BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

GENERIC CONFIGURATION INTERFACE FOR VIRTUAL MACHINES

OBECNÉ KONFIGURAČNÍ ROZHRANÍ PRO

VIRTUÁLNÍ STROJE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MARTIN KRAJŇÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. VLADIMÍR BARTÍK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

The aim of this work is to document the development process of configuration dialogs for oVirt entities. Main focus is placed on virtual machine dialog as it has plenty of dependencies related to each other. The work also describes communication with oVirt engine through REST API and experimentally ManageIQ REST API. Dialogs are created using modern Javascript frameworks React, Redux and Redux-Saga to secure proper content and state management in every possible situation. The development work done by this thesis should improve the code base, user-experience and speed up execution of basic tasks.

Abstrakt

Keywords

oVirt, Red Hat, React, Redux, Redux-saga, Javascript, Rest API, Manage IQ, PatternFly, configuration dialog.

Klíčová slova

oVirt, Red Hat, React, Redux, Redux-saga, Javascript, Rest API, Manage IQ, PatternFly, konfiguračný dialóg.

Reference

KRAJŇÁK, Martin. *Generic Configuration Interface for Virtual Machines.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Bartík Vladimír.

Generic Configuration Interface for Virtual Machines

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr Ing. Vladimír Bartík, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Martin Krajňák
May 10, 2017

Acknowledgements

I would like to thank my advisor Ing. Vladimír Bartík, Ph.D. for advices provided during writing this thesis. My thanks also belong to Ing. Tomáš Jelínek for providing technical guidance and lead with implementation and documentation of this work. I would also like to thank Mgr. Martin Beták for help with better understanding of Javascript. Finally I would like to thank my family and friends for support during my studies.

Contents

Chapter 1

Introduction

Virtualization has become very important and powerful tool used in various technology sectors. There are plenty of usecases including testing, learning and development. Availability and improvement of open source technologies is making this area even more competitive. As datacenters grow, there are several aspects that need to be considered when choosing the most suitable solution. Reasons and advantages why are businesses implementing virtualization solutions are growing[25].

oVirt[15] provides complete stack of management functions allowing to control and monitor the whole realm of virtual datacenters. The presence of rich RESTful API, even allows us to build our own custom tools such as moVirt[13] and Ansible[1].

Nowadays, the internet is being overwhelmed by modern single page applications created by advanced Javascript frameworks. This paper is written around the project, which makes effort to build similar application for oVirt. Main focus will be placed on dialogs as they administrate big entities like virtual machines and templates. Each of those entities has huge number of fields that might be in relation with one another. The challenge is to make dialogs quick, responsive and force them to always provide valid data. Regarding data validity there are two specials cases. The first case represents fact, that many of fields can be prefigured from template. The other one is a case when we need to edit particular entity, so it is crucial to display data belonging to right entity, which needs to be edited. This points to the fact, that dependency handling and excellent state management based on decision made by user can influence the data in one or few other fields.

Redux is technology designed especially for state management of React[18] applications. There are some recommendations not to use Redux[20] to manage state of dialogs. Configuration dialogs of oVirt entities can contain up to 62 fields as shown in AppendixA. Verifying data throughout whole configuration process, field after field, via technologies like jQuery can lead to pretty complex code. This is the reason why thoroughly designed solution for state management is required. But in this case, it is so complex that it is necessary to know the values in various fields to make sure that user is selecting valid data. Template belonging only to certain cluster provides good example.

React allows us to create presentional part of application. Similiarly like Redux, React itself has mechanisms to manage state of components, but as our application expands, large number of components may cause problems which lead to birth of Redux. This project is developed with open source spirit and so is the project design delivered by Patternfly[16] library of elements.

Chapter 2

oVirt

oVirt is an open source virtualization management tool that provides centralized management of virtual datacenters, hosts, virtual machines, storage and networking infrastructure. oVirt platform consists of two main parts – an oVirt engine and one or more oVirt nodes.

2.1 oVirt engine

oVirt engine is a Java application running as web service and represents the part where all management features resides. The service is communicating directly to VDSM(Virtual Desktop and Server Manager) allowing the users to deploy, start, stop, migrate and monitor virtual machines. It comes with advanced management features for virtual machine lifecycle, storage, networking and live migration. oVirt engine stores all the information about virtual machines, virtual networks and storages in PostgreSQL[17] database. User interaction with engine can be achieved via built-in web application for users and administrators. External application like ManageIQ and moVirt manage data centers via provided REST API. Overview of oVirt architecture is described in Figure 2.1.

2.1.1 Administration portal

Administration portal, also called Power user portal, is web based tool able to manage all available resources with user management. Administrator can grant and revoke user permissions and monitor data center via provided dashboards with graphs and statistics.

2.1.2 User Portal

More suited for end users is User Portal as it targets basic virtual machine management and access to virtual consoles secured by protocols SPICE[24] and VNC[28]. User has only access to virtual machines and resources which was allocated to him by administrator.

2.1.3 REST API

External applications may influence datacenter management thanks to RESTful API. As a demonstration can be used Android application moVirt, which allows to manage and monitor datacenter from a smartphone. oVirt REST API supports both XML and JSON formats and it will be crucial part of development part described in this document.

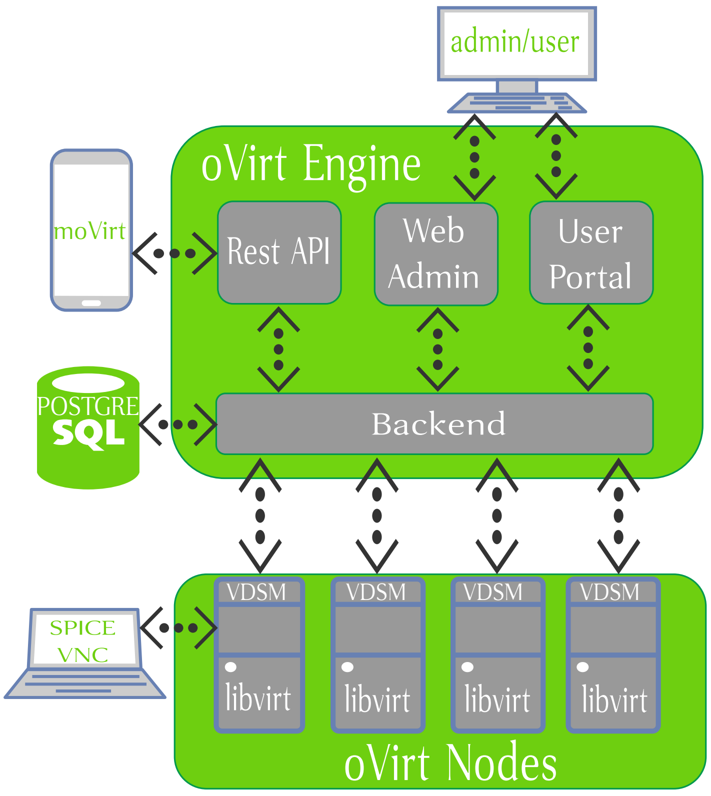


Figure 2.1: oVirt architecture [19]

2.2 oVirt node

Resources managed by oVirt engine belongs to one or more oVirt nodes, which are basically servers running RHEL, Fedora or Centos with enabled KVM[10] hypervisor and VDSM daemon. VDSM deamon is an application written in Python that has control of all available resources including storage, networking and virtual machines. VDSM-Hooks[27] allow to extend the VDSM functionality by a custom script which can be executed at certain lifecycle events of virtual machine. Management of virtual machines lifecycles and collection of statistics is possible via libvirt[11]. VDSM is also responsible for reporting all actions to engine.

2.3 oVirt Entities

Data managed by oVirt are structured to objects known as entities. Next few sections are focused on explanation of oVirt entities important for this thesis.

Cluster is logical group of hosts sharing the same storage domain and have the same CPU architecture or CPU family.

Template represents a copy of virtual machine. This functionality is very valuable especially in cases when you need to repeatedly create bigger amount of virtual machines with same of similar properties. Template also holds the information about hardware and software configurations of derived virtual machine.
There are two possibilities how virtual machines can be created from template:

1. **Thin provisioned** has an advantage that data storage of the virtual machine is just a thin copy so it saves disk resources. On the other hand there is also disadvantage in CPU capacity needed to manage disk diffs. Also once a new virtual machine is created by this method, template cannot be removed while the virtual machine exists in the environment.

2. **Clone provisioned** is case where whole disks are copied from the template, so this method requires more disc capacity. A virtual machine created this way is independent on template therefore it can be removed at any time.

Virtual Machine can be explained as actual computer system running in emulated environment and providing as much functionality as actual physical compute would provide.

Host is physical computer with installed hypervisor which allows to run multiple virtual machines on this host. oVirt usually has multiple host machines that are able to run as many virtual machines as resources allow.

Chapter 3

ManageIQ

ManageIQ is open source cloud management tool able to manage environments of different sizes, as shown in Figure 3.1. With support for platforms like oVirt, Open Stack, Kubernetes, Amazon Web Services, Google Cloud Platform, Microsoft Azure and many more allows user to control multiple technologies such as virtual machines, public clouds and containers from multiple vendors in single web application. Application itself is written in Ruby and it can be deployed as virtual machine image and Docker container.

From oVirt perspective ManageIQ can perform only basic tasks compared to tasks that are performable by oVirt web tools. Compared the all the functions implemented in oVirt Administration portal with A. Advantage is that user has data from every platform in one place with almost same amount of options as provided by each underlying tool. Disadvantage may be that user has to distinguish between products of various vendors which can become complicated when managing big number of entities. This project will focus on research of ManageIQ API from oVirt perspective and try to integrate it on similar layer as oVirt API. Next sections are focused on ManageIQ architecture and are based on Gert Jansens article[30].

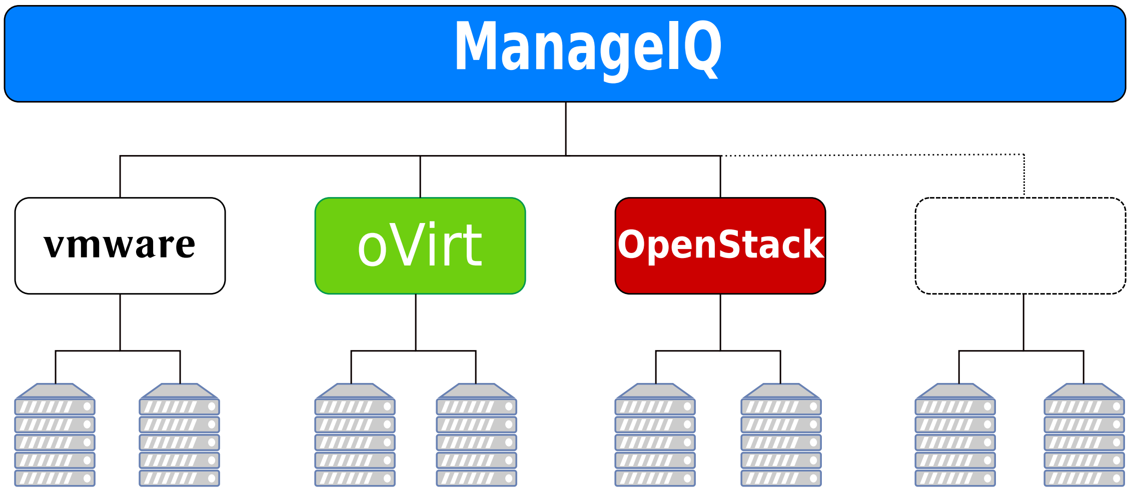


Figure 3.1: ManageIQ architecture overview[12]

3.1 Discovery

All platforms supported by ManageIQ are providing APIs. By integrating these API functions, ManageIQ can scan the environment and discover all virtual machines, hypervisors, containers, storages, networks and all the others resources. Discovered data of entities and its relations are stored in the Virtual Management Database(VMDB).

After initial setup ManageIQ listens to events that are indicating changes and use them to refresh the VMDB. This way ManageIQ VMDB has always almost up to date data. It also features an option to make a full re-scan, which is also scheduled every 24-hours. Data are presented to user via web interface. For oVirt instance displayed content are list of clusters, templates, virtual machines and all related attributes.

3.2 Operational management

Since API of various platforms allow us to control some of entities actions. Not all of the actions are covered, the goals is to be able to do main management features through ManageIQ. In case of oVirt entities user is able to create, edit virtual, clone and migrate virtual machines also perform basic tasks like power on, power off and reboot.

ManageIQ tracks the changes and can display reports about changes made to entities over time. It tracks attributes like discs, memory but in some cases it can track even software versions. Attribute changes can be compared to entities of same type or to entity itself from earlier time.

Resource management and monitoring is another advantage. ManageIQ provides various utilization charts of metrics like CPU, memory, disk with prediction when will these resources runs out of capacity.

ManageIQ can help in financial area. User can assign certain cost values to resources like Virtual machine memory and disk, so ManageIQ can provide report with costs of whole system or of certain group of users.

3.3 Self-service

This feature allows the administrator to create catalog of request that can be ordered by users. It saves a lot of time for an administrators and also for a user as the virtual machine or application are delivered to them faster. The administrator can create collection of service items represented as service bundle. Each item represent an entity which ManageIQ knows how to create for example a virtual machine or container.

Some services require amount of input from user like memory and disk size in case of virtual machine. For this purpose the administrator can create a dialog via integrated dialog editor. Once the service bundle and dialogs are created, the service bundle needs to be associated with with an entry point which defines how this resource(virtual machine or container) will be provided. After completion of this process the service bundle can be inserted in the service catalog where it can be ordered by user. Once service is deployed user can start and stop virtual machine and has access to console. Services also have lifetime which can be set by administrator and service can be automatically terminated upon lifetime expiration. User is notified about expiration via email and might have an option to extend service.

3.4 Compliance

With ManageIQ administrators also have a tool for enforcing policies to discovered entities. When user deploy his own system via self-service administrator has at least some amount of control given back.

But ManageIQ give the administrators even bigger power with SmartState Analyses (SSA) technology. It allows to define rules for content of virtual machines, hypervisors and containers. SSA is able to discover configurations, logs and even package databases and store them directly to VMDB. SSA is implemented agent-less, it access the disks of systems via platform-specific APIs, usually snapshots or backup APIs. Disks cannot be safely mounted by Linux kernel, so ManageIQ implements its own Ruby-based read-only file system that access disks from user space. The agent-less implementation provide a big advantage that guests are not required to be cooperative so SSA works even on virtual machines which are currently shut down.

Chapter 4

Javascript Technologies

4.1 React

React is an open source Javacript library dedicated to user interfaces. Application is divided to simpler components and each one of them is managing its own state. Components are built with emphasis on re-usability. Features like component nesting and conditional rendering[4] allow us to make user interface modular and easier to maintain.

There are two types of React components[18]:

1. **Stateless components** have no state management, they usually take props data and return what will actually be rendered on a page. The best way to define them might be via ES6 arrow functions[2] but `React.Component` class with only `render()` function is also a possible solution.

2. **Stateful components** provide the full state management with an option to use component life-cycle methods. Any change of state will cause re-invoking of `render()` method and update of data presented on page. Every component of this kind should also define its initial state in a constructor that is compulsory. Life-cycle of statefull components is controlled by life-cycle functions which are called on certain events (component update) in an specific order.

Typical React work-flow is to create a stateful component containing multiple stateless components and pass them the data via `props`. A good practice is to define `PropTypes` to make sure that the correct data types are being passed to our component and even `DefaultProps` which will be used in case that value is not defined in `props`.

4.1.1 JSX

JSX[9] is an Javascript extension recommended to be used with React. The implementation looks like actual HTML with the dynamic data from React variables. JSX has series of advantages:

- faster writing of HTML templates and better understanding of what content will actually be rendered

- there is an optimization while code is being compiled to Javascript which gives it a better run-time performance

- it is type-safe, so there is significant amount of error detected during compilation

One of the limitations of JSX is a fact that some of the XML tag attributes are in namespace collision with Javascript. Therefore in actual JSX code attributes like `class` and `for` are being replaced with `className` and `htmlFor`, respectively. The modern web browsers are able to warn programmer when this kind of mistake is made with warning message in developer tools console.

4.2 Redux

In the world of single page web applications requirements to manage state have become increasingly complicated. As application gains more complexity, more user interface elements and complicated API calls we can easily end up in a loop of events which source may be very hard to find. Of course there will be effort to make it right but it results to even more conditional event handling, thus created flaws are harder to reveal.

Redux is represented as a read-only tree of states called store. Every piece of data in store are describing the current state of application. The only way to change the state is to dispatch an action. Actions are predefined pure functions, therefore we can easily predict an actual change of state just from knowing dispatched action.

Actions are processed by pure functions called reducers. Reducer takes the current state and the action and returns a new state without mutation of the previous state. Because reducers are only functions, we are able to achieve specific state by dispatching right actions in right order. To conclude, Redux is based on three principles[26]:

1. Single store of truth – the whole application state is stored within single tree
2. Store is read-only – the only way to make a change is to dispatch an object describing the change (action)
3. Changes are made by pure functions – reducers

4.3 Redux-saga

Redux-saga[22] is a library providing functions for React/Redux applications which makes asynchronous actions like fetching data from external resources easier and better. Saga acts like separate thread which is responsible purely for side effects. Redux-Saga is a Redux middleware, so the thread can be started, paused and canceled via actions dispatched from application. It has also access to the data stored in the Redux store and can dispatch actions to influence it. The implementation is much easier thanks to the ES6 generator functions which make them easier to write and read because the code looks like synchronous Javascript.

4.3.1 Saga middleware and sagas

The sagas are connected to the Redux store through the saga middleware. The middleware has to be created before the store and applied via `applymiddleware()` function. After application phase of the middleware and successful creation of the Redux store, the middleware can run sagas dynamically by invoking `middleware.run()` with saga as an argument. The middleware will go through the generator and execute all the yielded effects.

Sagas are functions which return a generator object. The saga-library provides also various effects which allows us to start other sagas. The first iteration of the middleware invoked the `next()` method. The yielded effects will be executed by the middleware according to effects API which specifies how will the middleware execute the sagas. While the effect is being executed, the generator is suspended. After receiving the result of the execution, the generator will call again the `next(result)` with the result as an argument. This process of the effects execution is repeated until the generator is terminated normally or by throwing some error. Saga can be also canceled either by effects or manually. Effect's executions which result in an error will cause invocation of `throw(error)` method of the generator. Also if the currently executed `yield` instruction is wrapped inside a `try/catch/finally` block and an error appears, the catch block will be executed followed by corresponding `finally` block.[21]

4.3.2 Effect creators

The following functions do not perform any executions and each one of them returns a plain Javascript object. The execution is performed by the middleware which examines the effect description and performs appropriate actions.

`take(pattern)` creates an effect which instructs the middleware to wait until the action with desired `pattern` is dispatched to the store. The pattern is interpreted be the following rules:

- no arguments or pattern equals to a `'*'` means that all actions are suitable and every dispatched action is matched
- in case that the pattern argument is a function, an incoming action has to be evaluated by the given function as true
- when pattern is a string, it has to match `action.type`
- in case that the pattern is an array, there are two possible cases:
  - item incoming action has to match all the predicates if the pattern is the array of functions
  - it has to match all the strings in case of string array

The middleware also comes with a way to terminate all the sagas blocked on `take` effect which can be done by dispatching a special action `END`. Exceptions are sagas that have, in that particular moment, forked tasks. These sagas have to wait for children to end their tasks before terminating themselves.

`put(action)` creates an effect which instructs the middleware to dispatch a provided action to the store. The effect is non-blocking and the saga will not receive any thrown error feedback from a reducer. On the other hand `put.resolve(action)` is a variant of the effect which will wait until a promise returned from reducer is resolved.

`call(fn, ...args)` creates an effect that instructs the middleware to call the `fn` function with provided `args`. The passed function can be both generator function or normal function. The middleware will not only run the function but it also examines the result. The result can be a promise, an iterator function or a value. All three cases have different behavior:

- iterator object (generator) – a parent generator is paused until the child generator finishes its task, the parent generator is resumed with the value returned by the child

- promise – a generator is suspended until the promise is resolved or rejected
- value – the middleware returns value back to saga so it can continue its execution

4.3.3 Saga helper functions

Saga helpers are functions built on top of the action creators described above, mainly `fork` and `take`.

`takeEvery` is a helper function that spawns a saga on each action dispatched to the store that matches certain pattern. It allows to write concurrent action that is handled as many times as action was dispatched, which means that new saga is started every time, even the previous sagas has not ended yet. There is also a race condition problem because there is no guarantee that saga will be terminated in the same order in which they have been started.

`takeLatest` also spawns a saga each time an action with a pattern is dispatched with one major difference. The previously started sagas which has not been terminated yet, are automatically canceled. So if saga is listening for an action with a certain pattern and user triggers the action multiple times, an old request is obsoleted by a new one. Of course this is only true in case that the old request has not been finished yet.

`throttle` listens to dispatched actions and spawn sagas when the action is received with one difference from the other two helper functions. After receiving the first action it will hold the execution of incoming action by certain time. Actions that have been received during given time are placed in a sliding buffer which means that only one most recent action will be kept. This is particularly useful in cases when we want to prevent our server from being flooded by requests.

4.4 Redux-devtools

As application state grows it may become pretty unclear what actions are being dispatched, when and how are they affecting the state. Using Javascript console for debugging might be confusing and even mislead us.

Redux-devtools allows programmers to go through every single action dispatched from the initial application state to the current state. Reducers are pure functions, so taking a series of actions applying them to a state will always yield the same result. The application basically becomes a movie which can be rewinded back and forth. Data in a store can be seen in every moment and after clicking on the dispatched actions programmer is provided with diff what exactly was changed. There is also an option to view application states as oriented graph, and move through its paths. As we inspect dispatched action, actual UI of application is changing too, because its state depend on the Redux store. Thanks to this we can easily see which actions triggers the changes in the UI or debug animations. In order for the Redux-devtool to properly work, a npm package needs to be added to project. The package comes with its own middleware applied to a store.

The other part of Redux-devtools is composed from a web browser extension. The installed extension is accessible via button in the extension are of the browser. The extension icon automatically glows when the middleware is detected in loaded application. Extension menu allows user to display a new window with all the data in the store which are describing its current state of the application, with attached list of dispatched actions sorted chronologically from the start of the application.

4.5 ImmutableJS

ImmutableJS[7] is library providing data structures like List, Stack, Map, OrderedMap, Set, OrderedSet and Record. Once any instances of these structures are created they will provide persistent and immutable data. The only way to make a change is to yield new updated copy. Usage of immutable data structures makes the application state predictable and assure us that changes are being made only in module we want to change them and avoid unnecessary bugs caused by mutation. In case of Redux changes of application state are handled by reducers.

4.6 PatternFly

Group of designers and open source enthusiasts have gathered together and created a set of practices for building user interfaces of enterprise web applications. Patternfly features color combinations, icons, dashboards, interactive widgets, pop-up windows, notifications, charts and many more components that can be included in modern web application. To get and initial set of icons and theming to our project we need to install a Patternfly module via npm[14] or yarn[29]. Then just choose the right component for project via Patternfly wesite[16] where is located all related HTML code. Some more complex components have its own module and they need to be installed separately. To achieve some functionality, user has to be able to look through actual code of the module. They also might require jQuery to run properly.

Chapter 5

Proposed solution

Work done by this project becomes a part of an open source project oVirt Web UI. The goal of this project is to build a new basic User Portal 2.1.2 which is now written as GWT[5] application. Current version of User Portal does not contain an option to edit or create virtual machines, these features belong only to Administration portal. One of the main goals of this project is to introduce them also to User Portal. Dialogs in Administration portal need to update its state of fields after every change made by a user. The update itself can take up to few seconds and if the user works with the application on daily basis it can be pretty frustrating and lower the productivity. The current concept, in which are dialogs developed, also prone to bugs which are caused by a phenomenon called callback hell[3]. The main reason why is this phenomenon hard to avoid is a big amount of dialog properties which are in very close relation to each other and may influence one another. From long time perspective, maintaining the code base can be very hard and may cause bugs by creating unwanted loops between callbacks.

Important fact is to realize that purpose of the application is to provide user a tool that will be able to communicate with oVirt engine as well as the information about stored entities and a way to manage them. Application will not be able to work standalone and will depend on engine data. For development process we have been granted access to an oVirt instance used by oVirt developers in Red Hat.

Our application will try to solve this problem by fetching as much data as possible right from the start with shallow updates scheduled for every minute. With all required data saved in Redux store, we can take advantage of proposed technologies and manage the state of dialogs.

5.1 Comunication Layer

This solution implements two possible backends oVirt REST API and ManageIQ REST API. Manage IQ integration of REST API has only been implemented as a proof of concept demonstrating the possibility of porting the oVirt Web UI to communicate with ManageIQ. Reasons behind this approach are few problems which disallowed us to implement the same level of functionality as we were able to implement with API provided by oVirt engine. To maintain the transparency between them, layers providing operations against API are developed as separate modules. Each module implements functions needed to fetch or alter the data.

Both APIs have similarities and operations against APIs are handled in both modules by `jQuery.ajax()`[8] call for HTTP asynchronous request. Request has to have proper header including `Authorization` and `Accept` fields.

```
$.ajax(url, {
      'type': 'GET',
      'Accept': 'application/json',
      'Authorization': 'Basic YWRtaW46c21hcnR2bQ==',
    }).then(data => Promise.resolve(data))
```

Code sample 5.1: Fetching data from ManageIQ with Basic Authentication

Recommended authentication method for both platforms is an authentication via token. There is also an option to use basic authentication method for development purposes, where password in only encoded using variant of Base64 as shown in 5.1. `Accept` field will use in our case `application/json` value because it is more suitable data format for Javacript than XML. Modules use several kinds of HTTP protocol methods:

- `GET` method to obtain list of entities or one specific resource e.g. virtual machines, templates, clusters,
- `POST` method is handling case when the user wants to create new entity or resource,
- `PUT` to update resource data,
- `DELETE` to delete data,
- `OPTION` used by ManageIQ, explained in section 5.3,

Important part of the communication modules is to convert entities obtained from API to understandable form for front-end. Virtual machine is represented like an object with all properties required for the user. More entities are inserted in a list.

5.2 oVirt API

Every oVirt instance offers RESP API as another way to manage virtual data centers. An entry point to API is url `https://<hostaddress>/api` where we can access every available entity, url `https://<hostaddress>/api/vms` is where we can find list of all virtual machines.

Every entity has an unique ID, this ID can be used to access or edit one particular resource identified by the given ID. Internally we also have to pay attention to the type of entity because ID itself is not enough to address a resource. We have to classify the resource before we wake a request. Example for particular request would be the request to edit virtual machine, in which we have to send altered virtual machine data via `HTTP PUT` method to url `https://<hostaddress>/api/vms/<vmid>`.

Since there are policies and restrictions for certain data, oVirt engine may deny our request and answer it with an error message. We will be taking advantage of these messages, mainly because they are very descriptive so we immediately know what is wrong with our data. If a user typed for example a space inside virtual machine name, the error message would include description about restriction regarding virtual machine's names.

5.3 ManageIQ API

As described in Chaper 3, ManageIQ is quite different project which take an advantage of oVirt API in order to manage oVirt data centers. Because each one of the management tools has its own API, ManageIQ converting data to its own, unified representation. The resources provided by ManageIQ should contain all the data retrieved from underlying management tools. Main entry point of API can be accessed via url `https://<hostaddress>/api`. It contains JSON with basic information and all accessible entities and url.

5.4 Manage IQ implementation

This section is dedicated to the problems I encountered when I tried to implement ManageIQ support to the application. It also includes communication with ManageIQ developers and applying patches which were experimental and at that time, unapproved.

The same-origin policy[23] is an important security concept implemented in web browsers. Basically, it disallows the document or script to use resource that comes from another origin. Website has a different origin if it comes with different protocol, url or port. The intention of policy is to prevent a malicious website from reading the confidential information from other websites, it also prevents the application to read the data that might be offered by other website. A banking application with sensitive data is an good example demonstrating the purpose of this policy. Security is very important, but our application requires the data from ManageIQ API that are definitely not coming from the same origin as our application.

Thankfully HTTP protocol implements The Cross-Origin Resource Sharing(CORS)[6] mechanism to alleviate the same-origin policy, so Javascript is able to read REST API served from different origin. The CORS mechanism should be fully automatic, there is no need to alter the request headers. Any request from client which desires the cross-origin communication via `GET`, `POST`, `HEAD` method automatically includes on `Origin` field in header that describes the origin of a client. The server will evaluate the clients request and will either allow it or disallow it. If the first case is true, then the server will respond with requested resources and also include `Access-Control-Allow-Origin` field in response header. This does not mean that we can access the data, there is still a second part of evaluation which is made by the browser. The `Access-Control-Allow-Origin` has to be to present in the response header and must match with the request's `Origin` field. If those two do not match, browser will disallow our application to read the data. The `Access-Control-Allow-Origin` might contain * as a value, which means that everyone is able to access given resource but it is considered a bad practice.

There is one more complication to mechanism described above. If we make a request that is not considered simple, the web browser will make a preflight request. This request will basically ask if the application is allowed to access given resource, without actually performing it. Actual request is sent after the preflight has succeeded. Request is considered simple, if the client is making any of `GET`, `POST`, `HEAD` methods and content type is one of the following `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`. Also the fields in header are limited to `Accept`, `Accept-Language`, `Content-Language`. Because we need `Content-type` to be set to `application/json` and `Authorization` our requests are not considered simple and for every one of them requires a individual preflight request. The preflights are messages which use `OPTION` method, the server response contains list of allowed actions.

The routine described above should not cause a problem, but the server part of the routine was not implemented properly by ManageIQ. The response headers sent by the server did not contain `Access-Control-Allow-Origin` field, therefore any time a request was made by the client, the server did send an answer, however it was blocked by the web browser. This problem was reported as an issue[1] to ManageIQ Github page and the patch solving the issue was delivered. After patch application problem was solved only partially because preflight request were working only for the top level entities. So we were able to fetch the list of virtual machines but not the virtual machine information.

This problem can also be solved on client side but it is considered a bad practice that should be used only in development. The solution required to completely disable web browser security, thus no more preflight requests and no more same-origin policy enforcement. Since I believe that this problem will be solved by ManageIQ team in the future as a temporary solution I decided to disable it on Chromium browser by running it with`-disable-web-security -user-data-dir` parameters. Afterwards I was able to access all the entities.

5.4.1 oVirt entities

After solving initial problem everything was ready to fully integrate the application with ManageIQ. Initial research has shown that ManageIQ REST API provides only very small fraction of the information compared to oVirt REST API. Basic task like creating a virtual machine requires at least the information about clusters, templates and optionally about virtual machines. Data of mentioned entities are accessible, however a list of operating systems is missing.

When it comes to individual entities, they include the references to the original url from which it was given resource obtained. Most of the field contains url to oVirt API, some of the fields are even duplicated but they had different names. This leads us to the conclusion that these fields might have some internal purpose but for us are unusable.

The only useful information belonging to cluster are name and id. Template consists only from id, name and cluster. It lacks important information about memory, cpu operating system and many more. In case of virtual machine id and name are consistent, however fields like template and memory differs from machine to machine. The most interesting was cluster field which holds only the url of original oVirt API resource. This pointed to the fact that it was not possible to determine which virtual machine belongs to the specific cluster. Also some of the very important resources like operating systems and important CPU and memory fields were completely missing.

There were also several unsuccessful attempts to create a virtual machine. From our point of view it looked like `POST` requests are handled be ManageIQ in the same way as `GET` requests. The received responses supported this claim because the response code was always `400 OK` with list of resources instead of `401 created`. In the end from all the available actions we were able to perform only start and stop of the virtual machine.

Performance and speed is also not quite sufficient compared to oVirt API. The way virtual machines are arranged looks inappropriate. Lets say we want to download a list of all vms and all data for oVirt entities. Consider that ManageIQ also contains big number of virtual machine which belongs to different vendors. First `GET` to `/api/vms` will get us only the list of virtual machines with url and id of every single machine. So to get data of each virtual machine we need to make `GET` requests repeatedly for every machine with included

[1]https://github.com/ManageIQ/manageiq/pull/14368

preflight request. Only after downloading the virtual machine contents we can examine the vendor field and determine if machine belongs to oVirt. If not, we made useless request and we must drop the data and proceed to the next machine. Tests on our local ManageIQ instance with around 100 virtual machines shows that only fetching virtual machines lasts up to 10 seconds. Compared to oVirt API, Manage IQ is much slower considering that it contains less data than oVirt API.

Result of this research shows that the oVirt segment of ManageIQ REST API is not yet ready for integration. The small part that was implemented and tested is not ready for user, because it misses key actions and data. We also have to mention the security problem with wrong header for OPTION method.

5.5 Saga middleware

The interconnection between proposed API modules, Redux store and React components is made by Redux-Saga middleware. Right after the start of the application the created `sagaMiddleware` is applied to the store which means that from now on, it can properly run our sagas. The sagas are generator functions which allow us to make the code look more synchronous and prevent from creation of callback hell.

All created sagas are focused in a separate module `sagas.js`. Since we need to use more than one saga and running multiple sagas by the middleware might be quite unefficient, we will make one main `rootSaga`. The `rootSaga` is also a generator function which yields a list of saga helper functions `takeEvery` and `takeLatest`. Those helper functions map every created saga to a specific action which allow us to spawn one of the predefined sagas by dispatching an action. The behavior of spawned saga also depends on kind of provided helper function as described in 4.3.1. Dispatching an action is not the only way to spawn a saga, but I prefer it this way because of easier debugging through the Redux-devtools. The difference between generator functions and classic callbacks is that generators remain paused until the effect is resolved which is the main advantage for preventing the callback hell.

With all our generators prepared we can continue with the execution of the application. Assuming that we are using the development mode and we are logged in, the application will start to fetch the data from the oVirt engine. The process begins by calling `login` generator which will go step by step through series of generators fetching lists of all clusters, templates and virtual machines. Each one of mentioned generators are using local API instance that provides methods for fetching the data and their conversion to internal representation. Upon successful data retrieval and conversion, the data is dispatched to the store via specific actions. Those actions are processed by the corresponding reducers and saved to the Redux store. Part of this workflow is displayed on Figure 5.1.

5.6 React components

The work described to this point of the thesis is from user perspective practically invisible. The part that must handle user interaction will be implemented in React with state management interconnected to Redux.

Dialogs, which are we building have up to 62 fields representing a virtual machine and its properties. Since oVirt has been adding new features on relatively regular basis we need to provide easy way to include new fields when required. The key to keep dialog modular

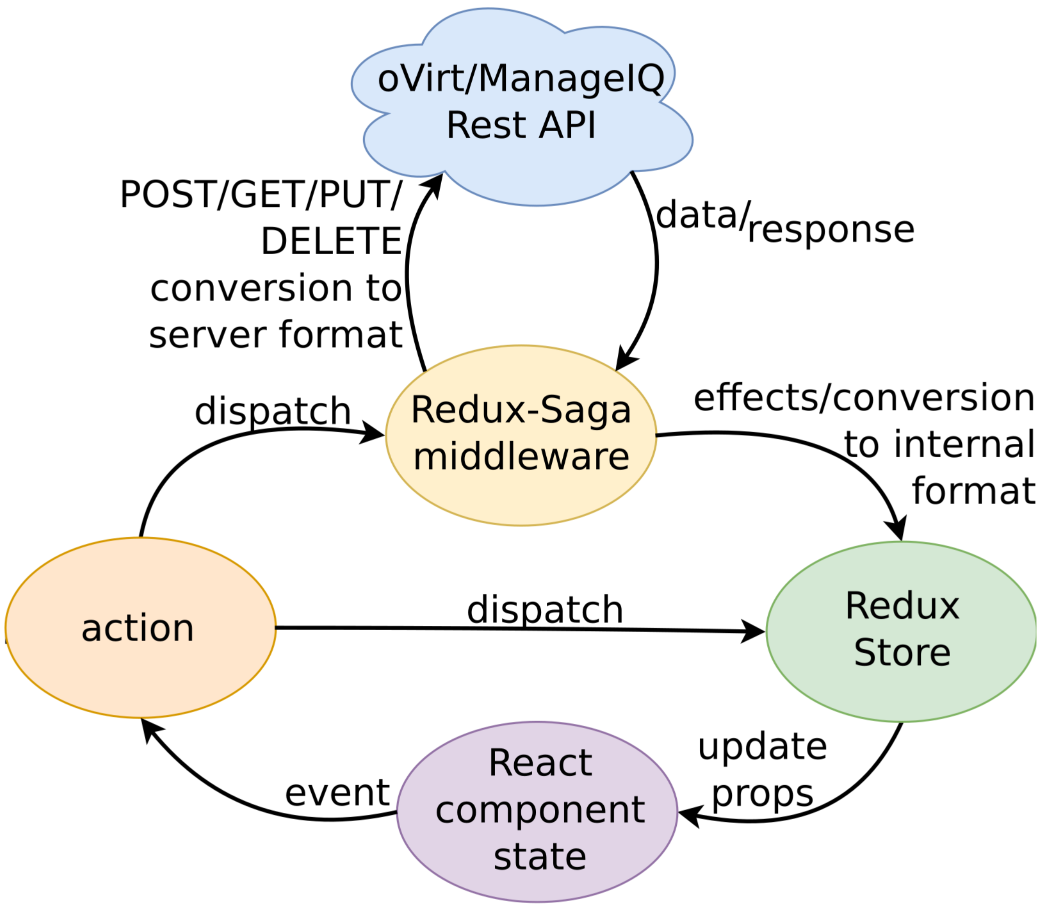


Figure 5.1: Application architecture

is to design sub-components which will allow us to easily add or remove fields. To achieve this goal we build standalone React components for every type of data we have to handle inside the dialog.

5.6.1 Stateless components

`LabeledTextField` is a component designed to obtain text input from a user. Input from user in this case may be represented either as a number or as text.

The variant dedicated to the numerical values can be initiated from a parent component by setting prop `type` to number. As shown in Figure 5.2, the field has arrows which allows user to move numerical value by a certain step. The default step is 1 but I have included dedicated prop `step` that allow us to configure this value. The minimal value achievable by clicking on arrows can also be restricted by prop `min`. Those two props are especially useful in case of virtual machine's memory where we can make sure that the value will not be negative and we can set step for example to 256MB which is more convenient increment value for memory. The arrow buttons might also act as a hint pointing to a fact that the field requires a numerical value, especially when filling less known fields.

`LabeledSelect` component provides the user with a restricted list of values from which he selects only one value. In this component I have included two widgets with different features and look. The type of widget can be configured via `selectClass` prop.

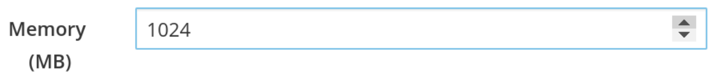


Figure 5.2: LabeledTextField number variant

The first – classic widget provides the user with list of the options with default option pre-selected. This representation is suitable for cases when the list of options is short.

The other widget gives us different look and adds the type-ahead functionality (Figure 5.3). User can simply start to type desired option and if it is available he can select it. The biggest advantage of the type-ahead functionality is maily in bigger lists of data where the values may be found much quicker compared to classic list. On the other hand there is a little disadvantage from the user perspective. Considering that user is able to type any value, I have implemented a verification process which checks if the value is really listed as an option. If not, user is notified via notification bubble with warning message specifying which field is wrong even before the data has been sent.

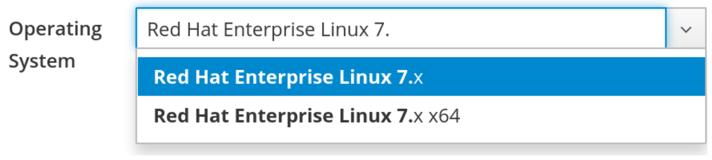
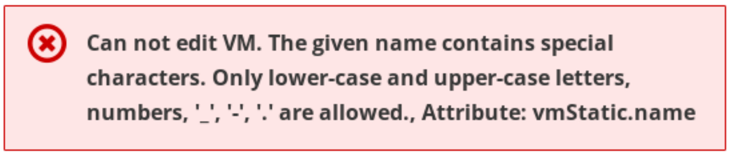

Operating System
Red Hat Enterprise Linux 7.
Red Hat Enterprise Linux 7.x
Red Hat Enterprise Linux 7.x x64

Figure 5.3: LabeledSelect example with demonstration of type-ahead functionality

`Alert` can display red rectangle window with an error message inside. This component provides a transparent and simple demonstration of conditional rendering in React. The only prop passed from parent is the message itself. If the message contain an empty string, which is also initial state, `Alert` is rendered only as empty `<div\>` without content or style. The error messages are in this case often result of failed REST API call. The errors are being detected by saga-middleware by examining the code from the server response. As soon as error is detected, saga will dispatch an action with an update which contains the error message from server. The message is processed by the corresponding reducer and store is updated. The update is broadcasted to React components which subscribed to updated part of store. In other words error message is displayed to a user immediately after the server response is received without any delays as shown in Figure 5.4.



Can not edit VM. The given name contains special characters. Only lower-case and upper-case letters, numbers, '_', '-', '.' are allowed., Attribute: vmStatic.name

Figure 5.4: Error message from API displayed by error

`LabeledSwitch` is an component dedicated to representation of boolean values. Like the previous components, it consists of two parts. A modern looking switch button (Figure 5.5) which clearly describes two of possible states on or off and a label describing the feature which it configures in dialog. The component implementation is not adjusted to React yet, so we had to use the jQuery in the same manner like we used in case of the `LabeledSelect` component. Similarly we had to use `componentDidMount` life-cycle method to initialize the component via custom method `bootstrapSwitch()`. Also the `onChange` callback within the React component cannot be used for triggering updates to the Redux store. The solution for correct detection and distribution of the changes made by user was the custom callback via jQuery triggered by `switchChange.bootstrapSwitch` event. But compared to the `LabeledText` the behavior was slightly different because we were not able to access proper state of the switch button via `ref` prop like we did in case of the other components. The value provided by the `ref` was `true` even though the switch was turned off. To solve this particular problem I had to examine the code of the widget itself and I discovered that the state is passed as an argument for every triggered `switchChange.bootstrapSwitch` event. This discovery allowed us to properly change the implemented callback which now can dispatch the state of the component right to the store.

Figure 5.5: LabaledSwitch for Smart card option

5.6.2 Stateful components

`AddVmDialog` is a React component that implements our dialog representing a virtual machine and its configurable properties. The component can be used either to create or to edit a virtual machine. The dialog variant is determined by a value from the Redux store. The create virtual machine variant renders most of the fields empty, except special cases like cluster field where the pre-selected cluster is always the first cluster provided by the connected oVirt engine. Other fields like template, operating system, memory and cpu number are taken from the Blank template which is the default template for every cluster. The second variant is pre-filled with values from the virtual machine which is being edited.

This dialog component is where we take an advantage of all the previously mentioned reusable sub-components. The difference between the `AddVmDialog` and previously described components is the fact that component is stateful whereas the others were stateless. This means that the component is created as a class which inherits from `React.Component`. Usage of a stateful variant is allowing us to use the life-cycle methods for proper sub-component initialization via jQuery. The component also needs to read the data from the sub-components which is achievable through `ref` attributes supported only by the stateful components.

The sub-components are placed in `AddVmDialog`'s `render()` method with all required props initiated for each one of them. Since components are reusable they have multiple options configurable via props. Therefore not all the props are required to be used in a parent component, only those which suit our current need. The props also come with the certain restrictions. Each of the sub-components defines its own `.propType` section with data type restriction and definition, which is describing whether the prop is or is not required. Optional props may acquire default values, e.g. `placeholder` prop is by default set to the same value as the `label` prop as long as it is not overwritten from parent.

The communication between the parent component is achieved through callbacks passed to sub-components via certain attributes (props). Those attributes are `ref` and `onChange`. The `ref` callback is a pure function which store the input in dialog field to a class property of the parent component. As an suitable example we can use cluster field where callback function is defined by following statement: `(input) => {this.cluster = input}`. Therefore the parent component is able to access the value of the cluster field by calling `this.cluster.value`. The `onChange` callback defines a function which will be called when change event occurs. In conclusion, these callbacks allow us to catch the change event in dialog field, read the data provided by `ref` and pass the obtained value to the Redux store.

The sub-component's props are not only the callbacks. The props are also used to carry values which are supposed to be rendered by a sub-component. This apply primarily to default values, placeholders and labels. The dialog is also using the props to configure the sub-components.

The special case is `LabeledSelect` where the list of values rendered to user is passed via prop. To properly describe the problem I will be using the example which is implemented in the project. The dialog has a field which represents a list of templates. Each template depends on certain cluster (template Blank is an exception). In our dialog we know exactly which cluster is selected because we have this information stored in Redux store. Since we want to provide a user with valid data, it is necessary to render only the list of templates which applies to the selected cluster. This is achieved by a pure function which compares the ID of selected cluster with cluster ID stored in every loaded template. Afterwards, the filtered list of templates is sorted and sent to the `LabeledSelect` component.

The dialog always renders the data stored in Redux store in section `addVmDialog`. This is achieved by mapping the props of our dialog to a redux store state inside a `connect` method provided by react-redux library.

`editTemplateDialog` is a second stateful React entity implemented in this project. The purpose of the dialog is to choose from the list of the templates provided by oVirt engine and to edit properties of single selected template. The implementation of this dialog has less fields but serves as a demonstration of transparency provided by designed React components. The added value is in another additional functionality brought to the oVirt-web-ui project.

5.7 Patternfly integration and design

All React components and HTML related content are using styles provided by Patternfly. The library itself is accessible via dedicated npm module which needs to be included to gain access to styles and widgets. Not all widgets are included by default, the more complicated widgets that are implemented in Javascript and require jQuery have its own npm module which need to be included separately. In our project we have included few additional modules. A `bootstrap-combobox` module which implements the `LabeledSelect` type-ahead functionality and a `bootstrap-select` module used by simpler variant of the `LabeledSelect`. The last included module is `bootstrap-switch` and it provides the foundation for the switch button used in the `LabeledSwitch` component.

The combination of React and jQuery is the source of multiple implementation problems in front-end. As user is going through dialogs fields, they have to reflect choices he has already made. List of options in `LabeledSelect` component has to be properly re-rendered e.g. when needed. In order to re-render component which is using jQuery we have to use of the React life-cycle methods, to be specific `componentDidMount()` and `componentDidUpdate()` method. Every component using jQuery has a specific method which need to be called initially for component to properly render and work. These methods must be called from mentioned life-cycle. Thanks to this whenever user make a choice, Redux store will be update. If this change apply to part of store mapped to React component that is using jQuery, `componentDidUpdate()` is called and widget has a new list of values. Determine the method which initiate the jQuery work-flow in case of type-ahead variant of `LabeledSelect` wasn't simple. But we took advantage of a fact that Patternfly is an open source project, so we had to go through the code where we were able to find the right `combobox()` method, which initializes the component properly.

Styles for simple widgets like buttons or headings are added by providing HTML element with attribute `class` or `className` in case of JSX. List of classes with demonstrations can be found on Patternfly website[16].

5.8 Future development

Chapter 6

Conclusion

Bibliography

[1] *Ansible module for oVirt/RHEVM.* [Online; visited 10.04.2017].
Retrieved from: http://docs.ansible.com/ansible/ovirt_module.html

[2] *Arrow functions, Mozilla developer network.* [Online; visited 17.04.2017].
Retrieved from: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

[3] *Callback hell.* [Online; visited 08.05.2017].
Retrieved from: http://callbackhell.com/

[4] *Condition rendering.* [Online; visited 17.04.2017].
Retrieved from:
https://facebook.github.io/react/docs/conditional-rendering.html

[5] *GWT overview.* [Online; visited 19.04.2017].
Retrieved from: http://www.gwtproject.org/overview.html

[6] *HTTP access control (CORS).* [Online; visited 19.04.2017] Last change April 18, 2017.
Retrieved from:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

[7] *Immutable documentation.* [Online; visited 14.04.2017].
Retrieved from: https://facebook.github.io/immutable-js/

[8] *jQuery documentation, ajax.* [Online; visited 19.04.2017].
Retrieved from: http://api.jquery.com/jquery.ajax/

[9] *JSX documentation.* [Online; visited 17.04.2017].
Retrieved from: https://facebook.github.io/react/docs/introducing-jsx.html

[10] *Kernel Virtual Machine.* [Online; visited 17.04.2017].
Retrieved from: https://www.linux-kvm.org/page/Main_Page

[11] *libvirt wiki.* [Online; visited 18.04.2017].
Retrieved from: https://wiki.libvirt.org/page/FAQ#What_is_libvirt.3F

[12] *ManageIQ architecture overview and supported platforms.* [Online; visited 17.04.2017].
Retrieved from: http://blog.octo.com/wp-content/uploads/2016/06/cmpbigpicture-1024x671.png

[13] *moVirt documentation.* [Online; visited 10.04.2017].
Retrieved from: https://www.ovirt.org/develop/projects/project-movirt/

[14] *Npm website.* [Online; visited 17.04.2017].
Retrieved from: https://www.npmjs.com/

[15] *Ovirt documentation.* [Online; visited 10.04.2017].
Retrieved from: https://www.ovirt.org/

[16] *Patternfly documentation.* [Online; visited 10.04.2017].
Retrieved from: https://www.patternfly.org/pattern-library/

[17] *PostreSQL.* [Online; visited 17.04.2017].
Retrieved from: https://www.postgresql.org/about/

[18] *ReactJS documentation.* [Online; visited 05.04.2017].
Retrieved from: https://facebook.github.io/react/

[19] *Red Hat Enterprise Virtualization Platform Overview.* [Online; visited 16.04.2017].
Last change Jun 5, 2014.
Retrieved from: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Virtualization/3.5/html-single/Administration_Guide/index.html

[20] *Redux documentation.* [Online; visited 10.04.2017].
Retrieved from: http://redux.js.org/

[21] *Redux-saga API reference.* [Online; visited 27.04.2017].
Retrieved from: https://redux-saga.js.org/docs/api/index.html

[22] *Redux-saga documentation.* [Online; visited 14.04.2017].
Retrieved from: https://redux-saga.github.io/redux-saga/index.html

[23] *Same-origin policy.* [Online; visited 19.04.2017] Last change September 28, 2016.
Retrieved from:
https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[24] *SPICE homepage.* [Online; visited 10.04.2017].
Retrieved from: https://www.spice-space.org/

[25] *Top 10 benefits of server virtualization.* [Online; visited 17.04.2017]. Last change November 2, 2011.
Retrieved from:
http://www.infoworld.com/article/2621446/server-virtualization/server-virtualization-top-10-benefits-of-server-virtualization.html

[26] *Tree principles of Redux.* [Online; visited 18.04.2017].
Retrieved from: http://redux.js.org/docs/introduction/ThreePrinciples.html

[27] *VDSM-Hooks.* [Online; visited 18.04.2017].
Retrieved from: http://www.ovirt.org/develop/developer-guide/vdsm/hooks/

[28] *VNC documentation.* [Online; visited 10.04.2017].
Retrieved from: http://www.hep.phy.cam.ac.uk/vnc_docs/howitworks.html

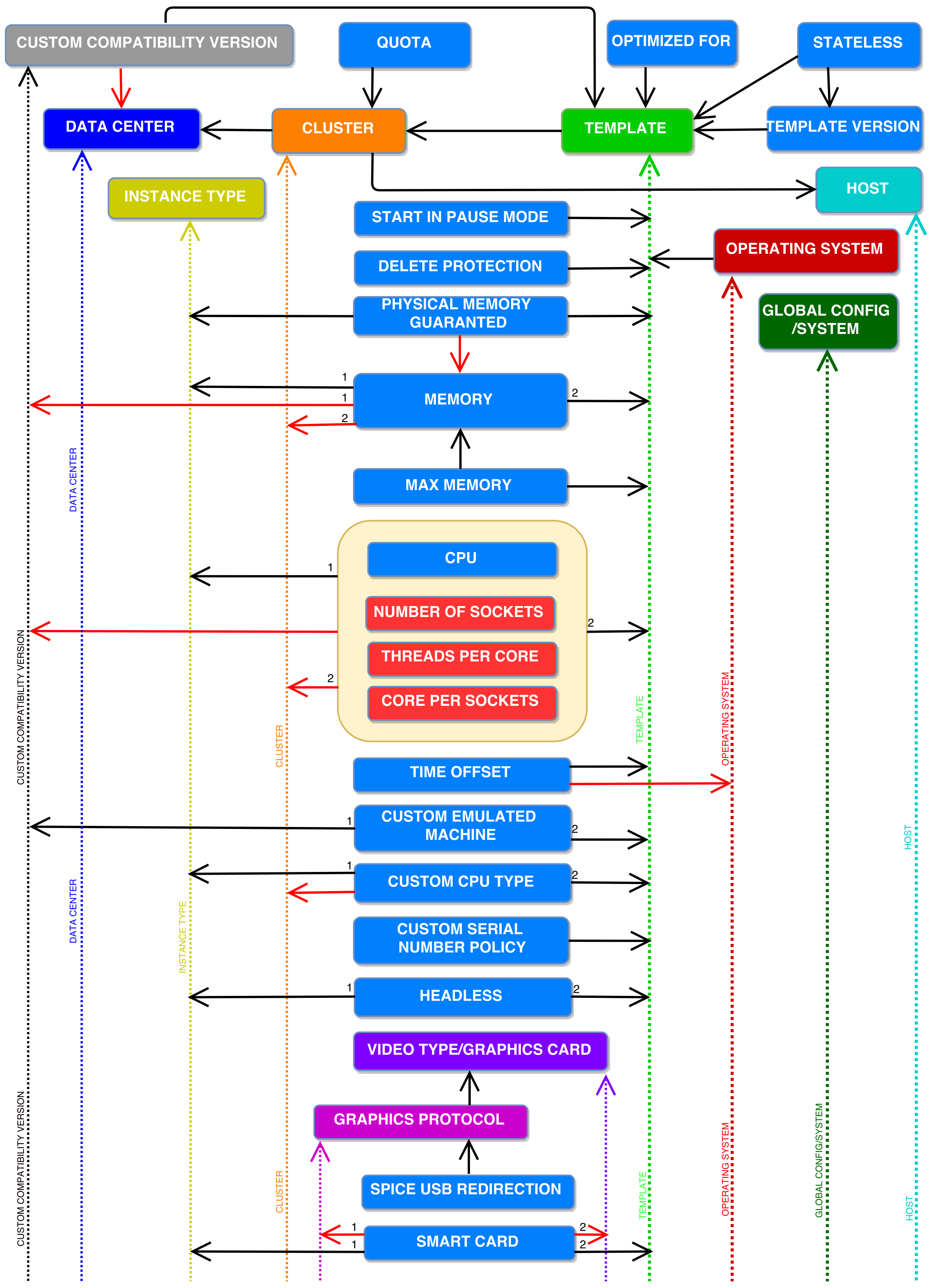
[29] *Yarn website.* [Online; visited 17.04.2017].
Retrieved from: https://yarnpkg.com/en/

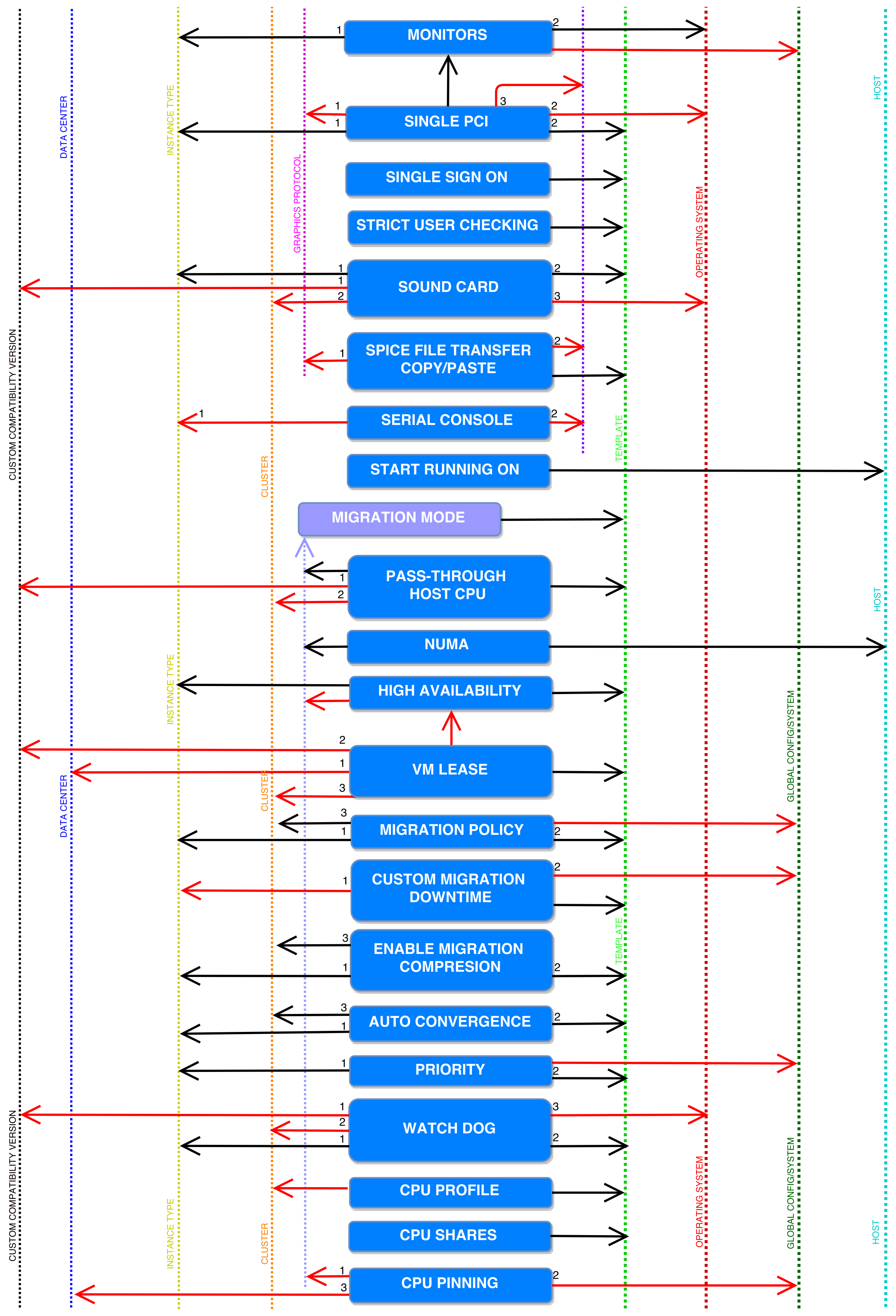
[30] Jansen, G.: *Managing heterogeneous environments with ManageIQ*. [Online; visited 13.04.2017].
Retrieved from: https://lwn.net/Articles/680060/

Appendices

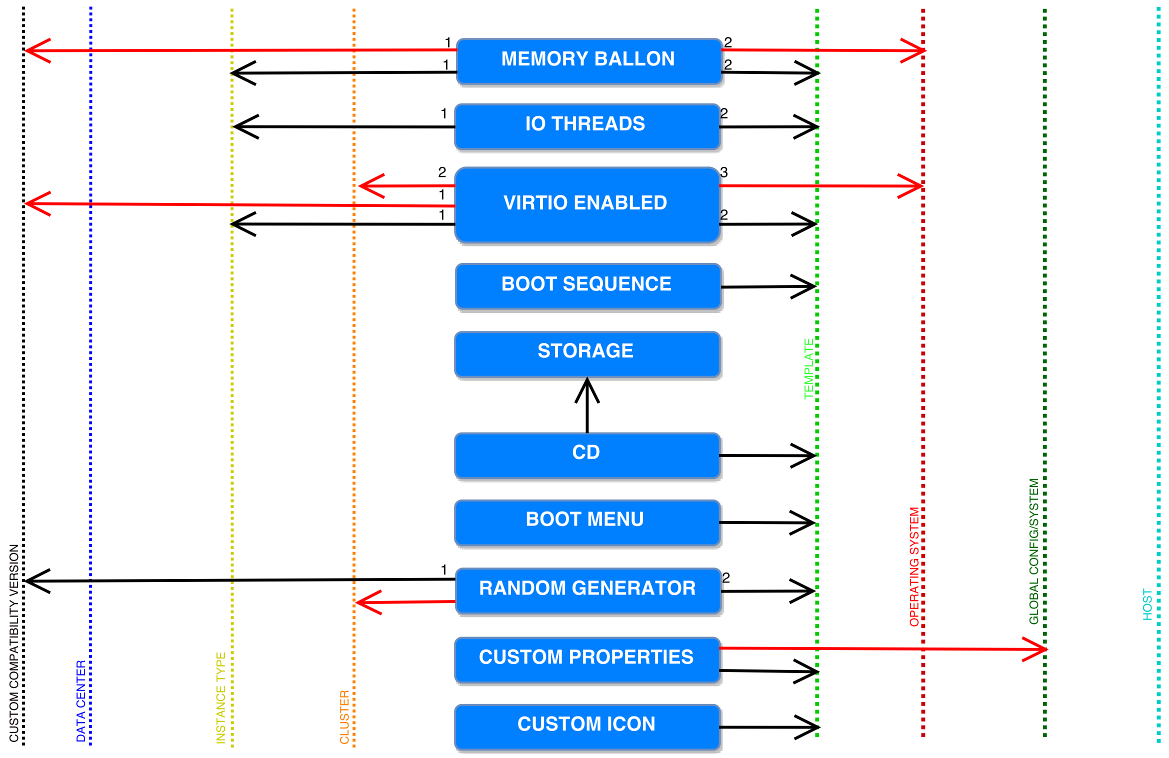
Appendix A

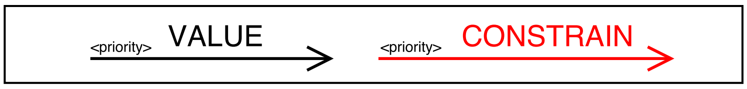
Complete oVirt virtual machine dialog dependency graph


CUSTOM COMPATIBILITY VERSION
QUOTA
OPTIMIZED FOR
STATELESS
DATA CENTER
CLUSTER
TEMPLATE
TEMPLATE VERSION
HOST
INSTANCE TYPE
START IN PAUSE MODE
DELETE PROTECTION
OPERATING SYSTEM
PHYSICAL MEMORY GUARANTED
GLOBAL CONFIG /SYSTEM
MEMORY
MAX MEMORY
CPU
NUMBER OF SOCKETS
THREADS PER CORE
CORE PER SOCKETS
TIME OFFSET
CUSTOM EMULATED MACHINE
CUSTOM CPU TYPE
CUSTOM SERIAL NUMBER POLICY
HEADLESS
VIDEO TYPE/GRAPHICS CARD
GRAPHICS PROTOCOL
SPICE USB REDIRECTION
SMART CARD



MONITORS
SINGLE PCI
SINGLE SIGN ON
STRICT USER CHECKING
SOUND CARD
SPICE FILE TRANSFER COPY/PASTE
SERIAL CONSOLE
START RUNNING ON
MIGRATION MODE
PASS-THROUGH HOST CPU
NUMA
HIGH AVAILABILITY
VM LEASE
MIGRATION POLICY
CUSTOM MIGRATION DOWNTIME
ENABLE MIGRATION COMPRESION
AUTO CONVERGENCE
PRIORITY
WATCH DOG
CPU PROFILE
CPU SHARES
CPU PINNING
CUSTOM COMPATIBILITY VERSION
DATA CENTER
INSTANCE TYPE
CLUSTER
GRAPHICS PROTOCOL
TEMPLATE
OPERATING SYSTEM
GLOBAL CONFIG/SYSTEM
HOST


MEMORY BALLON
IO THREADS
VIRTIO ENABLED
BOOT SEQUENCE
STORAGE
CD
BOOT MENU
RANDOM GENERATOR
CUSTOM PROPERTIES
CUSTOM ICON
CUSTOM COMPATIBILITY VERSION
DATA CENTER
INSTANCE TYPE
CLUSTER
TEMPLATE
OPERATING SYSTEM
GLOBAL CONFIG/SYSTEM
HOST


<priority> VALUE
<priority> CONSTRAIN

Appendix B

Installation guide

B.1 Prerequisites

Note: Work done by this thesis have partially been included in oVirt 4.1 release. To run the application in the most present version, it is recommended to use the development mode with engine url.

1. An oVirt engine running with configured hosts, disks and networks.
2. A ManageIQ instance running and configured to manage the oVirt engine (ManageIQ variant - read only mode)

B.2 oVirt API variant installation - Fedora/Red Hat Enterprise Linux

1. Copy the contents of attached CD
2. Open terminal
3. Navigate to directory of copied content
4. Make sure that you are in correct folder (oVirt API) `cd ovirt-web-ui`
5. Make sure npm is installed `yum install npm`
6. Install yarn `npm install yarn`
7. Install all project dependencies by running `yarn install`
8. Run oVirt-web-ui `ENGINE_URL=https://engine_address yarn start` (change the url to your oVirt engine)
9. Firefox window should automatically open with application, if not application url should be displayed in output of terminal window)

B.3 ManageIQ API variant installation - Fedora/Red Hat Enterprise Linux

1. Copy the contents of attached CD
2. Open terminal
3. Navigate to directory of copied content
4. Make sure that you are in correct folder (ManageIQ API) `cd manageiq`
5. Make sure npm is installed `yum install npm`
6. Install yarn `npm install yarn`
7. Install and update all required project dependencies by running `yarn install`
8. Run oVirt-web-ui `ENGINE_URL=https://engine_address yarn start` (change the url to your oVirt engine)
9. Firefox window should automatically open with application, if not application url should be displayed in output of terminal window)