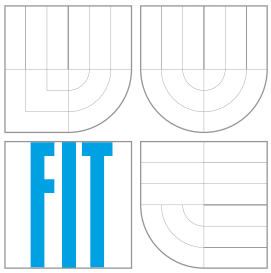
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

INTEGRATION OF THE IBM SOFTLAYER INTO THE MANAGEIQ FRAMEWORK

INTEGRACE IBM SOFTLAYER DO PROSTŘEDÍ MANAGEIQ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
TOMÁŠ COUFAL

SUPERVISOR
VEDOUCÍ PRÁCE
Mgr. ADAM ROGALEWICZ, Ph.D.

BRNO 2016

Zadání bakalářské práce/18948/2015/xcoufa09

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Coufal Tomáš**
Obor: Informační technologie
Téma: **Integrace IBM SoftLayer do prostředí ManageIQ**
Integration of the IBM SoftLayer into the ManageIQ Framework
Kategorie: Počítačové sítě

Pokyny:

1. Nastudujte problematiku Cloud Computing.
2. Nastudujte existující coudová řešení s důrazem na IBM SoftLayer.
3. Prostudujte prostředí ManageIQ určeného pro správu cloudu.
4. Navrhněte integraci IBM SoftLayer do prostředí ManageIQ.
5. Navrženou integraci implementujte a otestujte.
6. Připravte vaší implementaci pro nasazení do upstreemu projektu ManageIQ.

Literatura:

- http://manageiq.org/
- IBM SoftLayer http://www.softlayer.com/
- Fog knihovna pro SoftLayer: https://github.com/fog/fog-softlayer

Pro udělení zápočtu za první semestr je požadováno:

- Body 1-3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rogalewicz Adam, Mgr., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

As cloud computing gains its popularity the complexity of offered services is growing. There exist various different solutions and to manage them effectively is the task for cloud managers. One of them is a project named ManageIQ. This thesis aims to describe how to integrate a new provider into this software, what are the required bindings and how to work with the API of the provider. This thesis describes such a process on an IBM SoftLayer provider example. The achieved result enables the user to manage his SoftLayer devices via ManageIQ with ease and also provides him the functionality to provision new appliances.

Abstrakt

Cloudová řešení získávají na popularitě, spolu s tím však roste jejich složitost. Pro jejich efektivní správu a řízení existují různá řešení. Projekt ManageIQ je jedním z těchto nástrojů. Tato bakalářská práce se zabývá integrací nového poskytovatele cloudové infrastruktury, IBM SoftLayeru, do prostředí správce cloudových řešení ManageIQ. Na tomto příkladě je vysvětleno, jaká rozhraní je třeba použít pro správu poskytovatelů, jak pracovat s cloudovými API a jaké výzvy je třeba řešit při integraci nového poskytovatele. Výsledkem práce je funkční prostředí umožnující snadnou práci v IBM SoftLayeru skrze ManageIQ. To uživateli zprostředkovává sledování a úpravy dostupných zařízení či vytváření a spouštění nových součástí infrastruktury.

Keywords

cloud computing, ManageIQ, IBM SoftLayer, cloud providers, fog, Ruby

Klíčová slova

cloudové techologie, ManageIQ, IBM SoftLayer, poskytovatelé cloudu, fog, Ruby

Reference

COUFAL, Tomáš. *Integration of the IBM SoftLayer into the ManageIQ Framework*. Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Rogalewicz Adam.

Integration of the IBM SoftLayer into the ManageIQ Framework

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. Mgr. Adam Rogalewicz, Ph.D. (FIT BUT) and Mr. Mgr. Martin Povolný (Red Hat Czech, s.r.o). The supplementary information was provided by members of the ManageIQ developer team. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Coufal
May 17, 2016

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisors for the continuous support in my research, for their motivation, and immense knowledge. My sincere thanks goes to the ManageIQ developers team, most importantly to Bc. Ladislav Smola. Especially his guidance helped me throughout the time of research and implementation for this thesis.

Contents

Chapter 1

Introduction

In the past few years the term of Cloud computing resonates worldwide, and gains its popularity. There have been plenty of papers and articles written about it, and every large IT corporation interested in this new market has brought their own solution. Cloud computing has become a well established business, and ultimate answer for nearly every demand for IT infrastructure these days. But when it comes to the meaning of these two words, not everyone knows what exactly to expect. In a nutshell, it means a highly scalable and accessible platform, reachable through a network connection. The word platform in this definition stands for a huge variety of software: from virtual machines and specialized databases to applications like web office suites. In so called *Cloud*, whole internal corporate infrastructures are run along with end-user facing products with ease. Rising complexity and competition between different cloud providers and types of services, creates a demand for an easier management, providing more efficient yet reliable ways to ensure the same level of control over offered services.

In this thesis we describe the main challenges that cloud managing tools are facing while integrating with providers of various capabilities and including different systems and subsystems. The described process is going to be based on a real example, an implemented process of **IBM SoftLayer** cloud service integration into an open-source project, **ManageIQ**. This work covers how to distinguish and differentiate the same functionality across naming conventions and service capabilities, how to integrate them into **ManageIQ** and how to establish working communication between **IBM SoftLayer** and the cloud manager. This thesis also aims to simplify, sort out and sum up the knowledge needed to implement such integration for future **ManageIQ** provider integrators.

1.1 Cloud computing

Before we reveal the complexity of *Cloud computing* and describe the challenges in managing cloud services across providers, it is worth describing what the word *Cloud* actually stands for, why it is a need for huge variety of companies, and how the IT industry invented such technologies[2].

1.1.1 Transition from traditional computing to the cloud

In history, the general approach to implementing a solid and reliable IT infrastructure has changed several times. To understand well the thought process behind this evolution, let me shortly describe the needs and demands of the industry.

Traditional way

The historically first and simplest approach for a company to implement and manage their own service is to use their own machines and servers. To lower the risks of a hardware failure this solution requires to mirror the application and it's data over multiple servers or even into a cluster of serves. This brings a lot of investments and requires a lot of maintenance on the company's side.

Servers are considered a base unit that encapsulates all the necessary hardware, operating systems, storage, and any other utilities necessary. When the application reaches limits of its dedicated server, some additional hardware has to be provided. Despite the fact that an application can consume a lot of resources, it's not happening all the time. As an example, you can think of a delivery or ordering system. During the year the amount of transactions are equal but before e.g. Christmas the peak in transactions can be high. Nevertheless the downside is, once you have the server configured to run one application that can use all its resources during the peak, you can't utilize the resources left unused when the application is idle. Additionally, when the system encounters a failure, the recovery process is complicated. In the matter of scalability this approach is not functional enough.

Virtualized computing

Because of all the disadvantages listed above, a new approach needs to be invented. To lower the complexity of hardware scaling, IT industry moved towards an increase in software difficulty. Unlike hardware maintenance, this can be automated easily and requires less resources to deploy. Servers are no more considered atomic units. The fact that hardware itself can be abstract leads to an invention of *Virtualized computing*. The paradigm of virtualization presents a virtual machine manager also known as a *Hypervisor*. This is a specialized operating system designed to run multiple operating systems as applications. This manager provides the necessary layer that can encapsulate each environment. The isolation of hardware from operating systems makes it possible to run multiple services on one physical machine. Each virtual machine is provided by the resources it demands and when these are left unused, the hypervisor manages to pass them where they are needed.

However, when a physical failure appears, the situation remains unchanged. The service has to be moved to another device. What differs is the solution. Usually, hypervisors are run in clusters of physical servers where they can cooperate. When one hypervisor is facing a hardware failure, the services are smoothly swapped to another physical device managed by a different hypervisor within the same cluster. This can happen without any outage of service and without the need for running a parallel fallback machine. This flexibility also helps the scalability mentioned above. In the case of multiple services running on one physical device, the resources are assinged dynamically. And in a case when an application demands more resources than the hypervisor can offer, the less loaded virtual machines are transitioned to another server within the cluster. This creates an environment where no virtual machine suffers from a significant lack of resources.

Outsourcing virtualization

The core idea behind virtualization is the same for *Cloud Computing* as well. A company using a virtualized solution typically owns the physical servers and maintains them on their own. This produces much overhead costs. On the other hand in a *Cloud Computing* environment, there is no need to insist on keeping and housing the infrastructure by a

company. The operational responsibilities are shifted to the cloud provider who is now responsible for all the hardware and its maintenance. Providers offer remotely controlled virtual environments, location independent and highly scalable solutions. Advantages of virtual computing sustain, applications are still run in a virtual environment, scaled on demand and flexible. The creation of new virtual machines is a matter of minutes or less and no additional resources are needed[1].

Cloud Computing providers usually implement a pay-as-you-go model where all costs are based on actual usage and new appliances are purchased when needed. Advantage of this payment model becomes even more significant when the company has a lot of applications that need to be run concurrently and their transaction peaks are expected at the same period of time. Cloud providers dispose with load balancing mechanisms and thanks to size of their clusters, the availability of the application is always guaranteed. The actual costs still remain much lower compared to the situation when a company has to provide all the hardware on their own. When the peaks diminish, all the necessary additional resources can be reused by the cloud provider for other applications. The same situation in the virtual computing model would lead to a state where these resources would be left unused on the company side. The idea of shared resources in huge clusters is one of the strongest advantages of cloud computing.

1.1.2 Cloud typology

Among all the described advantages of *Cloud Computing*, not only the scalability has to be taken into account. There are plenty of fields where the cloud solution excels in. For example the *National Institute of Standards and Technology* of the USA defines cloud computing by these five most essential characteristics[7]:

1. On-demand self-service
2. Broad network access
3. Resource pooling
4. Rapid elasticity
5. Measured service

On-demand self-service stands for a possibility for consumers to provision computing power (meaning server time, dedicated storage, etc.) as needed and without the necessity to interact with the service provider in person by any means. This allows the customer to avoid the risk of not being able to scale his appliances when there is any kind of outage in the preferred type of communication established between him and the provider.

Broad network access is a term used to describe availability over network via standard communication channels while not discriminating client devices by type or platform. The term is mainly used in context of private clouds where this idea goes slightly against the security principles these clouds are designed for. The main reason to involve broad network access is to make the infrastructure available also for remote workers and via tablets and smartphones.

Resource pooling is a criterion considering dynamic assignment and reassignment of resources to different customers based on their demand in a multi-tenant model of cloud service. These resources are location independent and the customer is neither in control nor

has the knowledge of the exact location of the resources. Nevertheless the location can be revealed on a higher level of abstraction, on a country or data center scale.

Rapid elasticity presumes the resources are provisioned and released automatically. These actions are done in a short period of time and from the customer's point of view the capabilities available typically appear unlimited and any amount of resources can be up-scaled at any time.

Measured service is a term used for automatic control over cloud cluster resources in order to monitor, analyze, control, and optimize the usage. This mechanism provides additional transparency over the service for both the customer and the provider.

Cloud solutions have many shapes and forms in general[4]. To distinguish and differentiate between common types of the *Cloud*, multiple points of view should be mentioned. One of the discerning criteria to be considered is the availability to purchase a different deployment model. There exist private and public clouds. *Public Cloud* means the cloud infrastructure (not the appliances) is widely accessible by anyone. No matter if an organization or a person, anyone is able to use the service provided. The Amazon's *AWS EC2* [1], *OpenShift by Red Hat* [2], *OpenStack* [3], Microsoft's *Azure* [4] or *Google Cloud Platform* [5] can serve as an example of this type of cloud service.

When customers are using a *Public Cloud*, they share the same infrastructure for their virtual machines. In contrast, there is a second option available. These, so called *Private Cloud*s, are strictly used by one customer only, and they are based on a special contract between the cloud service provider and the customer. This provides additional options for control over the purchased infrastructure and more security advantages as well. Since there is no other user in that cloud, it minimizes the risk of any vulnerability, in the isolation of each application, being exploited.

Another option how to differentiate between available cloud solutions is by its level of abstraction: the service models. According to the service-oriented architecture, cloud computing providers offer three main types of service. These are (in stacking order) *Infrastructure as a service*, *Platform as a service* and finally *Software as a service*.

Infrastructure as a service (IaaS)

Firstly, there is the most low-level approach to providing services via cloud, providing base infrastructure. This reflects the need for customized setups which are trusted by customer. In this case the term of infrastructure stands for virtual machines or even bare metal ones. These are usually deployed based on an images built by customers themselves or generic ones which allow to quickly scale over predefined setups. Integrating a provider of this kind is the subject of this thesis.

Platform as a service (PaaS)

By utilizing a PaaS, a customer gains an environment that allows him to develop and run his own applications without the need for building and maintaining a complex infrastructure. Such customer has access to a solid, stable and reliable platform of his desire and focus solely on the application he develops and deploys. The described environment offers countless

[1] http://aws.amazon.com/ec2/
[2] https://www.openshift.com/
[3] https://www.openstack.org/
[4] https://azure.microsoft.com/en-us/
[5] https://cloud.google.com/compute/

setups and frameworks with or without included databases, continuous integration, etc. This approach brings the advantages of rapid, easy and secure deployment, along with other benefits of the cloud.

Software as a service (SaaS)

The most advanced and complex level of abstraction in cloud computing. *Software as a service* usually provides end-user facing applications accessible on demand. The provider installs and operates an application software for the customer in their cloud. Typically *SaaS* is licensed on a subscription basis offering parametrized environment, along with high availability insurance. Great examples of such kind of service are *Salesforce* [6] and web office suites like *Google Docs* [7] or *Microsoft Office 365* [8]. The portfolio of services covered by SaaS is huge and wide beyond imagination. From offering solutions supplying different kinds of analyses like social networks profiling tools and advertisement, over communication platforms including video, audio, mailing services etc., to mobile offices like the ones mentioned above.

1.2 IBM Cloud

Among others, the IBM company also offers their own cloud solution, the so called **IBM Cloud**. It's not a standalone project, it's a summary name for a whole portfolio of products. It comprises of complementary yet independent platforms and tools covering an extensive amount of application and use-cases delivering adjustable setups and products to the customer. When the customer demands IaaS, *IBM SoftLayer* is the product he's asking for. For platform-based requirements, IBM offers a cloud service named *Bluemix*. There is also a vast amount of SaaS solutions delivered by the company via their very own *IBM cloud* market. Some of them based on the *IBM Watson* intelligence, some standalone. This complex tooling allows IBM to deliver in the cloud market environment.

[6] https://www.salesforce.com/
[7] https://docs.google.com/
[8] http://office.microsoft.com/

Chapter 2

IBM SoftLayer

This IaaS cloud service provided by IBM Company is one of the world's largest cloud services available. In order to integrate this provider of cloud infrastructure it's better to acknowledge and recognize the merit. Exploring and studying its parts and internal structure helps to understand the way to a successful integration. The very next step is to identify and describe the ways how the provider can be accessed via APIs and cloud binding libraries and what are the benefits or drawbacks of each option available. Finally, by comparing these approaches, it's necessary to objectify which one is the most suitable for our needs of integration into **ManageIQ**.

2.1 Components

Before elaborating on the cloud bindings it is necessary to introduce and describe the main and most important areas of IaaS clouds, with focus on the **IBM SoftLayer**. The most notable parts that are required to understand are listed below. Each user deploying his machine in cloud needs to decide on the following:

- Where to place the machine?
- How is the virtualization encapsulated?
- What are the resources available for the purchased instance?
- Which software, operating systems are run and how to preserve data?
- How are the machines connected one to another?

Understanding these topics is essential for a customer to be able to successfully and effectively deploy his appliance.

2.1.1 Regions, Zones and Data centers

One of the most important yet changing parts when setting up the cloud infrastructure is to distinguish where the appliance is physically run (approximately). This is important in a case when customer wants to mirror his infrastructure around the globe with purpose to ensure its accessibility and reliability. It's a factor that can easily eliminate or at least reduce the connection issues caused by Internet service providers and exchange points outages. In other words, the application's swiftness is as good as it is possible and not dependent on

its user's geographical location and time. Since neither the customer nor cloud provider is responsible and in control over the connectivity provided to users, placing customer's devices as close as possible to its end-user destination makes sense. This is what usually the *Regions* are referring to. However, in a cloud environment the specification of the exact physical location is not dogmatic. The approximate and relative location is sufficient and the most common way is to refer to a continent or a market. For example the **Google Compute Engine**[3] specifies its regions as: `Central US`, `Eastern US`, `Western Europe`, etc.

On the other hand a complementary entity takes place within each region. These are called *Zones* or *Availability Zones*. Numerous zones are present in each *Region*. Each one is independent of another. The reason is to ensure that in a case of outage, scheduled maintenance or any other kind of issue affecting a zone the others are left untouched and available. When one zone is failing the others in the same region remain available and reliable.

By using and specifying proper *Regions* and *Availability Zones*, the customer can ensure and enhance reliability of his services, reduce latency and build a robust system which is both as close as the customer needs and distributed around the world in the same time. Each cloud provider has a slightly different philosophy of how they comprehend and implement these principles. Let's describe the two most common approaches.

We already mentioned the **Google Compute Engine** where the understanding of the area specifications is probably the most fitting the definition. There are *Regions* which refer to a continent or a country where the data centers are placed. Also, within each of them there are a couple of *Availability Zones*, usually 3 or 4 of them, and these are independent.

As a second example the **Microsoft Azure** provider can be mentioned. Their understanding of this scheme is a bit different. They provide a *Region* based solution only[6]. These are specified in much greater detail then in Google's case and there exist more of them as well. This provider is substituting *Zones* by making regions smaller so customers can easily target the end-user (by selecting not just the continent but even a specific country) and yet keep the backup instance in the closest region. As an example, this is a sample how the *Regions* are named and where they are placed in the **Azure** cloud provider: `Central US` in Iowa, `North Central US` placed in Illinois or `Japan West` set in Osaka.

Let's focus on how the scheme of*Regions* and *Zones* works in the **IBM SoftLayer** cloud provider. Their model is quite similar to the **Microsoft Azure**. However, besides other differences they do not use the term of *Region*. Instead, the *Data Center* term is used. This term truly refers to an exact location (for each center, the exact city is told). The product pages describe locations where data centers are placed[8], for instance such centers are: `Dallas 01`, `Dallas 09`, `Amsterdam 02`, `Washington, D.C. 01` or just `Paris`. As you noticed, there might be some redundancy within a location. When there are multiple data centers in one city, each affected data center is numbered.

Speaking in the matter of the scheme defined in previous paragraphs, it's definitely needed to fit these in. It's required to find a proper category, because the scheme is reflecting the principles implemented in **ManageIQ**.

At first, let's take a look at how the work flow goes for the **IBM SoftLayer**. When the user is provisioning a new appliance he needs to select a *Data Center* of deployment. This is the same situation as when deploying into **Google Compute Engine**, where the customer has to select the *Region*.

Another point of view is that when there are multiple data centers within one city (let's say in the same *Region*) the situation is reminiscent to a state when the provider offers

multiple *Availability Zones* for the region. Such behavior is reflected especially by these essential competences:

1. Data centers are in the same *Region*.
2. Each one is independent on another.

This view can evoke in us the idea that **IBM SoftLayer** provider is actually offering *Availability Zones* as well, and it is mixing them with *Regions* and calls them *Data Centers*.

On the other hand, each *Data Center* is selectable. In other words, it is required to pick one of them and deploy the appliance there. But this view goes strictly against the presumed policy that zones are managed internally by the provider and the user is usually not able to select and specify which zone is being used for his device. And there's also another reason why the comparison of zones and data centers within the same location is not accurate. There's no internal connection (besides the name) to link a *Data Center* to the others within the same city. The missing relation (via API or network, etc.) is finally the reason why to model every data center as a separate *Region* and do not bother the user with a new layer, new construct, the *Availability Zones*.

2.1.2 Servers

Once the user figures out where he can run his appliances, it's worth discovering what is being provisioned and run. In the clouds of the *IaaS* type, it is usually some kind of *Server*. Previously it's been mentioned 1.1.1 that the most common type of device is a *Virtual Server*, but this is not the only commodity available. For numerous security reasons, some cloud providers offer the possibility to run such a virtual server in three types of environment. **IBM SoftLayer** provides all of them.

Shared hardware

To run a *Virtual Server* on shared hardware is the cheapest variant. It is also suitable for most of the customer needs. When a customer does not demand any special treatment like enhanced security features or a specific type of hardware, this is the way to go. The provisioned virtual server is placed on a server within the *Data Center* and under a *Hypervisor*. This hypervisor also manages other appliances and does not differentiate between the customers, the owners of the virtual machines. Shared hardware means a shared environment in a sense of a communal hypervisor and a shared physical layer between users.

Dedicated hardware

The opposite option is to reserve *dedicated hardware*. It's more expensive, but on the other hand more secure. By using a dedicated machine for appliances owned by one user only, it provides another layer to secure the data. Also, some providers can offer an option to let the user be responsible over the *Hypervisor* too. In addition, running appliances on *dedicated hardware* makes the customer in complete control over the leased infrastructure.

Bare metal servers

There's also an option, offered only by a minority of companies, to use the cloud to provide so called *bare metal servers*. This means a completely different approach than cloud is

usually known for. If a user wants to keep his data super secure and isolated yet in a stable, reliable and affordable environment it's possible to order a specific physical rack and run the customer's server there. Basically, this approach resembles a kind of server housing with the benefits of the cloud.

2.1.3 Hardware Flavors

Once the customer knows where the virtual machine should be run and what kind of virtualization the deployed setup requires, it's worth choosing the hardware resources (no matter if virtual or physical). This specification involves aspects like the processor cores count and frequency, amount of memory available for the device, or how data are being stored (if they are stored on a local hard drive or available via network, what type of hard drive it should be, how many of them are attached, what should be their capacity etc.). All these properties can be specified manually and in special cases they are. However, the more common work flow is to store the favorite setups as *Flavors*. Each cloud provider also offers some default ones. For example the Table 2.1 below describes the default *Flavors* used in the **IBM SoftLayer** cloud.

Table 2.1: IBM SoftLayer default *Flavors*

Identifier	Name	CPU cores	Memory (RAM)	Hard drives (HDD)
m1.tiny	Tiny Instance	1	1 GB	1 × 25 GB
m1.small	Small Instance	2	2 GB	1 × 100 GB
m1.medium	Medium Instance	4	4 GB	1 × 500 GB
m1.large	Large Instance	8	8 GB	1 × 750 GB
m1.xlarge	Extra Large Instance	16	16 GB	1 × 1 000 GB

2.1.4 Images, Templates and Snapshots

Purposes of *Images* are to replicate a virtual server running in cloud, store setups or deploy preconfigured systems. Some cloud providers and managers use a term *Template* instead. They can be created in two possible ways.

The provider can produce some basic images with operating systems based on the default installation setups adjusted to reflect the cloud specific features, configurations, etc. Such images can also provide preconfigured application setups or platforms. As has been said, these templates are usually prepared by the provider to facilitate initial setups for the customer. Of course a user can deploy such images on his own too. But since this type of templates contains a default configuration, it's much more convenient for the customer to be provided with them.

There's also a second way how to create an image. There's a possibility to create a *Snapshot* of a running virtual machine. It is a pretty essential feature for each cloud to prepare a setup and save it as a template for backup and redeployment purposes. The user usually wants to scale his infrastructure and distribute it around the world in different regions or providers. By producing snapshots of his running virtual machines or uploading his own preconfigured ones it's easy to preserve state, data and environment and deploy, copy the instance, elsewhere. These *Images* and *Snapshots* created by a customer can be flagged as private and available for his own use only, or there's also a possibility to make such *Template* publicly available for other users of that cloud.

2.1.5 Networking

Finally, the last of the remarkable areas of IaaS clouds — networking of the appliances. While a customer is provisioning a virtual machine, this machine is normally a part of a greater infrastructure. Inside that complex system, its components need to be connected and linked with others. In order to achieve that, clouds and virtualized computing brings mechanisms of *Virtual networks*. These networks are modeled to behave and to offer the same functionality as their physical equivalents. There are couple of essential network infrastructure components. A list of the ones provided by **IBM SoftLayer** follows.

Network Port

Each virtual server is provided with some network interfaces with their own name, MAC address, etc. They are usually called *Metwork Ports*. They point to the network interface properties of the virtual machine but they are also propagated outside to the managing API and provider's service applications. Thanks to this propagation they can be dynamically modified via API or the tooling provided by the cloud provider.

Cloud Networks and Subents

Each port can be connected to a virtual *Cloud Network* or a *Cloud Subnet* while they work the same as in the world of regular computer networks. They are provided by IP address ranges (so the device can be connected, receive a mapping to one of them), speed limits, etc. They can also be controlled by *Firewalls*. However, the understanding of cloud networks and subnets is also different across providers.

For example, let's describe the meaning of cloud networks and how they work in **IBM SoftLayer**. Each virtual server is given access to the two default networks for the data center. It is a *Private Cloud Network* and the *Public Cloud Network*. They take place of a gate to different points of interest.

Private Cloud Network connects the device to internal appliances within the data center. For instance, such device can mean network storage volumes or databases, etc. On the other hand, the *Public Cloud Network* is a gateway to the outer world, outside the data center and the cloud. To be precise the cloud networks in this sense do not offer any routing capabilities, they work just as a label of the range of interests available (reachable resources) for the device. Inside these networks, *Cloud Subnets* can be set up. The subnets behave like the real networks or subnets. A customer can specify IP ranges and all other parameters he needs. The amount of subnets in the same network is not strictly limited for this cloud provider.

Network Router

Apart from the attached devices and addresses each network needs its *Network Router*. This router is not a physical device either. For each subnetwork (as they are defined in the **IBM SoftLayer**) exists exactly one such router. Usually they are given a name and IP address and the only use for them is to build a proper illusion of the real network. Each router can service multiple subnets as it is known from regular computer networking. Within each cloud network there can be multiple routers. However, the architecture used in **IBM SoftLayer** does not provide more complex hierarchy, more levels of subnetworks. On the contrary, any advanced networking is not very common for cloud infrastructures currently deployed and this flat approach is sufficient enough.

2.2 API access

For managing purposes each cloud provider has it's own web interface. Such a tool usually contains all needed functionality: visualization of leased devices and machines, networking adjustments, provisioning of new appliances, creating snapshots, viewing spendings and other billing information.

On the other hand, an API is needed once more advanced users want to automate their work flows or create scripts to handle some common tasks. There are numerous different libraries allowing a user to connect to the provider. In this case it is essential to focus on the **IBM SoftLayer** provider with respect to the needs and capabilities of **ManageIQ**.

2.2.1 Standard REST API

The standard way offered by provider is to use the **IBM SoftLayer**'s *REST API*[9]. This extensive API offers a complete access to nearly all features of the cloud infrastructure. It is a low level standard defining how to communicate with the cloud, how to format requests and what responses should be expected etc. The major advantages of the API is in its complexity and independence on a programming language.

User authentication

When a user is managing the cloud services via web interface, the normal and most common way to identify himself is to use *username* and *password*. While using APIs, the situation is a bit different. Each cloud defines the way on their own. **Google Compute Engine**, for instance, requires to specify the project name, user email and then uses a special *Google JSON key* which is basically a project specific certificate for the user. The **Amazon Web Services EC2** cloud uses a generated pair of *Access key ID* and a *Secret Access key*. Finally, when it comes to the **IBM SoftLayer**, there are two factors used for authentication. It involves the normal *User ID* (a customer specific ID) and a specially generated *Secret API key* for each of the customer's administrator.

Language bindings

The described API is pretty basic and low level in the aspect of connection handling and abstract operations. For better integration of this API into user projects the derived libraries have been built upon it to offer bindings and object based interface for different programing languages. Since **ManageIQ** is written in *Ruby* we should focus and elaborate on the libraries created for this programming language.

2.2.2 Softlayer API for Ruby

First of them is a *SoftLayer API* brought by the SoftLayer developers[10]. This library provides a Ruby Gem package named `softlayer_api`. However, because of the lack of proper structure and disorganized code, it is not easy to follow for beginners. All provided activities are available via one service and the naming conventions abide by the **IBM Softlayer** standards. This makes it a bit complicated in the matter of maintenance while the provider is being integrated with others. By using this library, each developer willing to maintain the code has to understand internal **IBM Softlayer** work flows. The example Code 2.1 describes the basic usage of this library.

Code 2.1: Example code for the `softlayer_api`

```
require "softlayer_api"

# Specify the provider and credentials
SoftLayer::Client.default_client = SoftLayer::Client.new(
  :username => "<username>",
  :api_key  => "<api_key>"
)
# Connect
account = SoftLayer::Account.account_for_client()

# Lookup all provisioned servers
account.servers

=> [<SoftLayer::VirtualServer:0x00000002090ec8
      @softlayer_client=
        <SoftLayer::Client:0x00000002124ce0
        # client details like used credentials, API, etc.
        >
      @softlayer_hash=
        { "domain"=>"example.com",
          "fullyQualifiedDomainName"=>"centos.example.com",
          "hostname"=>"centos",
          "id"=>17784479,
          "maxCpu"=>1,
          "maxMemory"=>1024,
          # etc.
        }
   ]

# Get the first server and check its state
server = SoftLayer::VirtualServer.server_with_id(17784479)
server[:powerState]

=> 'Running'
```

2.2.3 Fog the Ruby cloud service library

There's also a `fog-softlayer` gem, a *Fog* library for this provider[11]. *Fog* is a cloud service library for Ruby available for many different providers across market. This makes the Gem easily understandable and compatible with other providers. It is also much simpler to maintain the structure and follow a pattern of other providers already available in the **ManageIQ** which are also implemented via *Fog*. This library also offers extensive documentation (for *Fog* in general) and example code for the `fog-softlayer` covering the desired functionality.

To contrast the `fog-softlayer` with `softlayer_api`, the services and objects are more structured here. They also offer standardized cloud naming conventions which make it easier to follow and maintain. Its structure corresponds with every other provider in *Fog*. It is kept and managed via 5 basic services which can operate separately.

- `Fog::Account` accessing customer account's organization if it is grouped with others
- `Fog::Compute` is the most important service allowing the user to control servers (monitoring, provisioning, deployment, creating snapshots and more)
- `Fog::Network` offers bindings to manage cloud networks, subnets, routers, etc.
- `Fog::DNS` for managing DNS records
- `Fog::Storage` provides connection to *Bluemix* storage service

Each of the services has its own purpose, but the most important in a sense of cloud management are the `Fog::Compute` and `Fog::Network`. By using just these two, the user can easily manage his running appliances and deploy new ones. The code sample 2.2 shows the basic work flow for listing running servers and how to provision a new one.

Code 2.2: Example work flow for Fog SoftLayer

```
require "fog/softlayer"

# Specify the provider and credentials
options = {
  :provider => "softlayer",
  :softlayer_username => "<username>",
  :softlayer_api_key  => "<api_key>"
}

# Connect to the Compute and Network service
compute = ::Fog::Compute.new(options)
network = ::Fog::Network.new(options)

# Lookup all provisioned servers
compute.servers.all

=>   <Fog::Compute::Softlayer::Servers
    [ <Fog::Compute::Softlayer::Server
        id=17784479,
        name="centos",
        domain="example.com",
        fqdn="centos.example.com",
        cpu=1,
        ram=1024,
        # etc.
      >,
      # other servers
    ] >

# Get the first server and check its state
server = compute.servers.get(17784479)
server.state

=> 'Running'

# Provision a new instance from image
provison_options = {
    :flavor_id => "m1.small",
    :image_id  => "1394bf94-e4e5-43bf-90ec-5eedbdcc420d",
    :name      => "ubuntu",
    :domain    => "example.com",
    :datacenter => "ams01"
}
new_instance = compute.servers.create(provision_options)
new_instance.id

=> 17784894
```

Chapter 3

ManageIQ

A project named **ManageIQ** is an open-source technology developed by a community supported and sponsored by the Red Hat company. This project aims to provide an easy management over cloud solutions across providers by offering comfortable import, appliances management, network links visualization allowing modifications, and infrastructure provisioning capabilities[5]. Expectations are high since all integrated technologies are different considering functionality and capability. To cover those variations, **ManageIQ** provides abstract internal bindings that should suit most of the needs.

3.1 Providers

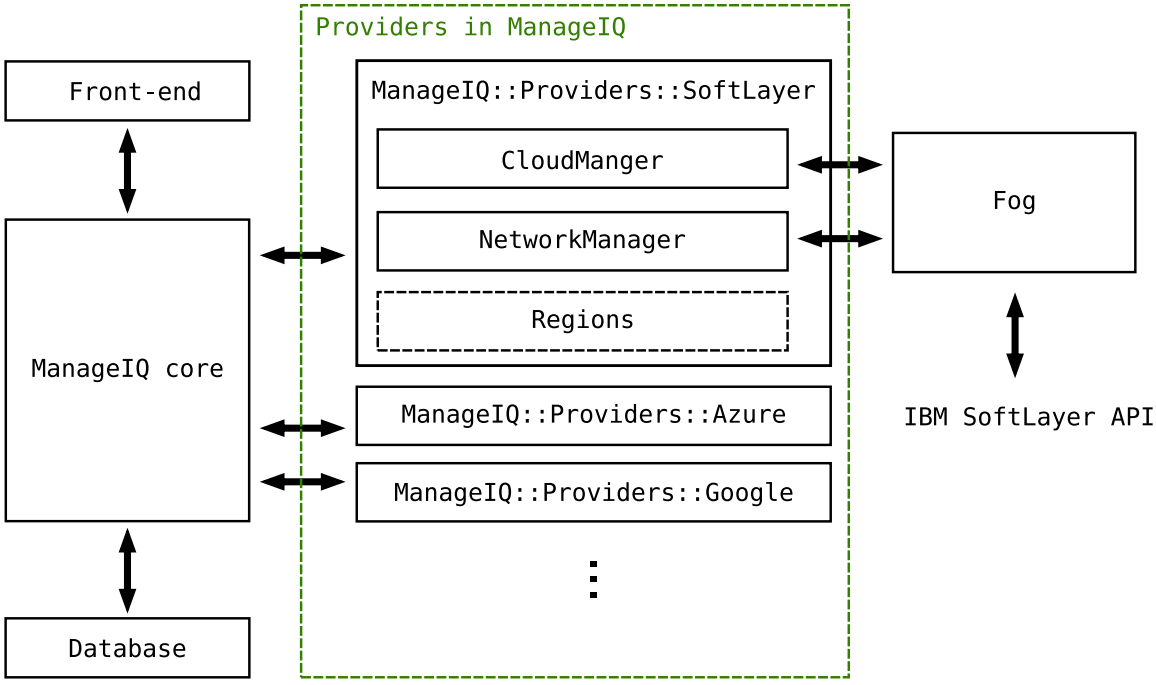


Figure 3.1: Providers in *ManageIQ*

The Developer's guide[12] defines available bindings for each provider. Each provider is represented by a so called *External Management System*. The reason for such name is mainly

historic, today each provider offers a complex functionality which needs to be broken into smaller, separate managers. A Figure 3.1 categorize the providers into the **ManageIQ**'s architecture. This reveals the overall picture how these managers are related with other parts of the framework. In the case of **IBM SoftLayer** these are the *Cloud Manager* and the *Network Manager*. Each of them is given it's namespace and can be derived from a base manager of that type. These managers cover, or try to cover these areas:

- *Inventory* for listing and tracking all the cloud properties
- *Event Collection and Handling* for event driven work flows and dynamic system
- *Metric Collection and Handling* for analysis over inventory objects, utilization etc.
- *Provisioning and Orchestration* is adding features to dynamically deploy new setups in the provider
- *Lifecycle* for managing the already deployed ones
- *SmartState Analysis* is a low-level analysis tool for *Virtual Machines* and *Images*

Currently, due to missing features in the API for implemented provider we are able to cover just the *Inventory*, *Provisioning and Orchestration*, and *Lifecycle* features.

In general, there exist three ways how to collect data about the provider:

- *Refresh Worker*
- *Event Worker*
- *Metric Worker*

Because of the reasons described above, we are currently able to implement and use just the first one of them, the *Refresh Worker*. The properties imported by the refresh are listed and described below.

3.2 Implemented Interfaces and Models

ManageIQ uses a hierarchical scheme of models. While adding a new provider it is a `ManageIQ::Providers` which has to be implemented. For **IBM SoftLayer** as a cloud provider there are 3 main parts, classes required:

- `"ManageIQ::Providers::Softlayer::CloudManager"` for the appliances management
- `"ManageIQ::Providers::Softlayer::NetworkManager"` for networking
- `"ManageIQ::Providers::Regions"` enlisting all available *Regions*

Both of the managers inherit from their *Base Manager*, for example, for the *Cloud Manager* it is `"ManageIQ::Providers::CloudManager"`. These managers implement the behavioral principles for *Cloud Refresh* or event driven management purposes. In order to store connection bindings, which are common for both, there is a special *Manager Mixin*.

These models are places in the `app/models/manageiq/providers/<provider>` folder where `<provider>` address the implemented provider. As this thesis describes the implementation of **IBM SoftLayer**, let's consider as a `<provider_root>` the folder `app/models/manageiq/providers/softlayer`.

In the `app/models/manageiq/providers/softlayer/` folder there can be found (the main components):

- `regions.rb` as a *Regions* storage
- `manager_mixin.rb` for the *Manager Mixin*
- `cloud_manager.rb` defining the *Cloud Manager* behavior
- `cloud_manager` folder containing all components required by the cloud manager
- `network_manager.rb` containing the *Network Manager*
- `network_manager` folder with content required by the required by the network manager

The following sections describe each one of them.

3.3 Connection Management

The connection is a common feature and requirement for both of the implemented managers. This connection handler is kept separated from them for easier maintenance. The class for that is called *Manager Mixin*. This mixin also contains the basic *Discovery* work flow which is described bellow in the Section 3.5. But the main purpose of the mixin remains in establishing the connection via *Fog*, verifying credentials and providing the manager with proper service. How to connect via *Fog* to the **IBM SoftLayer** is described in the Code 2.2.

3.4 Regions

What *Regions* are has already been described. However, for the integration purposes it's needed to understand how they are treated in the **ManageIQ**.

When a user is adding a new provider he has to specify the *Region*. Basically, he is adding a *Region* specific provider. This is a policy intended to keep the UI and the overall environment clean for the user. Imagine a situation where such a user has his devices deployed in many providers. In the case when all *Regions* of each provider are added, his interface can be flooded by them. The other approach is to add one *Region* per provider. And once the user needs to attach another one, he adds a new provider.

It is also required to list all available regions before a user connects to the provider. The application uses one step form for adding a provider, so he has to be able to specify the *Region* before his credentials are proceed. And since every API client requires to authenticate before any actions can be done, it is required to list all regions statically in a file `<provider_root>regions.rb`.

3.5 Discovery

The work flow described above can be slightly inconvenient when the user already has devices deployed in many different *Regions*. For such situations, there is a *Discovery* process. There, the user just specifies his credentials and the process does the work for him. The application places his discovery into a tasks queue and when the discovery proceeds, it iterates over every *Region*. While there are *Virtual machines* present in that region, it is registered as a new provider and its *Refresh* is scheduled. This work flow is specified in the `ManagerMixin` class.

3.6 Cloud Manager

Cloud Manager is a class specified for each provider and inherits from a base `CloudManager`. In this case it is specified as `ManageIQ::Providers::Softlayer::CloudManager`. It provides an interface over the cloud devices. The main purpose of this class is to cover all functionality needed to refresh and manage *Virtual Servers* and every other aspect needed in order to provision and access them. All the links between devices are discovered separately and this class does not care about the internal representation of network components. Nevertheless, this manager maps the networking identificators of discovered devices into an internal database and once the networks are discovered, the system links them together.

All the entities and submodules described in this manager are populated by the *Cloud Refresh*. There are also implemented relations and delegations for networking properties into the *Network Manager*.

3.6.1 Cloud Refresh

To run a discovery service or refresh of the provider, the *Cloud Refresh* namespace has to be defined. This is the most important class for each manager. It covers the import and mapping of every supported entity which is watched by the manager. The *Refresh* procedure itself is defined and proceed in three separate phases:

- `:Refresher`
- `:RefreshWorker` and `:Runner`
- `:RefreshParser`

This *Refresh* is the base and most important back-end part of a new provider. It's main role is to fill the internal database with all inventory needed for further usage. The scheme that defines which attributes and inventory models are available is prescribed in the Developer's guide and documentation[12] in the *Providers database architecture* and *Provider Overview* guides.

Refresher

The `:Refresher` specifies the work flow of the refresh. It tells the **ManageIQ** what refresh should be run, how to store all discovered appliances and which database schemes and tables should be affected. The major issue which this class takes care of is to identify which provider the devices belongs to. This refresher, when initiated, also queues a refresh of the providers *Network Manager*.

Refresh Worker

The worker is a specification of the way an import is handled. Normally, no modification to the standard process is required. It's the same for this provider. Each provider needs a `:RefreshWorker` in order to register the run of refresh. The naming conventions and scheme of **ManageIQ** require also the `:Runner` sub-class to be defined. Both of them inherit all the functionality from the base cloud manager's refresh worker.

Refresh Parser

This class is the most important part of the refresh process. It defines which devices are added and where these should be registered in **ManageIQ**. Here, it is specified how the mapping of each of the following entities is done between the provider client library (in the case of **IBM SoftLayer** it is `fog-softlayer`) and the internal structure. The mapping (whether the device is *Virtual Server* or *Flavor* etc. and its properties) where to store these information is specified as well.

3.6.2 Availability Zones

As we elaborated in the Subsection 2.1.1, there are no sufficient *Availability Zones* present in **IBM SoftLayer**. On the other hand, each cloud manager requires a *Zone* to be defined. To satisfy this need, a new default zone has been created. It does not affect any operation over the provider, although for the internal needs of the data hierarchy inside **ManageIQ**'s database it is present.

To preserve the naming conventions of **ManageIQ** even in this case, it is required to use two separate mapping functions. The first one, normally used to fetch data from the provider, now creates a list containing one element, the default zone. Then, it lets to parse it via the second function and store it into the designated place. The model for the zone uses a modified instance of `Fog::Model` as the default zone 3.1. This approach has been used instead of defining a special class for the zone because there is no need to have such a class available in the namespace. There is always going to be only one instance present in total and only during the refresh.

Code 3.1: Declaration of the default *Availability Zone*

```
# Create new fog model
default_zone = ::Fog::Model.new

# Inject methods for :name and :id to return default values for provider
{:name => @ems.name, :id => 'default'}.each do |method, value|
  default_zone.define_singleton_method(method) { value }
end

a_zones = [default_zone]
```

Table 3.1: *Availability Zone* attributes mapping

Attribute	Corresponding value
ID	`availability_zone.id.downcase`
`:type`	`ManageIQ::Providers::Softlayer::CloudManager::AvailabilityZone.name`
`:ems_ref`	`availability_zone.id.downcase`
`:name`	`availability_zone.name`

3.6.3 Authentication Key Pairs

Since **ManageIQ** aims to completely manage over different providers, besides monitoring purposes, it is essential to provide user with the possibility to connect to the machine. While there are any *Authentication Key Pairs* attached to the device this is the class which takes care of them. Each key pair, when created, contains a name and a fingerprint. This provider specifies the key pair with a label as a name and a certificate as a fingerprint. Fetching *Authentication Key Pairs* from the provider is easily done via *Fog* by a simple run of `compute.key_pairs.all` command.

Table 3.2: *Authentication Key Pair* attributes mapping

Attribute	Corresponding value
ID	`key_pair.id`
`:type`	`ManageIQ::Providers::Softlayer::CloudManager::AuthKeyPair.name`
`:name`	`key_pair.label`
`:fingerprint`	`key_pair.key`

3.6.4 Flavors

What *Flavors* are was already discussed in Flavors **??**. *Fog* allows to list all available *Flavors* via `compute.flavors.all` command. The Code 3.2. All data available about the flavor are listed in plain format except the disk sizes. The total size has to be counted for each *Flavor* separately by summing the sizes of all available disks.

Code 3.2: Sample of *Flavors* data

```
# Fetch Flavors from SoftLayer
compute.flavors.first
=>   <Fog::Compute::Softlayer::Flavor
    id="m1.tiny",
    cpu=1,
    disk=[{"device"=>0, "diskImage"=>{"capacity"=>25}}],
    name="Tiny␣Instance",
    ram=1024
  >
```

Table 3.3: *Flavor* attributes mapping

Attribute	Corresponding value
ID	`flavor.id`
`:type`	`ManageIQ::Providers::Softlayer::CloudManager::Flavor.name`
`:ems_ref`	`flavor.id`
`:name`	`flavor.id`
`:description`	`flavor.name`
`:cpus`	`flavor.cpu`
`:cpu_cores`	`flavor.cpu`
`:memory`	`flavor.ram * 1.megabyte`
`:root_disk_size`	Counted separately, data taken from `flavor.disk`

3.6.5 Virtual Machines

This is the most important commodity that has been imported and discovered. Each *Virtual Machine* carries information about its name, domain, hardware configuration and even *Flavor*, *Image* which it is based upon or *Authentication key pairs* attached. The attribute mapping is described in Table 3.4. Every attribute listed is successfully mapped except the *Disks* discovery. The *Softlayer API* described in the Subsection 2.2.1 provides every piece of information needed, except the effective disk size. Currently it's been decided to wait until the support for this value is in place. For now, no disks are mapped for any *Virtual Machine* in this provider but the functionality is prepared so once these data are available, it can be easily updated.

Each *Virtual Machine*, once mapped into the **ManageIQ**, offers an interface for its power management. This management also has to be binded to propagate the requests into the provider. This is done via the sub-class `:VM` and its management modules `:Operations::Power` and `:Operations::Guest`. **IBM Softlayer** and `fog-softlayer` allows to:

- Switch off the instance via `instace.stop`
- Turn back on using `instance.start`
- Reboot the appliance by `instance.reboot`
- And finally terminate it by invoking the `instance.destroy` command

Unfortunately **IBM SoftLayer** does not offer to *Suspend*, *Pause* or *Hibernate* the instance.

Table 3.4: *Virtual Machine* attributes mapping

Attribute	Corresponding value
ID	`instance.id.to_s`
`:type`	`ManageIQ::Providers::Softlayer::CloudManager::VM.name`
`:uid_ems`	`instance.id.to_s`
`:ems_ref`	`instance.id.to_s`
`:name`	`"#{instance.name}.#{instance.domain}"`
`:vendor`	`"softlayer"`
`:raw_power_state`	`instance.state`
`:flavor`	`instance.flavor_id`
`:operating_system`	`{:product_name => instance.os_code}`
`:availability_zone`	Fetched the defaulf *Availability Zone* from data storage
`:key_pairs`	Fetched from data storage
Hash for `:hardware` information	
`:cpu_sockets`	`instance.cpu`
`:cpu_total_cores`	`instance.cpu`
`:cpu_cores_per_socket`	1
`:memory_mb`	`instance.ram`
`:disks`	Not effectively populated

3.6.6 Image Templates

In each *Region*, there are specific *Images* available. There are *Public Images* created by other users or made publicly available by the cloud provider. Also, there are *Private Images* created just for the needs of the customer. While importing them, we need to distinguish which images belong to what category. Also, there are other aspects that would be great to categorize. Unfortunately, this is not available via staging APIs. For example we would like to be able to categorize an image by operating system or designated and required disk space. It would also be helpful to differentiate which images are available for purchase and which are already old and their base operating system image is already missing. Currently, none of this functionality is available via API and just the basic data is accessible. Because of the defect mentioned above, the additional features have not been implemented. The mapping used for images can be seen below.

Table 3.5: *Image* attributes mapping

Attribute	Corresponding value
ID	`image.id`
`:type`	`ManageIQ::Providers::Softlayer::CloudManager::Template.name`
`:uid_ems`	`image.id`
`:ems_ref`	`image.id`
`:name`	`image.name`
`:vendor`	`"softlayer"`
`:tempalte`	`true`
`:publicly_available`	`image.public?`

3.6.7 Tags

One of the most important aspects of **ManageIQ** is the capability to *Tag* user's devices across providers and types. **IBM SoftLayer** also offers tagging capabilities. Each virtual machine can be tagged and labeled, and multiple labels can be applied on the device. The same can be said about the **ManageIQ**. However, this provider offers such functionality just for provisioned appliances, but not for other types of devices like *Cloud Networks*, *Network Routers* or *Images*. This makes the import of tags from provider less efficient. There's also another problem that blocks the correct usage of tags in the cloud manager.

The issue is that **ManageIQ** currently does not offer a way to propagate tag changes into the provider. So, in a situation when a user modifies tags of a server in the cloud manager and then wants to refresh the provider, old *Tags* are imported again and his changes are overwritten. Due to this fact, the tags are currently being ignored and skipped during refresh and discovery.

3.6.8 Provision

The default provision model used in **ManageIQ** is *Cloning*. But it does not refer to cloning in a sense of replicating an already available instance. It is used for the *Image* deployment into a new *Virtal Machine*. The work flow basically follows the standard form used by the `fog-softlayer` seen in the Code 2.2. The user selects the desired *Image*, specifies *Flavor*, *Cloud Subnetworks* via web UI and declares the name and domain to use for the newly created device.

The code in the back-end then processes user input and based on his selection, prepares a `cloning_options` hash. The source code bellow describes a function to prepare such data for **IBM Softlayer**.

Code 3.3: Prepare cloning options

```
def prepare_for_clone_task
  clone_options = super
  ems = source.try(:ext_management_system)
  vlan_id = get_option(:cloud_network)
  # NOTE: Private vlan is represented in the provisioning form
  # as a :cloud_subnet (rendered as a same type so there's no
  # need for a new specialized field)
  private_vlan_id = get_option(:cloud_subnet)

  # NOTE: Cloning might fail for some images due to missing base
  # OS (in the image). This information is not available via API,
  # it's safe to use own images though.
  additional_options = {
    :flavor_id    => instance_type.name,
    :image_id     => source.uid_ems,
    :name         => dest_name,
    :domain       => get_option(:vm_domain),
    :datacenter   => ems.provider_region,
    :vlan         => find_cloud_network_in_vmdb(vlan_id),
    :private_vlan => find_cloud_network_in_vmdb(private_vlan_id),
  }

  clone_options.merge(additional_options)
end
```

3.7 Network Manager

In order to connect and visualize relations of discovered devices, **ManageIQ** brings the class `NetworkManager`. A separate *Network Manager* is currently a new feature included in **ManageIQ**. The **IBM Softlayer** is one of the first providers adapting this structure. The current implementation uses the *Cloud Manager* as default for delegating some of the workload onto this manager. This manager has it's own refresh capabilities and the refresh and discovery is it's main purpose. Today, neither **ManageIQ**, nor `fog-softlayer` is capable of manipulating networks. This layer is static for display and monitoring purposes.

The main change in the process of connecting to the provider's API is using the `Fog::Network` service instead of the `Fog::Compute`. The connection to this service is made analogically as for the compute service so the following text is referring to the `network` variable described at Code 2.2.

3.7.1 Network Refresh

In the core, the purpose of *Network Refresh* is the same as for the *Cloud Refresh*. And the work flow is also the same: this refresh requires the same steps to be implemented (`:Refresher`, `:RefreshWorker`, `:Runner` and `:RefreshParser`). They provide the same role as has already been explained.

The `:RefreshParser` is capable of exploration over:

- *Cloud Networks* and *Cloud Subetworks*
- *Network Ports*
- *Network Routers*

Each component's discovery is initiated via a call to a corresponding function. In network discovery, these functions are registered: `get_cloud_network`, `get_network_ports`, `get_cloud_subnets\,(cloud_network)` and `get_network_routers`.

3.7.2 Cloud Networks

It's already been described how **IBM Softlayer** treats *Cloud Networks*. They are used as labels without networking attributes, yet defining the connection and subnetwork purposes. There exist two types of such networks and a specific sub-class has been created for each of them:

- `:CloudNetwork::Public` for public networks that interface the Internet and outer connections outside of the data center.
- `:CloudNetwork::Private` which purpose is to connect devices with each other and with network storage or any other internal service within the data center

Table 3.6: *Cloud Network* attributes mapping

Attribute	Corresponding value
ID	`cloud_network.id.to_s`
`:type`	`"ManageIQ::Providers::Softlayer::NetworkManager::CloudNetwork"` + suffix (`"::Private"` or `"::Public"`)
`:ems_ref`	`cloud_network.id.to_s`
`:name`	`cloud_network.name` if present or a custom string `"Public"` (or `"Public"`) + `"VLAN on <hostname>"`
`:status`	`"active"`
`:cidr`	`nil`
`:enabled`	`true`
`:cloud_subnets`	Initiated discovery and fetched the data from local storage
`:network_router`	`cloud_network.router`

3.7.3 Cloud Subnetworks

In case of *Cloud Subnets* the situation is a bit more difficult. These networks are not directly displayed via *Fog* like the other network components. However, both entities using and interfacing these subnetworks contain desired data. These two entities are *Cloud Networks* and *Network Ports*. Each of them provides enough data to register a subnet. It is considered a better practice while describing components to start from the greater picture to smaller parts. The *Cloud Network* can be considered a bigger and more important component because it can contain more *Network Ports* and represents a parent for each *Cloud Subnet*.

While running a discovery over a *Cloud Network*, all related subnetworks are discovered as well. This behavior is ensured by calling `get_cloud_subnets(cloud_network)` during parsing action for each network. When all subnets are found, they are parsed, registered and stored in **ManageIQ**'s local storage. Later, when the refresh process reaches the *Network Ports* discovery, their identificators are found and they can be located in the database and linked to the port.

Table 3.7: *Cloud Subnetwork* attributes mapping

Attribute	Corresponding value
ID	`cloud_subnet.id.to_s`
`:type`	`"ManageIQ::Providers::Softlayer::NetworkManager::CloudSubnet"`
`:ems_ref`	`cloud_subnet.id.to_s`
`:name`	`cloud_network_name`
`:cidr`	`"#{cloud_subnet.network_id}/#{subnet.cidr}"`
`:ip_version`	`cloud_subnet.ip_version`
`:network_protocol`	`"ipv#{cloud_subnet.ip_version}"`
`:gateway`	`cloud_subnet.gateway_ip`
`:availability_zone`	Fetch the default *Availability Zone*

3.7.4 Network Ports

Running the *Network Ports* refresh process is the next step in order to complete whole network discovery. *Network Ports* are used as a virtual replacement of physical network interfaces. Their data are available via querying each *Virtual Machine*. To successfully access it, the refresh process of the corresponding *Cloud Manager* has to be run before the *Network Manager*.

Since the *Network Subnets* were already discovered using the networks, now it only remains to map them to maintain and register the relation inside **ManageIQ**. This is done by a special mapping function. When specifying a mapping key `:cloud_subnet_network_ports` in the stored data hash, the value is internally converted into opposite relation before saving and correctly mapped onto relevant *Network Subnets*. This is due to **ManageIQ**'s internal mapping and database scheme where the information for each *Network Port* to *Cloud Subnet* relation is stored inside the *Cloud Subnet*. This is due to the fact that some providers are capable of mapping one *Network Port* on more subnets.

Code 3.4: *Network port* to *Cloud subnet* mapping

```
# Using a converter for a relation in opposite direction
network_port[:cloud_subnet_network_ports] = [{
  :address      => network_port.primary_ip_address,
  :cloud_subnet => @data_index.fetch_path(:cloud_subnets, subnet_id)
}]
```

Table 3.8: *Network Port* attributes mapping

Attribute	Corresponding value
ID	`network_port.id.to_s`
`:type`	`"ManageIQ::Providers::Softlayer::NetworkManager::NetworkPort"`
`:ems_ref`	`network_port.id.to_s`
`:name`	`"#{network_port.name}#{network_port.port}"` if provided `network_port.mac_address` otherwise
`:status`	`network_port.status.downcase`
`:mac_address`	`network_port.mac_address`
`:device_ref`	`device_ref`
`:device`	`parent_manager_fetch_path(:vms, device_ref)`
`:fixed_ips`	`network_port.primary_ip_address`
`:cloud_subnet_network_ports` — see the Code 3.4	

3.7.5 Network Routers

The only remaining network component to import is a *Network Router*. Routers are used to manage *Cloud Subnets* and each router can handle more than one subnet. New routers are created only on purpose when a user requests a new subnet, but he does not list it under any present router. Despite the API does not provide much information for this entity, they are a necessary link between other elements in the network.

Table 3.9: *Network Router* attributes mapping

Attribute	Corresponding value
ID	`network_router["id"].to_s`
`:type`	`"ManageIQ::Providers::Softlayer::NetworkManager::NetworkRouter"`
`:ems_ref`	`network_router["id"].to_s`
`:name`	`network_router["hostname"]`

3.8 Register a provider

Properly registering the implemented provider is required. Otherwise the **ManageIQ**'s sub-systems wouldn't recognize any of the added functionality. There are two important steps to properly entitling and enabling the provider.

1. Register the managers for displaying, monitoring and refreshing purposes
2. Enable to provision via this provider

The first step is done by enlisting the provider in the main controllers for the **ManageIQ** workers. The files, where to mention the brand new *Cloud Manager* and *Network Manager* are:

- Register a new vendor type:
  `app/models/vm_or_template.rb`
- Define the setup and teardown order for the provider:
  `app/models/miq_server/worker_management/monitor/class_names.rb`

Then, it is necessary to enable the provision automation engine. Without this, the provider would be still functional for monitoring purposes but the user would not have any option to provision a *Virtual Server* there. Desired behavior can be achieved by adding and registering the vendor here:

- Register the vendor as eligible for provisioning for front-end:
  `app/models/manageiq/providers/cloud_manager/template.rb`
- Do the same for the back-end:
  `app/models/miq_provision.rb`
- Bind the managed properties with provisioning automation engine in folder:
  `lib/miq_automation_engine/service_models/`

3.9 Front-end

Now, once the provider is properly listed in the back-end of the **ManageIQ** system, it necessary to create and enable the user to access the new functionality. Various templates have to be adjusted, some others newly created. These are the main ones modified and created for the **IBM SoftLayer**:

- Enable the form for adding a provider in:
  `app/views/shared/views/ems_common/angular/_form.html.haml`
  `app/controllers/ems_common.rb`
  `app/assets/javascripts/controllers/ems_common/ems_common_form_controller.js`
- Register the discovery template:
  `app/views/ems_cloud/discover.html.haml`
- Define what credential fields to show in:
  `app/views/layouts/angular/_multi_auth_credentials.html.haml`
- Add provision template into folder:
  `product/dialogs/miq_dialogs/`

- Adjust the data evaluation to pass proper values to the back-end:
  `app/controllers/application_controller/ci_processing.rb`
- Add and register the provider icon:
  `app/services/network_topology_service.rb`
  `app/assets/images/svg/vendor-softlayer.svg`

3.10 Not implemented functionality

Unfortunately, there are some areas left not implemented into **ManageIQ**. The lack of this functionality is caused by lack of proper API bindings. There are none yet available. The features that API for **IBM SoftLayer** currently does not contain are:

- It does not list disk capacities for provisioned *Virtual Machines*.
- Missing *Event* driven monitoring.
- Lack of any advanced analysis available via API.

Once these features are available, they would be implemented.

Chapter 4

Conclusion

This thesis managed to describe how *Cloud Computing* and *Cloud Providers* work, how complex and diverse systems they can be, and what are the main challenges when a project like **ManageIQ** tries to integrate one of them. The main focus was on the process of integration of a new provider, the **IBM SoftLayer**. This provider is a cloud vendor of the *IaaS* type and offers *Virtual Machines* and *Bare metal Servers*.

The process of integration was described by using the main **ManageIQ** bindings. They were elaborated on in detail and used with connection of the *Fog* Ruby cloud library. Each of the parts needed was described and the mapping of the provider's namespace into the ManageIQ was presented.

Currently, the result code is under a review process [1] by upstream community. Eventually, the code will be merged into the upstream and will stand as a standard way for the communication between **ManageIQ** and **IBM SoftLayer**. To see this solution in work, do not hesitate to use the enclosed appliance and deploy it. Or you can watch a demo demonstrating the functionality as well.

[1] https://github.com/ManageIQ/manageiq/pull/8324

Bibliography

[1] Amazon Web Services, Inc.: What is Cloud Computing? [online]. 2016. [accessed 28 November, 2015].
Retrieved from: http://aws.amazon.com/what-is-cloud-computing/

[2] Cervone, H.F: An overview of virtual and cloud computing. *OCLC Systems & Services: International digital library perspectives*. vol. 26, no. 3. 2010: pp. 162–165.

[3] Google, Inc.: Google Compute Engine: Regions and Zones. [online]. 2016. [accessed 20 April, 2016].
Retrieved from:
https://cloud.google.com/compute/docs/regions-zones/regions-zones

[4] Hassan, Q.F: Demystifying Cloud Computing. *The Journal of Defense Software Engineering*. vol. 24, no. 1. 2011: pp. 16–21.

[5] Jansen, G.: Managing heterogeneous environments with ManageIQ. [online]. 16 March, 2016. [accessed 29 April, 2016].
Retrieved from: https://lwn.net/Articles/680060/

[6] Microsoft Corporation: Azure Regions. [online]. 2016. [accessed 21 April, 2016].
Retrieved from: https://azure.microsoft.com/en-us/regions/

[7] Shuijing, H: Data Security: the Challenges of Cloud Computing. *Sixth International Conference on Measuring Technology and Mechatronics Automation*. 2014: pp. 203–206.

[8] SoftLayer Technologies, Inc.: Our Platform: Data Centers. [online]. 2016. [accessed 22 April, 2016].
Retrieved from: http://www.softlayer.com/data-centers

[9] SoftLayer Technologies, Inc.: SoftLayer API Overview. [online]. 2016. [accessed 28 April, 2016].
Retrieved from: http://sldn.softlayer.com/article/Softlayer-API-Overview

[10] SoftLayer Technologies, Inc.: The softlayer_api Gem. [online]. 2016. [accessed 29 April, 2016].
Retrieved from: http://softlayer.github.io/softlayer-ruby/

[11] fog-softlayer—SoftLayer module for fog. [online]. 2016. [accessed 17 February, 2016].
Retrieved from: https://github.com/fog/fog-softlayer

[12] ManageIQ Developer Documentation. [online]. 2016. [accessed 12 November, 2016].
Retrieved from: http://manageiq.org/documentation/development/

Appendices

Appendix A

CD Content

A.1 Thesis PDF

A.2 Thesis Source

A.3 ManageIQ with IBM SoftLayer support snapshot

A.4 Implementation source code

A.5 Demo video

A.6 Setup guide