last, I have added some interfaces in each service to be deployed for helping with the management of the system.

Datasource service deployment

Going through deployment of the *Datasource service*, a *Docker Compose YAML* file has been defined linking the new *Datasource service* container with a *MongoDB* container instance. Both are complemented with a *mongo-express* container, which implements a web interface form *MongoDB* database.

```
datasource:
  image: ind3x/datasource
  ports:
```

```
    - "3001"
  links:
    - "db:mongo"

mongo-express:
  image: mongo-express
  ports:
    - "8081"
  links:
    - "db:mongo"

db:
  image: mongo
  ports:
    - "27017"
```

As a result, it is possible to list created *Datasource service* that is running in *Swarm* cluster node located in United Kingdom.

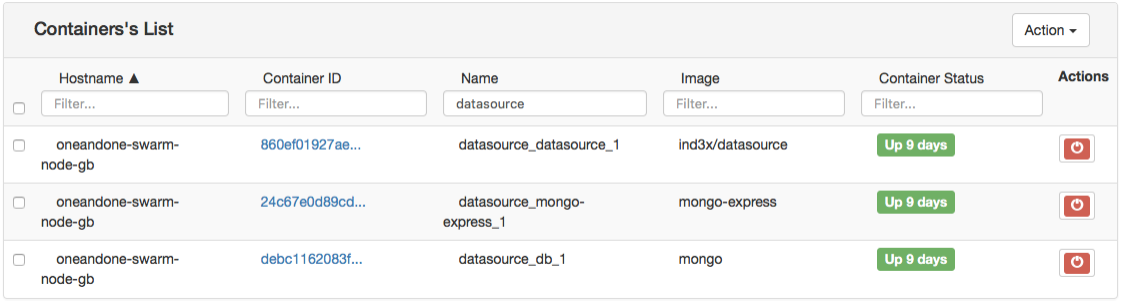

Figure 20. Running containers for Datasource service

Configuration-discovery service deployment

Like the deployment of the *Datasource service*, *Configuration-discovery* service has a similar *Docker Compose YAML file*.

```
configuration-discovery:
  image: ind3x/configuration-discovery
  ports:
    - "3003"
  links:
    - "db:mongo"

mongo-express:
  image: mongo-express
  ports:
    - "8081"
  links:
```

```
    - "db:mongo"

db:
  image: mongo
  ports:
    - "27017"
```

In this case, the result is similar and the service is running in a cluster node located in United States of America.

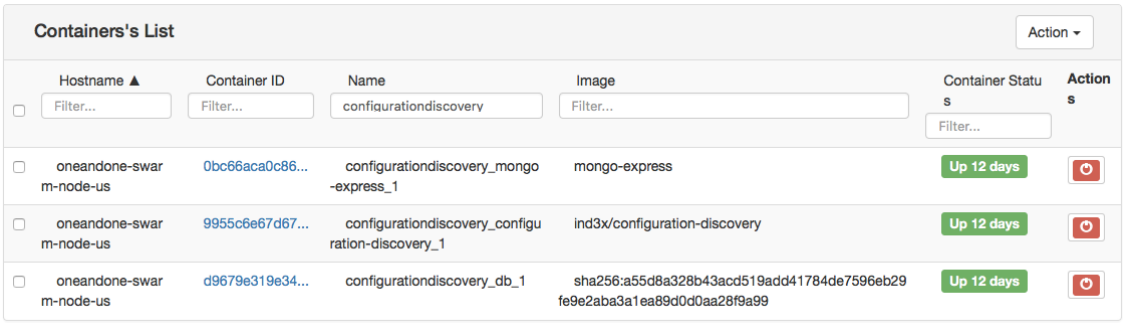

Figure 21. Running containers for Configuration-discovery service

Orchestration-engine service deployment

Finally, *Orchestration-engine service* has to be deployed. In this case, the *Docker Compose YAML file* is somewhat different and as commented in the introduction of the chapter, *the Orchestration-engine service* must be deployed in the *Swarm* node where the *Swarm* manager is specifying the host to the *Docker Compose CLI*.

The *Docker Compose YAML file*.

```
orchestration-engine:
  image: ind3x/orchestration-engine
  ports:
    - "3004"
  volumes:
    - /etc/docker:/etc/docker
```

And the command for running it in the *oneandone-swarm-manager node*.

```
docker-compose -H $(docker-machine ip oneandone-swarm-master):2376 up
```

Thus, the result after deploying is that the service is running in *oneandone-swarm-master* node located in Spain.

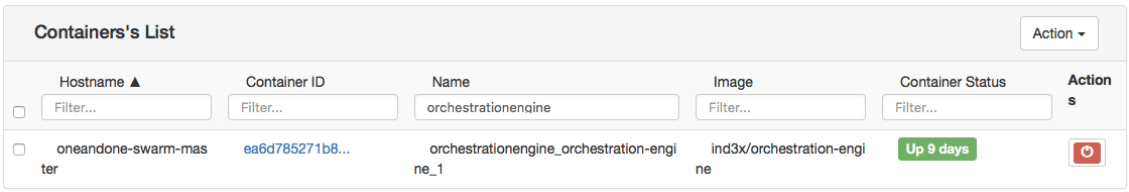

Figure 22. Running container for Orchestration-engine service

3.3.3. Evaluation

Once all the system is deployed, I have started with the evaluation of the distributed data scraper system. The idea is to evaluate data and system flow from correspondent *Scraper service* to *Endpoint service* by entering a new configuration.

```
{
    "type": "scraper",
    "collection": "restaurants",
    "scraper": {
    "source": "http://www.eltenedor.es/busqueda/logrono/316013?cc=16771-
61c&gclid=Cj0KEQjwj7q6BRDcxfG4pNTQ2NoBEiQAzUpuW19bZTzwQHswm-
uzigz80U0NsnT22bnu0p0UNNy0s9YaAi5H8P8HAQ",
        "scope": "li.resultItem",
        "selector": [
        {
            "title": "h3.resultItem-name a",
            "item": "div.resultItem-address"
        }
    ],
        "paginate": "div.pagination li.next a@href",
        "limit": 3
},
    "ttl": 300,
    "container": null
}
```

New configuration like this one is going to scrap information from a list of data from a specific web page, generating a system flow like in *Figure 23*.

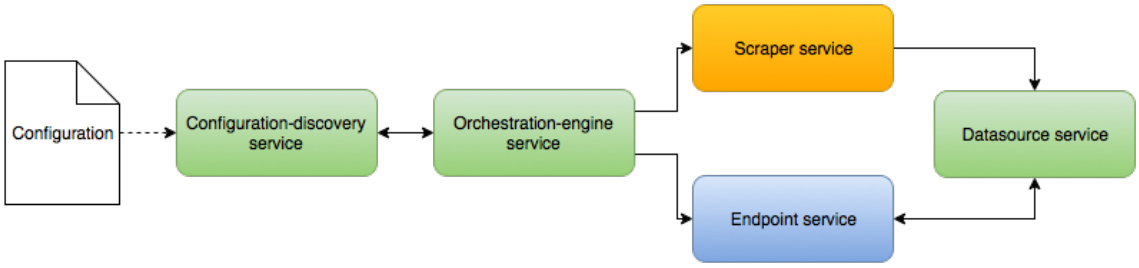

Figure 23. Decentralized data scraper system flow

That is, when a new configuration is added to *Configuration-discovery service*, *Orchestration-engine service* assigns and runs a *Scraper service* and *Endpoint service* from correspondent configuration. When both start, *Scraper service* gets data from the given configuration and sends it to *Datasource service*, processing the data and leaving it to *Endpoint service*, which gets and exposes this new data to the public.

Going in depth, *Configuration-discovery* generates two new configurations, one for the new *Scraper service*, and other for *Endpoint service*.

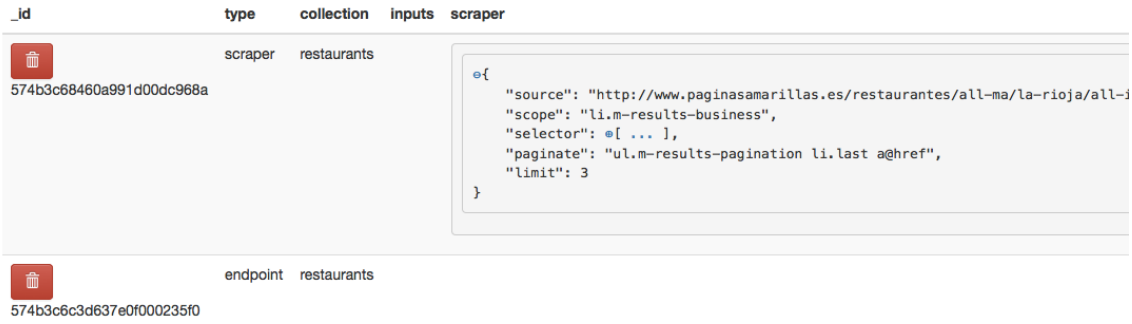

Figure 24. New configurations example

Moreover, *Orchestration-engine service* uses and assigns these configurations to run respective new services containers in the most appropriate node from the *Swarm cluster* by using *Swarm Remote API*, that communicates to *Swarm manager* to run these containers in the most appropriate node. *Swarm manager* selects a node using spread strategy that computes rank according to a node's available CPU, its RAM, and the number of containers it has.

Figure 25. Running services for new configuration

In addition, it is possible to check that new *Scraper service* container has collected data and sent it to *Datasource service* looking *MongoDB* interface deployed with it.

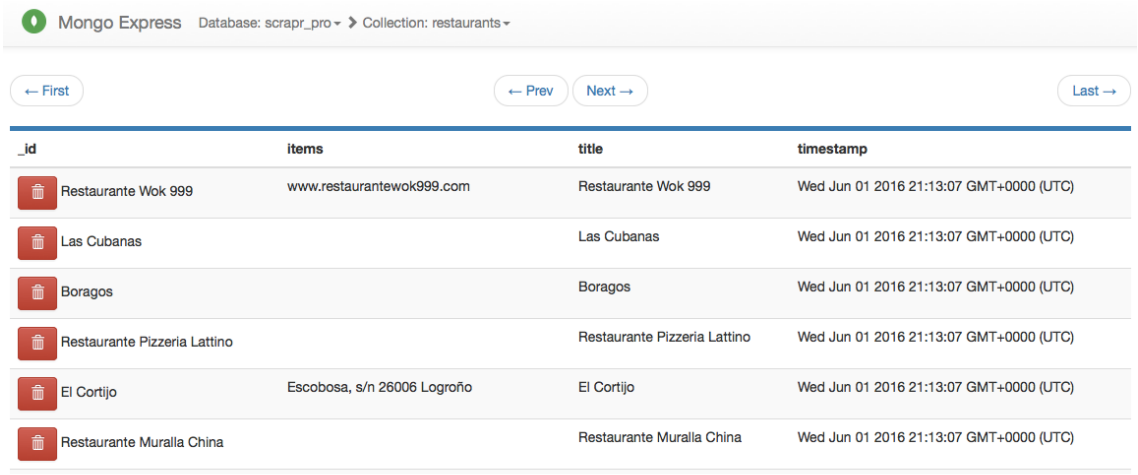

Figure 26. Scraped data in Datasource service database.

Finally, new *Endpoint service* must be able to get new processed data from *Datasource service* and expose it to the public using implemented *RESTful API*.

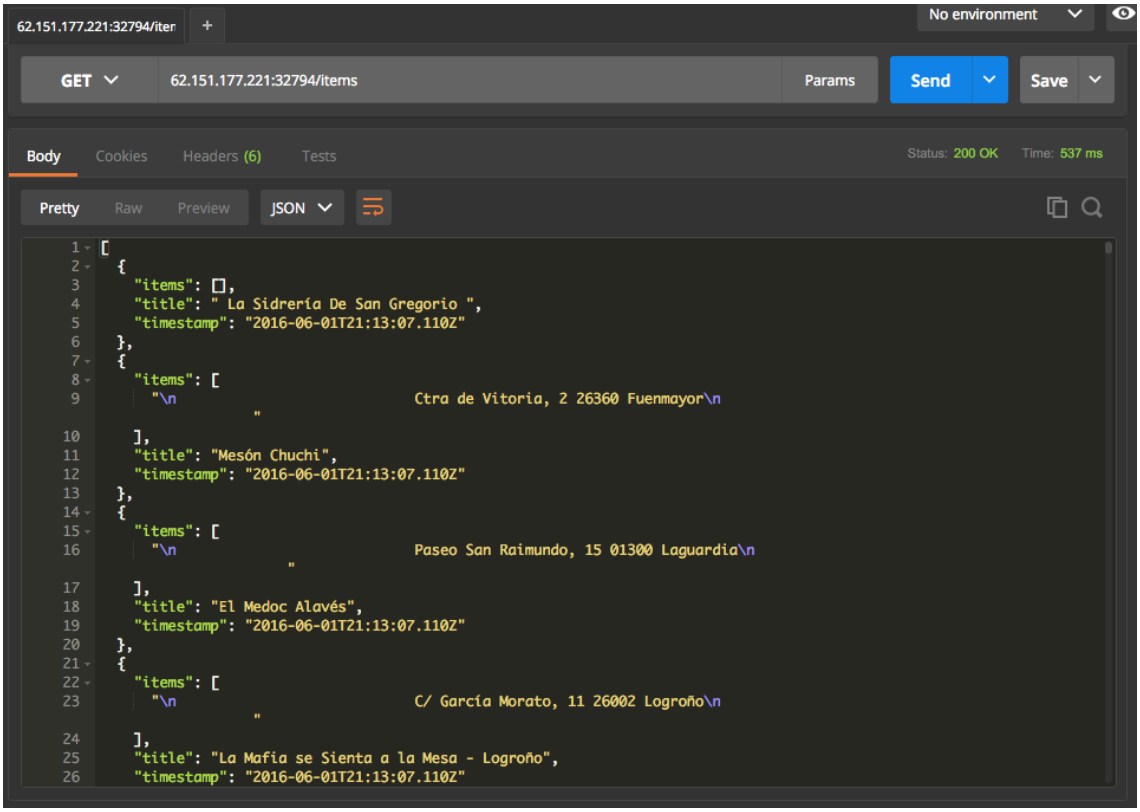

Figure 27. Response from Endpoint service

To summarise, deployment of the *Distributed data scraper system* has been satisfactory and fully demonstrated.

4. Conclusions

In conclusion, as a global overview of the *Final Master's degree Project*, I have to say that the result of the study, development, implementation and deployment of a distributed system with *microservice architecture* using *Docker* has been a great challenge. Not only for the potential that the system can offer itself and to the advantages that *microservice architecture* and *Docker* offer explained in both case studies in chapter 2. Furthermore, for the reason that microservice architecture and Docker go hand in hand and both are the new Legos for cloud computing and for building highly available distributed applications.

In general terms, I can mention some benefits that show application or system development by implementing *microservice architecture* and *Docker* like separation of concerns, encapsulation of services, used technologies, fast system deployment and decrease of deployment cost.

Separation of concerns means that you can distribute different responsibilities of the system into different smaller pieces or services that enhance the cohesion and decrease the coupling, making it easier to change, add functions and qualities to the system. The result of applying this design principle with *microservice architecture* and *Docker* is that the use case presented on this project is divided in five independent services, where each is focused to provide a particular functionality around the responsibility it has. Furthermore, it allows a better continuous refactoring, reducing the need for a big up designs and allows for releasing the software early and continuously.

Moreover, I have realised that *microservices* and *Docker* increased the benefits of separation of concerns by giving all the boundaries when choosing the best technology stack for each service. For example, *Scraper service* is developed with *Node.js*, a highly reusable server-side JavaScript, which enable the best performance when collecting data. The *Datasource service* instead, implements *MongoDB*, a *NoSQL* database which provides high availability plus native frameworks and functions like *Aggregation Framework* and *Map-Reduce* for processing a high volume of data.

Also, I have found that the implementation of *Docker* agile system development and deployment. *Docker* lets the freedom to define an environment on each service just by reusing and linking together available third-party *Docker images*. Furthermore, *Docker* has given me the choice of deploying the system without the complexity of configuring the infrastructure for each service as a consequence of the portability it offers across machines. A *Docker container* gives all needed components and other tools like *Docker Machine* and *Docker Swarm*, give all necessary to provision a *Docker Swarm* cluster of multiple *Docker hosts* as nodes.

At last, I have chosen architecture and platform, which provide a decrease of deployment costs due to a better exploitation of resources. *Docker* operates at process level, isolating every running container. This is a big advantage when scaling a system. *Microservice architecture* and *Docker* provide an optimal solution to scale a service from the system on demand and in the same machine, taking advantages of all resources from a machine instead of having multiple replicas of the entire system on different machines.

In spite of all advantages, I also have gone through with different disadvantages during the development and implementation of the *Final Master's degree Project*, probably all of then due to my lack of experience with terms and technologies used in the project. Nevertheless, all of them easy to workaround and solve.

Microservice architecture and *Docker* have demonstrated that there is an addition of complexity caused by a building block nature of both approaches, causing the needs of having everything properly controlled and monitored for troubleshooting any problem that can occur when working with containers. Hence, one consequence is the possibility of falling into *nanoservices anti-pattern*, a tend to split a piece of an application to smaller and smaller chunks when designing it, resulting in too fine grained services that overheads communications between services and complicates system maintenance.

Finally, I am in obliged to finish this conclusion with a little reflexion. Have I learnt everything I expected to know? The answer is yes. I have learnt all things related with the objectives I proposed on this *Final Master's degree Project*. From the infrastructure configuration that implies *Docker Swarm* and distributed systems to the most popular development technologies like *Docker*, *Node.js* and *MongoDB*, up to architectural style like *microservices*.

Besides, I have dealt with different kind of issues like configuring the *Docker Swarm* cluster infrastructure, integrating *Consul* or *Swarm Remote API* and struggling with asynchronous nature of *Node.js*, which can be solved by using JavaScript Promises.

Apart from those issues, I have also encountered deviations in the work plan due to bad estimations and, for this reason, I have had to take the decision to discard a study case about data scraping and decrease the dedicated time to develop and implement the brief use case to guarantee the success of the project. However, suggested approaches and planning have been followed.

4.1. Next steps

To conclude and as mentioned before, the system is a tiny system that has had as an objective to put into practise and experiment with lessons learned about the case studies expose in this work. Then, I have seen that there are numerous steps that must be done to improve the system itself if there is a possibility to get advantage of this *Distributed data scraper* in a production environment:

- Implement a parameters validator in the *RESTful API*s of each service for performing security around data that incomes in the system.
- *RESTful API* versioning must be implemented on each service to avoid issues with applications that uses it.
- It is recommended refactoring the code as defined by standards for a proper development of a new product from this system.
- Integrate *Consul DNS Interface* in the system for improving service discovery.

- Set a better *Docker Swarm cluster* configuration like replication to make the cluster failure tolerant and increase high availability and security.
- Set up needed configuration for security improvements on each component involved in the system.
- Implement a system that scales the system automatically by provisioning new machines and joining them as a node to the cluster.
- Implement a solution like a *remote observer patter* approach, *event-driven architecture* or even thou a crontab on each service under *microservice architecture* to improve the system performance and an efficient solution for detecting new configuration and data to process instead of using *JavaScript* native timeout and intervals.
- Implement the possibility of setting data inputs in *Scraper service*, offering the possibility to scrap data with an input given by *Endpoint Service*.
- Implement *socket.io*, from communication between services to external communications, which increase data availability by integrating real-time capabilities.
- Integrate an authentication service like *OAuth* to ensure a proper utilization of information exposed by *Endpoint service.*
- Integrate *NGINX* for load balancing access to each service.

5. Glossary

Architectural style: defines a family of systems in terms of a pattern of structural organization; a vocabulary of components and connectors, with constraints on how they can be combined [37].

Bulkheads: is a mechanism that imposes a damage contention principle where an application can isolate a fail for protecting the entire system from total disruption. The most common technique is to replicate the system among a number of servers [2].

Circuit Breakers: is a mechanism that wraps with a component all dangerous operations for preventing it from new requests when the system is not available [2].

Cluster: set of connected computers that work together so that, in many respects, they can be viewed as a single system.

Cohesion: is defined in computer programming when a class is responsible of a well-defined task.

Continuous Delivery: is the ability to get changes of all types, including new features, configuration changes, bug fixes and experiments, into production, or into the hands of users, safely and quickly in a sustainable way [30].

Coupling: is defined in computer programing as the level of dependency of a class with other.

Data scraper: is a technique in which a computer program extracts data from human-readable output coming from another program [31].

Datasource: is described as source where a program relies on to get data.

Docker Compose: a tool for creating and managing multi-container applications where containers are all defined and linked in a single file spinning up all containers in a single command.

Docker container: similar to a directory, which holds everything needed for an application to run. A Docker container is created from a Docker image and is an isolated and secure application platform. Docker containers are the run component of Docker.

Docker Engine: a lightweight runtime and containerization platform that uses Linux kernel namespaces, which give the isolated workspace and control group to a *Docker host*, enabling containers to be built, shipped and run.

Docker host: machine or host where *Docker Engine* is installed.

Docker image: a read-only template used to create *Docker containers*. *Docker images* are the build component of *Docker*.

Docker Machine: a tool that automatically provisioning *Docker hosts* and install *Docker Engine* on them, providing more efficient processes.

Docker Registry and Docker Hub: defined as stateless, highly scalable server side applications that stores and lets distribute *Docker images*.

Docker Swarm: a native clustering tool that clusters *Docker hosts* and schedules containers using scheduling strategies, also turns a pool of host machines into a single virtual host.

Docker swarm cluster: a cluster of *Docker hosts* in which is *Docker Swarm* configured.

Enterprise Service Bus or ESB: a software architecture model used for designing and implementing communication between mutually interacting software applications in a *service-oriented architecture* or *SOA* [27].

Endpoint: is an entry point to a service in a *service-oriented architecture*, *microservice architecture*.

Eventual consistency: is defined as the point of a future when data from a distributed database is consistent [28].

Express: a *Node.js* minimal and flexible web application framework ideal for creating a robust *RESTful API*.

Git: distributed version control system designed to handle everything from small to very large projects with speed and efficiency [43].

Gulp: is a *Node.js*-based task runner.

Gulpfile: a *JavaScript* file containing defined Gulp tasks.

Microservice architecture: is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms like HTTP resource API.

MongoDB: is an open-source *NoSQL* document database that provides high performance, high availability, and automatic scaling [41].

Monolithic architecture: is an application built as a single unit or single logical executable that normally handle HTTP requests, execute domain logic retrieving or updating data from database which populate a view for sending to the browser.

Nanoservices anti-pattern: a tend to split a piece of application to smaller and smaller chunks when designing it, resulting in a too fine grained services that overheads communications between services and difficult system maintenance.

NoSQL database: refers to a group of non-relational data management systems; where databases are not built primarily on tables, and generally does not use SQL for data manipulation. *NoSQL* database management systems are useful when working with a huge quantity of data when the data's nature does not require a relational model [35].

Node.js: is a JavaScript runtime built on Chrome's V8 JavaScript engine. *Node.js* uses an event-driven, non-blocking I/O model that makes it lightweight and efficient [40].

REST or RESTful: is web standards based architectural style that uses HTTP Protocol for data communication. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods [36].

Redis: is an open source in-memory data structure store, used as database, cache and message broker [39].

Service-oriented architecture or SOA: is an architectural style that support service-orientation, being a way of thinking in terms of service and service-based development and the outcomes of services.

Timeouts: is a simple mechanism that stops waiting for a response that can consider as a loosed response [2].

X-Ray: a Node.js package that make web scraping a really simple affair.

YAML: is a human friendly data serialization standard for all programming languages [38].

6. References

1. **Martin Fowler and James Lewis. Microservices.** *http://martinfowler.com/articles/microservices.html*
2. **Sergio Maurenzu. Microservicios.** *http://sergiomaurenzi.blogspot.com.es/2015/04/microservicios-parte-i.html*
3. **Chris Richardson. Introduction to Microservices.** *https://www.nginx.com/blog/introduction-to-microservices*
4. **How to cook Microservices. Architecture Pattern.** *http://howtocookmicroservices.com/architecture*
5. **Wikipedia. Microservices.** *https://en.wikipedia.org/wiki/Microservices*
6. **Wikipedia. Component-based software engineering.** *https://en.wikipedia.org/wiki/Component-based_software_engineering*
7. **Wikipedia. Service-oriented architecture.** *https://en.wikipedia.org/wiki/Service-oriented_architecture*
8. **SOA Manifesto.** *http://www.soa-manifesto.org*
9. **Andrés Hevia. Microservicios y SOA.** *https://pensandoensoa.com/2014/10/27/microservicios-y-soa*
10. **Txema Rodríguez. Trabajar con microservicios.** *http://www.genbetadev.com/paradigmas-de-programacion/trabajar-con-microservicios-evitando-la-frustracion-de-las-enormes-aplicaciones-monoliticas*
11. **C. Matthew MacKenzie,Ken Laskey,Francis McCabe,Peter F Brown, Rebekah Metz, Booz Allen Hamilton.** *"Reference Model for Service Oriented Architecture 1.0".* 12 October 2006, OASIS Standard.
12. **The Open Group. Service Oriented Architecture: What Is SOA?** *https://www.opengroup.org/soa/source-book/soa/soa.htm*
13. **Docker documentation.** *https://docs.docker.com*
14. **Jhonny Tu. Self-Paced Training from Docker Training.** *https://training.docker.com/self-paced-training*
15. **Deni Bertovic. Supercharge your development environment (using Docker).** *https://denibertovic.com/talks/supercharge-development-env-using-docker*

16. **Thomas Peham. Containerize your web development: How Docker is solving real world problems for web developers.** *http://usersnap.com/blog/docker-for-web-developers*
17. **Martin Rusev. Web Developer's Guide to Docker.** *https://www.amon.cx/blog/web-developer-guide-docker*
18. **Howtocookmicroservices. Development with Docker Compose.** *http://howtocookmicroservices.com/docker-compose*
19. **Chris Richardson. Service instance per container.** *http://microservices.io/patterns/deployment/service-per-container.html*
20. **Piotr Ciemielewski. Microservices and macro mistakes.** *http://allegro.tech/2016/01/microservices-and-macro-mistakes.html*
21. **Lucas Carlos. 4 ways Docker fundamentally changes application development.** *http://www.infoworld.com/article/2607128/application-development/4-ways-docker-fundamentally-changes-application-development.html*
22. **Vinay. Docker: Lightweight Containers vs. Traditional Virtual Machine.** *http://blogs.site24x7.com/2015/06/05/docker-lightweight-containers-vs-traditional-virtual-machines*
23. **Rory Hunter. Docker And The Unix Philosophy.** *http://www.blackpepper.co.uk/docker-unix-philosophy*
24. **Jeff Lindsay. Understanding Modern Service Discovery with Docker.** *http://progrium.com/blog/2014/07/29/understanding-modern-service-discovery-with-docker*
25. **Justing Ellingwood. The Docker Ecosystem: Service Discovery and Distributed Configuration Stores.** *https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-service-discovery-and-distributed-configuration-stores*
26. **Wikipedia. Computer cluster.** *https://en.wikipedia.org/wiki/Computer_cluster*
27. **Wikipedia. Enterprise service bus.** *https://en.wikipedia.org/wiki/Enterprise_service_bus*

28. **Peter Bailis and Ali Ghodsi.** *"Eventual Consistency today: Limitations, extensions, and Beyond".* May 2013, Vol. 56, no. 5, Communications of the ACM.
29. **Mis apuntes de programación.** *https://misapuntesdeprogramacion.wordpress.com/2013/01/16/acoplamiento-cohesion-y-encapsulacion*
30. **Continuousdelivery.com.** *https://continuousdelivery.com*
31. **Wikipedia. Data scraping.** *https://en.wikipedia.org/wiki/Data_scraping*
32. **Stackoverflow. Crawler vs. Scraper.** *http://stackoverflow.com/questions/3207418/crawler-vs-scraper*
33. **Neal. No API? No problem! Fake it with browser automation and web scraping.** *http://blog.devpost.com/post/124154812036/no-api-no-problem-fake-it-with-browser*
34. **Stackoverflow. Difference between Database and Data Source.** *http://stackoverflow.com/questions/3698044/difference-between-database-and-data-source*
35. **A B M Moniruzzaman, Syed Akhter Hossain.** *"NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison"*. International Journal of Database Theory and Application, Manuscript ID.
36. **Tutorialspoint. RESTful Web Services – Introduction.** *http://www.tutorialspoint.com/restful/restful_introduction.htm*
37. **Wikipedia. Architectural pattern.** *https://en.wikipedia.org/wiki/Architectural_pattern*
38. **YAML official web page.** *http://yaml.org*
39. **Redis official web page.** *http://redis.io*
40. **Node.js official web page.** *https://nodejs.org*
41. **MongoDB documentation.** *https://docs.mongodb.com/manual/introduction*
42. **Express official web page.** *http://expressjs.com*
43. **Git official web page.** *https://git-scm.com*
44. **Gulp.** *http://gulpjs.com*
45. **X-Ray.** *https://github.com/lapwinglabs/x-ray*

46. **Arun Gupta. Docker Swarm Cluster using Consul.** *http://blog.arungupta.me/docker-swarm-cluster-using-consul*
47. **Pascal Cremer. Getting started with Docker Compose and Node.js.** *https://github.com/b00giZm/docker-compose-nodejs-examples*
48. **Scott's Weblog. Docker Swarm on AWS using Docker Machine.** http://blog.scottlowe.org/2016/03/25/docker-swarm-aws-docker-machine
49. **1&1. Cloud Driver for Docker Machine.** *https://github.com/1and1/docker-machine-driver-oneandone*
50. **Iksose. Creating a REST API using Node.js, Express, and MongoDB.** *https://gist.github.com/iksose/9401758*
51. **Scotch.io. Build a RESTful API Using Node and Express 4.** *https://scotch.io/tutorials/build-a-restful-api-using-node-and-express-4*
52. **Cho S. Kim. Understanding module.exports and exports in Node.js.** *https://www.sitepoint.com/understanding-module-exports-exports-node-js*
53. **Consul.** *https://www.consul.io*
54. **External Configuration Store Pattern.** *https://msdn.microsoft.com/en-us/library/dn589803.aspx*
55. **SwarmUI.** *https://github.com/Ptimagos/swarmui*

7. Appendices

Appendix A: Distributed data scraper documentation

This appendix has as a result to give more information about each service evolved in the system like addresses to administrator interfaces and *RESTful API* information.

Docker Hub

https://hub.docker.com/u/ind3x/

Swarm cluster

Swarm cluster interface

http://82.223.33.193:9000

List of Swarm cluster nodes

- *oneandone-swarm-master (Spain):* 82.223.33.193
- *oneandone-swarm-node-gb (UK):* 109.228.59.50
- *oneandone-swarm-node-us (USA):* 62.151.177.221

Services

Scraper service

RESTful API

URLs from *Scraper service RESTful API.*

- **GET** */*
  Redirect to **GET** */scraper/start*
- **GET** */ping*
  Use to test if service is running or not.
- **GET** */configurations*
  Get the service configuration.
- **PUT** */configurations*
  Update the service configuration.
- **GET** */scraper/start*
  Scrap data from the given configuration.

Datasource service

Database administrator interface

http://109.228.59.50:32830/db/scrapr_pro

RESTful API

URLs from *Datasource service RESTful API.*

Example: *109.228.59.50:32831/ping*

- **GET** */*

  Redirect to **GET** */collections*
- **GET** */ping*

  Use to test if service is running or not.
- **GET** */collections*

  Get a list of available collections.
- **GET** */collections/:collection*

  Get all items from a specific collection name.
- **POST** */items*

  Add a batch of scraped items to a capped collection for being processed.

Endpoint service

RESTful API

URLs from *Endpoint service RESTful API.*

- **GET** */*

  Redirect to **GET** */items*
- **GET** */ping*

  Use to test if service is running or not.
- **GET** */configurations*

  Get service configuration.
- **GET** */items*

  Get cached items retrieved from *Datasource service*.

Configuration-discovery service

Database administrator interface

http://62.151.177.221:32792/db/scrapr_pro

RESTful API

URLs from *Configuration-discovery service RESTful API.*

Example: *http://62.151.177.221:32791/ping*

- **GET** */*
  Redirect to **GET** */configurations*
- **GET** */ping*
  Use to test if service is running or not.
- **GET** */configurations*
  Get a list of available configurations.
- **POST** */configurations*
  Add a new configuration.
- **GET** */configurations/:id*
  Get a configuration with a specific id.
- **PUT** */configurations/:id*
  Update a configuration with a specific id.
- **DELETE** */configurations/:id*
  Delete a configuration with a specific id.

Orchestration-engine service

RESTful API

URLs from *Configuration-discovery service RESTful API.*

Example: *http://82.223.33.193:32836/ping*

- **GET** */ping*
  Use to test if service is running or not.

Consul service

This service uses Consul as service discovery backend and is running in a independent Docker host provisioned in Germany.

Consul interface address is *http://217.160.15.75:8500*