



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

AUTOR PRÁCE

MARTIN KRAJŇÁK

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. VLADIMÍR BARTÍK, Ph.D.

BRNO 2017

Zadání bakalářské práce

Řešitel: **Krajňák Martin**

Obor: Informační technologie

Téma: **Obecné konfigurační rozhraní pro virtuální stroje
Generic Configuration Interface for Virtual Machines**

Kategorie: Web

Pokyny:

1. Seznamte se s problematikou virtuálních strojů a používanými backendy v této oblasti.
2. Navrhněte a vytvořte abstraktní vrstvu pro komunikaci s různými backendy (API ManageIQ, oVirt REST API).
3. Navrhněte a vytvořte mechanismus, který umožní uživatelskému rozhraní konfigurovat entity podobného typu (virtuální stroje, šablony, ...). Navrhněte také způsob reprezentance závislostí mezi komponentami virtuálního stroje.
4. Navržená řešení implementujte a vložte je do prostředí oVirt WebAdmin.
5. Implementované řešení otestujte.
6. Zhodnoťte dosažené výsledky a další možná pokračování tohoto projektu.

Literatura:

- Lesovsky, A.: Getting Started with oVirt 3.3. Packt Publishing, 2013. ISBN 9781783280070.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

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: **Bartík Vladimír, Ing., Ph.D.**, UIFS FIT VUT

Datum zadání: 1. listopadu 2016

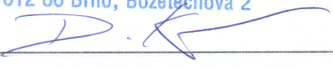
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta informačních technologií

Ústav informačních systémů

612 66 Brno, Božetěchova 2


doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

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

Cieľom tejto práce je dokumentácia procesu vývoja dialógov, slúžiacich na konfiguráciu entít oVirt-u. Práca sa sústreďuje na dialóg konfigurujúci virtuálne stroje, keďže disponujú veľkým počtom závislostí, ktoré sa vzájomne ovplyvňujú. Práca taktiež popisuje komunikáciu so systémom oVirt prebiehajúcu prostredníctvom REST API a experimentálne aj ManageIQ REST API. Dialógy sú tvorené pomocou moderných Javascriptových nástrojov React, Redux a Redux-Saga, ktoré zabezpečujú správnosť ich obsahu a správu ich stavov v každej možnej situácii. Vývojová časť tejto práce by mala zlepšiť úroveň kódu, prácu používateľa a zrýchliť vykonávanie základných úloh.

Keywords

oVirt, React, Redux, Redux-saga, Javascript, REST API, ManageIQ, PatternFly, configuration dialog.

Klíčová slova

oVirt, React, Redux, Redux-saga, Javascript, REST API, ManageIQ, PatternFly, konfiguračný dialóg.

Reference

KRAJNÁK, 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 16, 2017

Acknowledgements

I would like to thank my advisor Ing. Vladimír Bartík, Ph.D. for advices provided during writing of 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 all the support.

Contents

1	Introduction	3
2	oVirt	4
2.1	oVirt engine	4
2.1.1	Administration portal	4
2.1.2	User Portal	4
2.1.3	REST API	5
2.2	oVirt node	5
2.3	oVirt Entities	6
3	ManageIQ	7
3.1	Discovery	8
3.2	Operational management	8
3.3	Self-service	8
3.4	Compliance	9
4	Javascript Technologies	10
4.1	React	10
4.1.1	JSX	11
4.2	Redux	11
4.3	Redux-saga	11
4.3.1	Saga middleware and sagas	12
4.3.2	Effect creators	12
4.3.3	Saga helper functions	13
4.4	Redux-devtools	13
4.5	ImmutableJS	14
4.6	PatternFly	14
5	Proposed solution	15
5.1	Communication Layer	15
5.2	oVirt API	16
5.3	ManageIQ API	17
5.4	ManageIQ implementation	17
5.4.1	oVirt entities	18
5.5	Sagas	19
5.6	React components	22
5.6.1	Stateless components	22
5.6.2	Stateful components	24

5.7	Patternfly integration and design	25
5.8	Testing and verification	26
6	Alternative solution	28
7	Conclusion and future development	29
	Bibliography	31
	Appendices	34
A	Complete oVirt virtual machine dialog dependency graph	35
B	Installation guide	39
B.1	Prerequisites	39
B.2	oVirt API variant installation - Linux	39
B.3	ManageIQ API variant installation - Linux	40
C	Contents of attached CD	41

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]. The project will also try to integrate a secondary backend solution provided by ManageIQ project that should provide REST API on the similar level.

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 special cases. The first case represents fact, that many of fields can be preconfigured from templates. 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 Appendix A. 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, the problem is so complex that it is necessary to know the values in various fields to make sure that user is selecting valid data. A template belonging only to the certain cluster provides good example.

React allows us to create a presentational part of application. Similarly 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 the 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. An 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 with ability 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]. A user has only access to virtual machines and resources which was allocated to him by an administrator. Based on permissions provided by the administrator, the user has access either to Basic User Portal or Power User Portal. The basic variant does not contain any dialogs at all, only basic operations with assigned virtual machines.

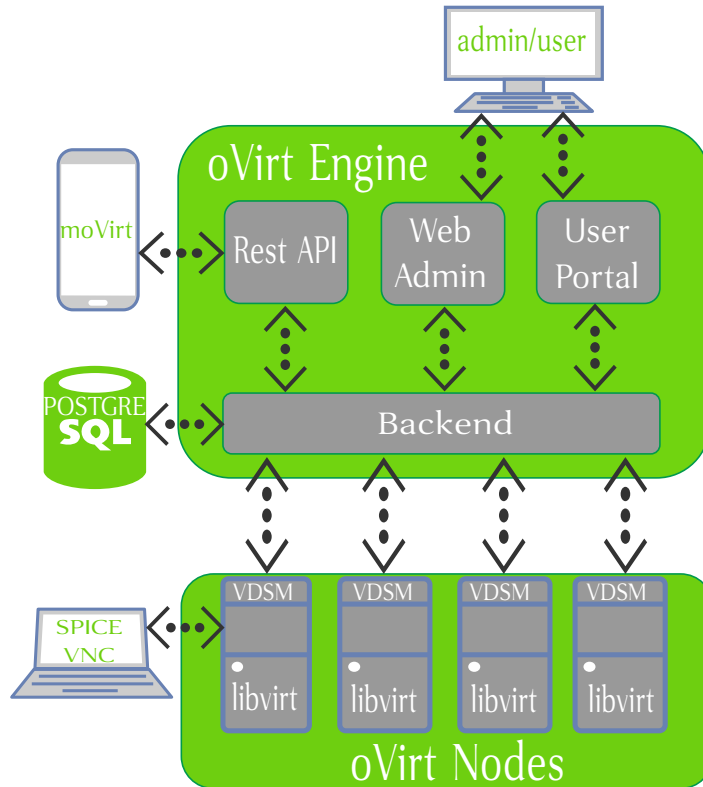


Figure 2.1: oVirt architecture [19]

2.1.3 REST API

External applications may influence datacenter management thanks to RESTful API. oVirt REST API supports both XML and JSON formats and it makes a crucial part for development of this project. One of the demonstrations of REST API utilization is an Android application moVirt that allows users to manage and monitor datacenters from a smartphone.

2.2 oVirt node

Resources managed by oVirt engine belongs to one or more oVirt nodes, which are basically servers running Red Hat Enterprise Linux (RHEL), Fedora or CentOS with enabled KVM [10] hypervisor and VDSM daemon. The VDSM daemon is an application written in Python that has control over 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 that are sharing the same storage domain and have the same CPU architecture or CPU family.

Template represents a copy of virtual machine. The presence of this entity is very valuable, especially in cases when users need to repeatedly create bigger amount of virtual machines with the same or 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 templates:

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 a 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 being copied from the template, so this method requires additional 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 an actual computer system running in an emulated environment and providing as much functionality as the actual physical computer would provide.

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

Chapter 3

ManageIQ

ManageIQ is an 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 users to control multiple technologies such as virtual machines, public clouds and containers from multiple vendors in a single web application. Application itself is written in Ruby and it provides multiple forms of deployment, including virtual machine image and docker container.

From oVirt perspective ManageIQ can perform only basic tasks compared to tasks that are performable by web tools from oVirt. The advantage is that users have data from every platform in one place with almost same amount of options as provided by each underlying tool. On the contrary, the disadvantage may be the fact that users have to distinguish between products of various vendors which can become complicated, especially when managing big amount of entities. This project will focus on a research of ManageIQ API from oVirt perspective and try to integrate it on similar level as oVirt API. Next sections are focused on ManageIQ architecture and are based on Gert Jansens article [30].

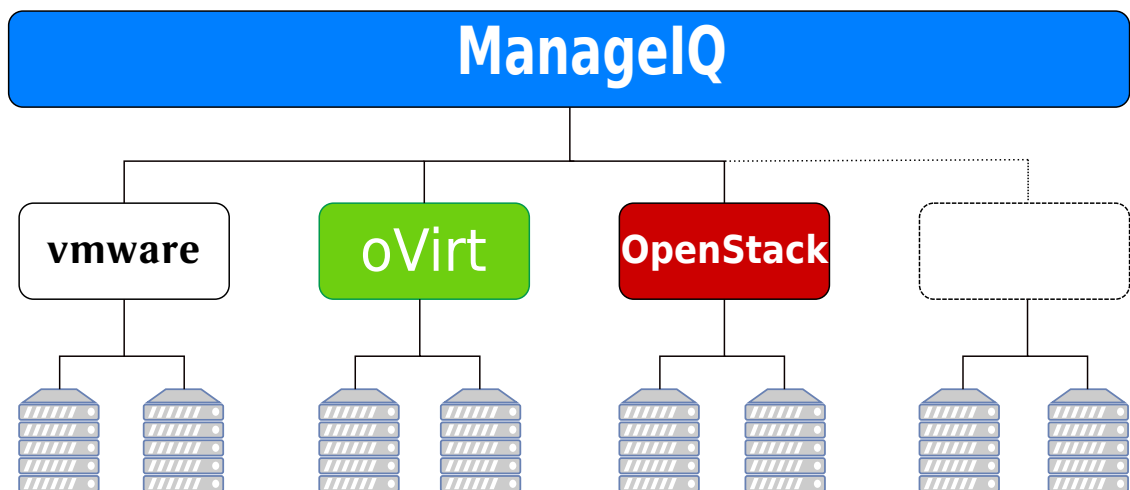


Figure 3.1: ManageIQ architecture overview [12]

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 or disk with prediction when will these resources run out of the capacity.

ManageIQ can help even in financial area. Users 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 administrators to create catalog of requests that can be ordered by users. It saves a lot of time for administrators and also for users as virtual machines and applications are delivered faster. Administrators can create a collection of service items represented as a service bundle. Each item represent an entity which ManageIQ knows how to create for example a virtual machine or container.

Some of the services may require an input from users. For this purpose, administrators can create dialogs via integrated dialog editor. Once the service bundle and dialogs are created, the service bundle needs to be associated 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 users. Once service is deployed, users can start and stop virtual machine and have an access to consoles. Every service also has a lifetime which is set by administrators. The lifetime option is important because services are automatically terminated upon their lifetime expiration. Users are notified about expiration via email and they might have an option to extend services.

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, administrators have at least some amount of control given back.

But ManageIQ give the administrators even bigger power with SmartState Analyses (SSA) technology which allows to define rules for content of virtual machines, hypervisors and containers. SSA has capabilities to discover configurations, logs, 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 big advantage of agent-less implementation resides in the fact 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 Javascript 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 receive data through props 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 `render()` function only provides 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 stateful components is controlled by life-cycle functions which are called on certain events (component update) in a specific order.

Typical React work-flow is to create a stateful component containing multiple stateless components and pass them required data via `props`. A good practice is to define `PropTypes` which provide constrain for values passed through props. React also allows to define `DefaultProps` which will be used in case that required value is not passed via `props` but they are not as required as `PropTypes`. The state of application is manage by stateful components. Any update of the state may cause that props are changed and the update is pushed to underlying stateless components, thus they are forced to re-render. Stateless components are not capable of any logic but the are able to trigger some events directly to the stateful parent. The parent can define a callback function which is passed to underlying components as a prop. Every event triggered from the stateless component will result in a call of the function passed from the parent.

4.1.1 JSX

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

- faster writing of HTML templates and better understanding of what content will actually be rendered
- code is optimized before compilation, so it has better run-time performance
- it is type-safe, so there is a significant amount of error detected during compilation

One of the possible limitations of JSX might be the fact that some of its attributes are in namespace collision with Javascript. Therefore HTML attributes like `class` and `for` must be replaced with JSX attributes `className` and `htmlFor`. Modern web browsers are able to warn the programmer when this kind of mistake is made with a warning message in the developer tools console.

4.2 Redux

In the world of single page web applications, requirements to manage state have become increasingly complicated. As an application gains more complexity, more user interface elements and complicated API calls, it might be very easy to end up in the loop of events and callbacks. An effort to correct this state may lead to even more conditional event handling, thus created flaws are even harder to reveal.

Redux is represented as a read-only tree of states called store. Every piece of data in the store is describing the current state of the application. The only way to change the state is to dispatch an action. Actions are predefined pure functions, therefore every change is predictable.

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

1. single store of truth – the whole application state is stored within a 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 a 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 the 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 acts exactly 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 to it via `applymiddleware()` function. After middleware application phase and successful creation of the Redux store, the middleware can run sagas dynamically by invoking `middleware.run()` method 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. In the first iteration, the middleware invokes the `next()` method to retrieve the next effect. The yielded effect 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 also be 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 that instructs the middleware to wait until the action with desired `pattern` is dispatched to the store. The pattern is interpreted by 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:
 - 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

`fork(fn, ...args)` behaves similarly as `call` with one difference, the operation is non-blocking. So the middleware will not wait until the `fn` is resolved and it will continue in generator execution as soon as `fn` is invoked.

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 actions handled as many times as action was dispatched. Therefore a new saga is started by each action, even though previous sagas have not ended yet. There is no guarantee that sagas will be terminated in the same order in which they have been started.

`takeLatest` also spawns a saga each time an action with a certain 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. 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 the only 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 which actions are being dispatched, when and how are they affecting the state. Using the Javascript console as a tool for debugging these actions might be confusing and even misleading.

Redux-devtools comes as a web browser extension. The installed extension is accessible via button in the extension area of the browser and the icon automatically glows when the custom middleware is detected in loaded application. The extension menu allows programmers to display a new window with the content of the store. The content is showing the current state of the application, with attached list of dispatched actions sorted chronologically from the start of the application. Programmers are allowed to go through every single

action dispatched from the initial application state to the current state. Since reducers are pure functions, applying the series of actions to the state will always yield the same result. The application basically becomes a movie which can be rewinded back and forth. Data in the store are visible in every moment from the beginning of the application. Every dispatched action also comes with a diff describing what exactly was changed by dispatching the given action. There is also an option to view application states as an oriented graph and move through its paths. Rewinding the application back and forth also allow easier examination of actions that cause UI changes.

The requirements for this tools are composed from the npm package which need to be included in the project and the mentioned browser extension available for both Chrome and Firefox. Additionally, the custom middleware has to be created and applied to the store.

4.5 ImmutableJS

ImmutableJS [7] is library providing several data structures, including `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 provide the assurance that changes are being made only in intended modules. In case of Redux changes of application state are handled by reducers where Immutable objects help to prevent mutation of the state.

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 applications. The library is available either as npm [14] or yarn [29] package. Patternfly website [16] provides a showcase of all components and related code with documentation. More complex components require installation of dedicated modules and jQuery for proper functionality.

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 version of Basic User Portal 2.1.2 which is now written as GWT [5] application. The current version of Basic User Portal does not contain an option to edit or create virtual machines, these features belong only to Administration portal and Power User Portal. One of the main goals of this project is to introduce them also to Basic User Portal. The performance speed of current dialogs implemented in current oVirt solutions is insufficient. The state of dialog fields require an update almost after every change made by a user. The update itself can take up to few seconds and the experience for users working with the application on daily basis can be frustrating and counterproductive. 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 in this implementation 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. The solution of this problem should be provided by ES6 generator functions implemented by the saga middleware.

Important fact is to realize that purpose of the application is to provide 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.

```
1 $.ajax(url, {
2   'type': 'GET',
3   'Accept': 'application/json',
4   'Authorization': 'Basic YWRtaW46c21hcnR2bQ==',
5 }).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. A secondary option allows the use of Basic authentication for development purposes, where the password is only encoded by Base64 as shown in 5.1. `Accept` field will use in our case `application/json` value because it is more suitable data format for Javascript 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,
- `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 internal form understandable for front-end. Both network operations and conversions are implemented in `ovirtapi.js` module. Virtual machine is represented like an object with data stored as its properties. Several entities of the same kind are inserted inside list. The module is implemented as the class which can be imported and instantiated in any other module.

5.2 oVirt API

Every oVirt instance offers RESP API as one of the several ways to manage virtual data centers. An entry point to API is the url `https://<hostaddress>/api` showing which data are available with their addresses. The listed url `https://<hostaddress>/api/vms` provides the list of all virtual machines. Obtaining these resources requires sending a HTTP GET request to the given url. Additionally, to make sure that we have an access to full contents of virtual machine properties, it is required to adjust the header of the request by adding the field `All-content: true`.

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 the ID itself is not enough to address a particular resource. We have

to classify the resource before we make a request. Example for a request would be the request to edit a virtual machine. The request has to contain altered data the virtual machine in JSON format. Data are delivered using the HTTP PUT method to the 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 about what is exactly wrong with our data. Messages will be forwarded to frontend and shown to the user. Messages are in some cases very specific, for example a mistake in a virtual machine's name would lead to the error message with description about restrictions regarding virtual machine naming policy.

5.3 ManageIQ API

As described in Chapter 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 retrieved data from underlying management tools. Main entry point of API can be accessed via the url `https://<hostaddress>/api`. Similarly to oVirt REST API, it contains data in JSON format with basic information about all accessible entities and their addresses.

5.4 ManageIQ implementation

This section is dedicated to the problems encountered during the implementation process of ManageIQ backend module. The work also includes communication with developers of ManageIQ project, testing and verification of patches which were experimental and unapproved at that time.

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 from reading the data that might be offered by other website. A banking application with sensitive data is a 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. Therefore Javascript is able to read REST API data served from a different origin. The CORS mechanism should be fully automatic without any need to alter the request headers. Any request from client which desires the cross-origin communication via GET, POST, HEAD method automatically includes the `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 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 the `Content-type` field to be set to `application/json` and the `Authorization` field, our requests are not considered simple. Therefore, every one of them requires a individual preflight request. Preflights are messages which use `OPTION` method and the server response contains list of allowed actions for the application which sent the given preflight.

The routine described above should not cause any problems 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 provided data were blocked by the web browser. This problem was reported as an issue¹ to ManageIQ Github page and the patch solving the issue was delivered shortly. After patch application, problem was solved only partially because preflight requests were working only for the top level entities. So we were able to fetch the list of virtual machines but we were unable to fetch the data of any particular machine.

This problem can also be solved on client side but it is considered a bad practice that should be used only during development. The solution requires 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, operating systems and optionally about virtual machines. Data of mentioned entities are accessible, unfortunately the list of operating systems is not available.

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

The only useful fields belonging to cluster entities are name and id. Templates consists only from id, name and cluster. They lack important fields about memory, cpu, operating system and many more. In case of virtual machine, ids and names are consistent, however fields like template and memory differs from machine to machine. The most interesting was

¹<https://github.com/ManageIQ/manageiq/pull/14368>

cluster field which holds only the url of the 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 by 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 urls and ids of every single machine. So to get data of each virtual machine we need to make GET requests repeatedly for every machine with included 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 have just made the 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 integration with external applications. It misses key actions and crucial data required for management of oVirt data centers. We also have to mention the issue with wrong response headers for OPTION method which has not been fixed yet. Another imperfection is probably missing vendor field in the list of entities which may result in downloading useless data.

5.5 Sagas

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 the creation of callback hell. The difference between generator functions and classic callbacks is that generators remain paused until the effect is resolved which is the main advantage for the callback hell prevention.

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 inefficient, we will make one main `rootSaga`. The `rootSaga` is also a generator function which yields the list of saga helper functions `takeEvery` and `takeLatest`. Those helper functions map every created saga to the specific action. Consequently, these functions allow us to spawn one of the predefined sagas by dispatching an action. The behavior of spawned sagas also depends on the kind of provided helper function as described in [4.3.1](#). Dispatching an action is not the only way to spawn a saga but it has certain advantages which are suitable for the Redux workflow. Generally, it is much easier for debugging through Redux-devtools because it shows order of actions which spawn generators one by one.

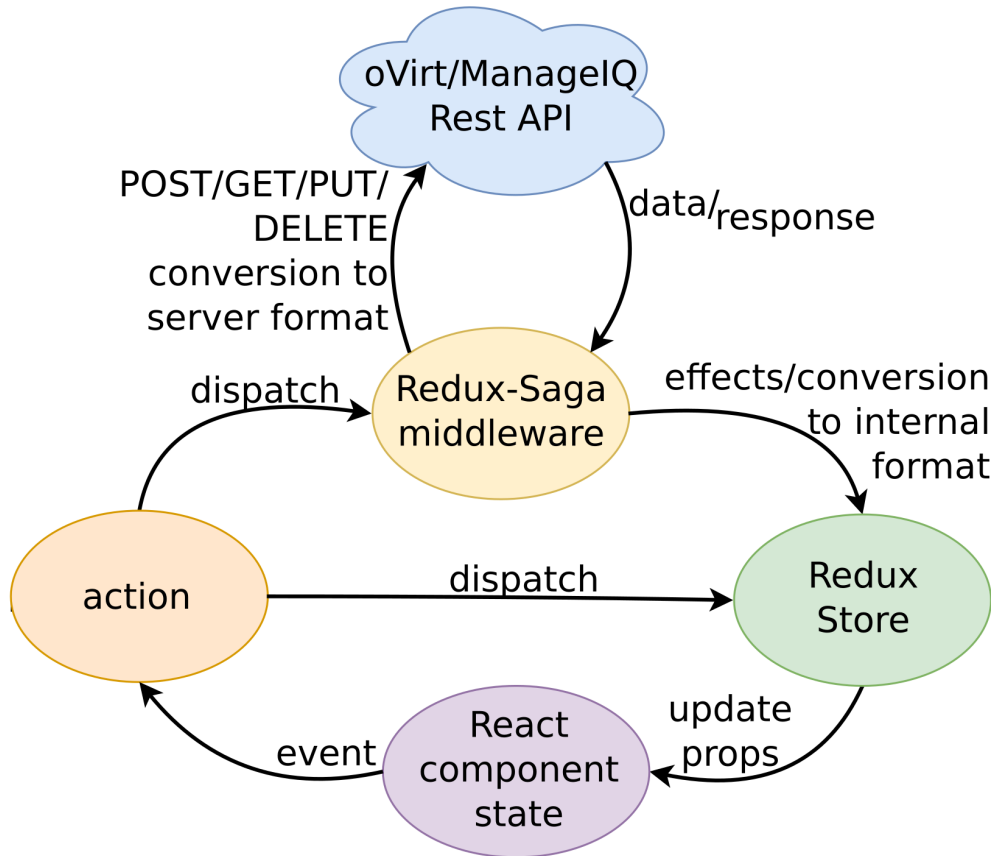


Figure 5.1: Application architecture

With all generators prepared and `rootSaga` running by the middleware, the application execution can continue. Assuming that we are using the development mode and we are logged in, the application will start with fetching 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 being continuously 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.

Now, that the application has all the necessary data and we are able to see the list of virtual machines ready for user interaction. The user can click on the *Add virtual machine* button located above the list of virtual machine which triggers an action `SHOW_BLANK_DIALOG` caught by the `rootSaga` and spawns `showAddNewVm` saga. This saga will dispatch another set of actions required to open the dialog and several actions which will make sure that the dialog is initialized with empty values. There are also exceptions to this rule, for example some fields are set from the default template named *Blank*.

Furthermore, the Code Sample 5.5 explains initiation of dialog that serves for editing virtual machines. Similarly like the Add virtual machine dialog, the dialog can be opened by the action from the user. In this case, the user has to click on the pencil button located either in the list of virtual machines or in the virtual machine detail window accessible after

selecting any virtual machine from the list. Both of these options dispatch the same action which spawns the saga shown in the code sample. The saga is showing how we utilize generator functions within the project. In the beginning, the saga is provided with the action composed from a message type and a payload. The message type is in this case the string value `SHOW_EDIT_VM` which causes that the helper function defined in the `rootSaga` will spawn the `showEditVm` saga. The payload contains the data of virtual machine that is being edited.

The first effect of the saga will make sure that we have the most recent data of the virtual machine by refreshing informations about the virtual machine. The `yield` keyword provides an assurance that the saga will be paused until the effect is resolved. Therefore the saga will continue only after the most recent informations about the virtual machine are fetched from the engine and stored in the Redux store. Since the Redux is the single store of truth, the next step of the saga will lead to a selection of the virtual machine's data from the Redux store. These data are picked by a custom function written in the dedicated `Selectors.js` module. The function will deliver the data from the store based on the provided virtual machine id.

The next step updates the dialog type which has to be done before opening the dialog. There are few things which might be rendered or initiated differently based on `dialogType` value from the Redux store.

The remaining sequence of effects is passing through the properties of the virtual machine object selected from the Redux store. Properties are picked and dispatched to the different part of the Redux store which is responsible for state of the dialog. It is necessary to keep the right execution order because by changing some dialog fields we may trigger a change to several others. For example, changing the cluster will also cause the change of the template to the default value `Blank`. This is another case serving as a demonstration of the asset provided by generator functions because the saga is paused until the effect is resolved. Just note that as described in 4.3.2, the `put` effect is non-blocking and it won't wait for result of the effect. Since the generator is only making the plain updates to the store, all that matters is the sequence in which updates are dispatched. After updating all the dialog fields, the generator dispatches the last action which will open the dialog. The generator has ended and the dialog is shown and ready to handle the user input. The Code Sample 5.6 does not contain all the code from real generator because the intent is only to provide demonstration of Redux-saga workflow.

```
1 function* showEditVm (action) {
2   yield fetchSingleVm(getSingleVm({vmId: action.payload.vm.get('id')}))
3   const vm = yield Selectors.getVmById(action.payload.vm.get('id'))
4   ...
5   yield put(updateDialogType('edit'))
6   const cluster = Selectors.getClusterById(vm.get('cluster').get('id'))
7   yield put(updateCluster(cluster))
8   ...
9   yield put(updateTemplate(template))
10  yield put(updateVmName(vm.get('name')))
11  yield put(updateVmDescription(vm.get('description')))
12  ...
13  yield put(openVmDialog())
14 }
```

Code sample 5.2: Saga responsible for initiating and showing the Edit VM dialog

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 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 users. The input 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 users 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 a 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.



Figure 5.2: `LabeledTextField` number variant

`LabeledSelect` component provides users with a restricted list of values but only one value can be selected at time. In this component I have included two possible widgets with different features and look. The type of widget can be configured via `selectClass` prop.

The first, classic widget provides users with simple 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). Users can simply start typing the desired option and if it is available, they can select it. The biggest advantage of the type-ahead functionality comes with bigger lists of data where the values may be found much quicker compared to classic widget. 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.

`Alert` can display a 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 the 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

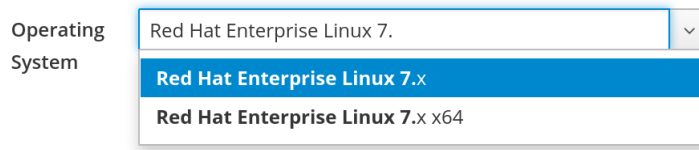


Figure 5.3: LabeledSelect example with demonstration of type-ahead functionality

are being detected by saga-middleware by examining the code from the server response. As soon as the error is detected, saga will dispatch an action with an update which contains the error message provided by the 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.

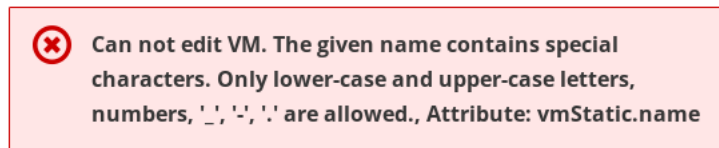


Figure 5.4: Error message from API displayed by error

`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 can be turned on or off. 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 users 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 the 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 about to be edited as shown in the Figure 5.6.

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. Not all props are required to be used, only those which suit our current need. The props also come with the certain restrictions. Each of the sub-components defines its own `.propTypes` 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. the `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 of the dialog field to a class property of the parent component. As an suitable example we can use the 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 used only as callbacks. The props are also used to carry values which are supposed to be rendered by sub-components. 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 the field which represents the 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 valid data in every possible situation, 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 the 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 the Redux store inside the `connect()` method provided by react-redux library.

`editTemplateDialog` is the 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 few 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 result is in another valuable 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 Testing and verification

The application was tested continuously during the whole development process. The Redux-devtools extension have been used primarily for testing reducers, actions and the content of the Redux store with occasional help from the developer console. The testing process included manual execution of certain scenarios, including opening and closing dialogs, switching values of certain fields and sending the data to the engine. The extension also served as a verification tool for sagas. Since sagas were dispatching actions, the extension allowed us to check the proper order of actions, their payloads and changes made by them to the store.

Another part of testing process was to verify data that should be created or updated by dialogs. In this case, verification process was in the beginning as simple as checking the return code from HTTP protocol. But as we were adding more fields after field to dialogs, it was required to verify every altered value. The main verification tools were the already implemented oVirt solutions. We started with the Administration Portal and moved to the REST API, where we were able to simply open a web page containing data of virtual machine stored as XML and verify changed data by simply refreshing the web page. After every successful change made to the oVirt engine, the generator will fetch updated data and store them in the Redux store. Therefore we were able to determine if the fresh data fetched from the oVirt engine include the change we made through the Redux-devtools extension or by examining the data shown by the `editVM` dialog.

Edit Virtual Machine

Cluster	x86_64	✕
Template	Blank	✕
Operating System	Other OS	✕
Name	mkrajnak_test1	
Description	mkrajnak_test_desc123	
Comment	mkrajnak_test_comment123	
Memory (MB)	1024	⬇ ⬆ ⬇
Max memory (MB)	4096	⬇ ⬆ ⬇
Memory guaranteed (MB)	1024	⬇ ⬆ ⬇
CPU	1	
Memory balloon	<input type="checkbox"/> OFF	
Delete Protection	<input checked="" type="checkbox"/> ON	
Start in paused mode	<input checked="" type="checkbox"/> ON	

Console

Graphics Protocol	SPICE	⌵
Smartcard	<input type="checkbox"/> OFF	
Spice file transfer	<input type="checkbox"/> OFF	
Spice Copy Paste	<input type="checkbox"/> OFF	

Boot order

First device	Network (PXE)	⌵
Second device	CD-ROM	⌵

Other hardware

Sound Card	<input type="checkbox"/> OFF	
Virtual SCSI	<input type="checkbox"/> OFF	
IO Threads	<input checked="" type="checkbox"/> ON	
High Availability	<input checked="" type="checkbox"/> ON	

Figure 5.6: Dialog for editing virtual machines

Chapter 6

Alternative solution

This chapter describes a possible alternative solution that was proposed and tested during the development of the thesis. This solution uses all the previously mentioned technologies with small difference in state management.

Most of the content of generator functions located in `sagas.js` module was moved to stateful React components. Operations which remained in sagas were used only for asynchronous network operations and conversion to internal format. Both dependency handling and state management were completely moved to the `addVmDialog`. Dialogs weren't dispatching any of changes to the Redux store, the state of dialogs was handled exclusively by the React component. The dependency handling was provided by internal functions inside the component. Dialogs were capable only of a couple changes to the Redux store. These changes were closing the dialog and sending an actual request with the data from the dialog fields.

The burden that was placed on the Redux store was decreased because the actions weren't dispatched as often as in the proposed solution. The alternative solution provided suitable results at time but there was a concern that with increasing amount of dialog fields, it might lead to creation of the callback hell. The main reason behind this concern is a need to dynamically fetch the additional resources during the work with the dialogs. Therefore I decided to use the approach with generator functions which offer a secure way to enforce the execution order of callbacks and callback hell prevention.

Chapter 7

Conclusion and future development

The goal of this thesis was to design and implement a generic configuration interface for virtual machines and embed it to the oVirt Web Admin. The development focus is being shifted to oVirt Web UI project with a chance that current oVirt web based tools might be obsoleted in future by this project. For this particular reason the proposed solution was embedded to the oVirt Web UI instead.

The solution has mapped all properties of the virtual machine which is the biggest entity from the configuration standpoint. All dependencies and constraints were documented to the graph (Appendix A) that will possibly help with the project development in the future. The only alternative source of data shown in graph is the source code of the oVirt engine.

The implemented back-end module is able to download list of virtual machines, templates and clusters. The module also includes methods for conversion of these entities to their individual internal representations. Furthermore, it contains methods for creating and editing virtual machines and also other methods which mostly secure the communication with the oVirt engine.

The thesis has also made a small research by experimentally implementing an alternative back-end module which uses the ManageIQ's REST API instead of the oVirt's REST API. The result of research has proven that the ManageIQ's REST API is not on the same level as the oVirt's REST API. Additionally, the research lead to a discovery of the bug caused by the wrong CORS implementation. This discovery might help to improve the ManageIQ project in the future. The arrangement of oVirt entities provided by the ManageIQ REST API is not ideal and can negatively influence the performance of external application by downloading useless data. Since the API is missing some key entities and actions, the application was ported only in read-only mode. The read-only mode can only display the list of available virtual machines and requires disabling browser security.

I also experimented with the alternative solution for state management of dialogs secured only by React itself, without Redux. Even though the Redux is not recommended to be used for state management of the dialogs, our dialogs are unique and we have proven that it is possible and also very important for solving the callback hell which have caused many problems in the previous dialog implementation. The solution of the callback hell has been achieved through ES6 generator functions. The proposed callback hell solution is demonstrated on the implemented virtual machine dialog with 23 fields.

The final results include multiple dialogs. Probably the most important dialog is able to create a new virtual machine and warn about possible errors which appeared during this process. The second dialog allows the user to edit parameters of previously created virtual

machines. The last of created dialogs offers a possibility to change parameters of templates managed by the oVirt engine.

The proposed React components also have some flaws. General flaw in components is caused by usage of jQuery which should be avoided in React. The Patterfly have already identified the problem and started a custom project which should provide components designed especially for React without jQuery as additional dependency¹. Another minor issue might be the user experience flaw caused by the proposed combo box solution which requires additional click to cancel the previously selected value before selecting a new value from the list. The problem was also identified on Github page² of component but the solution has not been included in upstream yet. The proposed solution only removes the X button which appears when the value is selected in the widget. The change can be done manually in source code of the component but it would lead to another problem with distribution of altered npm package.

The implementation side of this thesis has also tried to keep touch with an upstream project³. The smaller patch with few dialog fields, React components and most of the back-end infrastructure has been included in adjusted form. The development of oVirt Web UI project continues and one of the first versions is available as a preview in oVirt 4.1. The future development should divide dialog to smaller sections and implement the remaining dialog fields.

¹<https://github.com/patternfly/patternfly-react>

²<https://github.com/danielfarrell/bootstrap-combobox/issues/149>

³<https://github.com/oVirt/ovirt-web-ui>

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/>

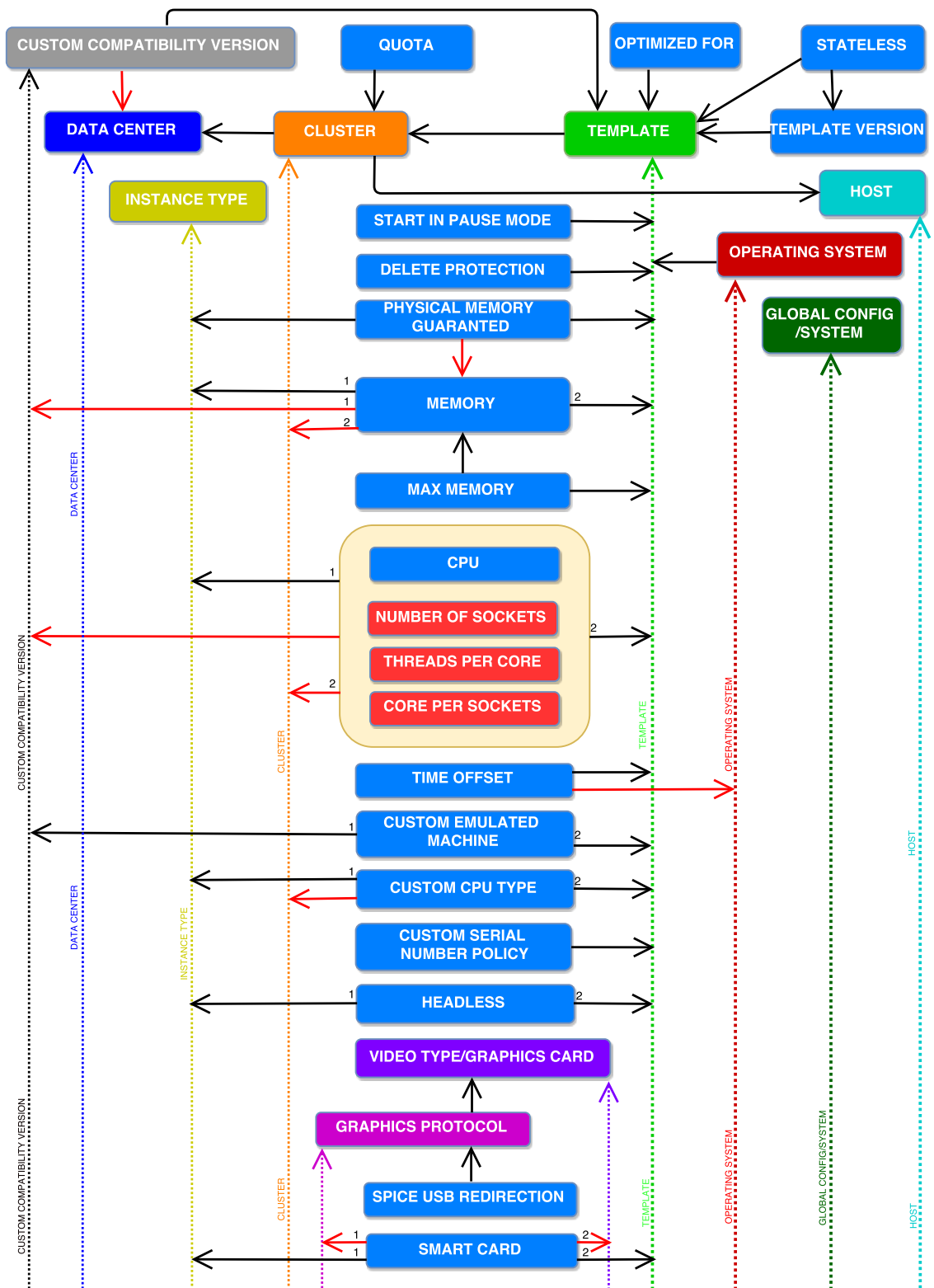
- [14] *Npm website*. [Online; visited 17.04.2017].
Retrieved from: <https://www.npmjs.com/>
- [15] *Ovirt documentation*. [Online; visited 10.04.2017].
Retrieved from: <https://www.ovirt.org/>
- [16] *Patternfly documentation*. [Online; visited 10.04.2017].
Retrieved from: <https://www.patternfly.org/pattern-library/>
- [17] *PostgreSQL*. [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/>

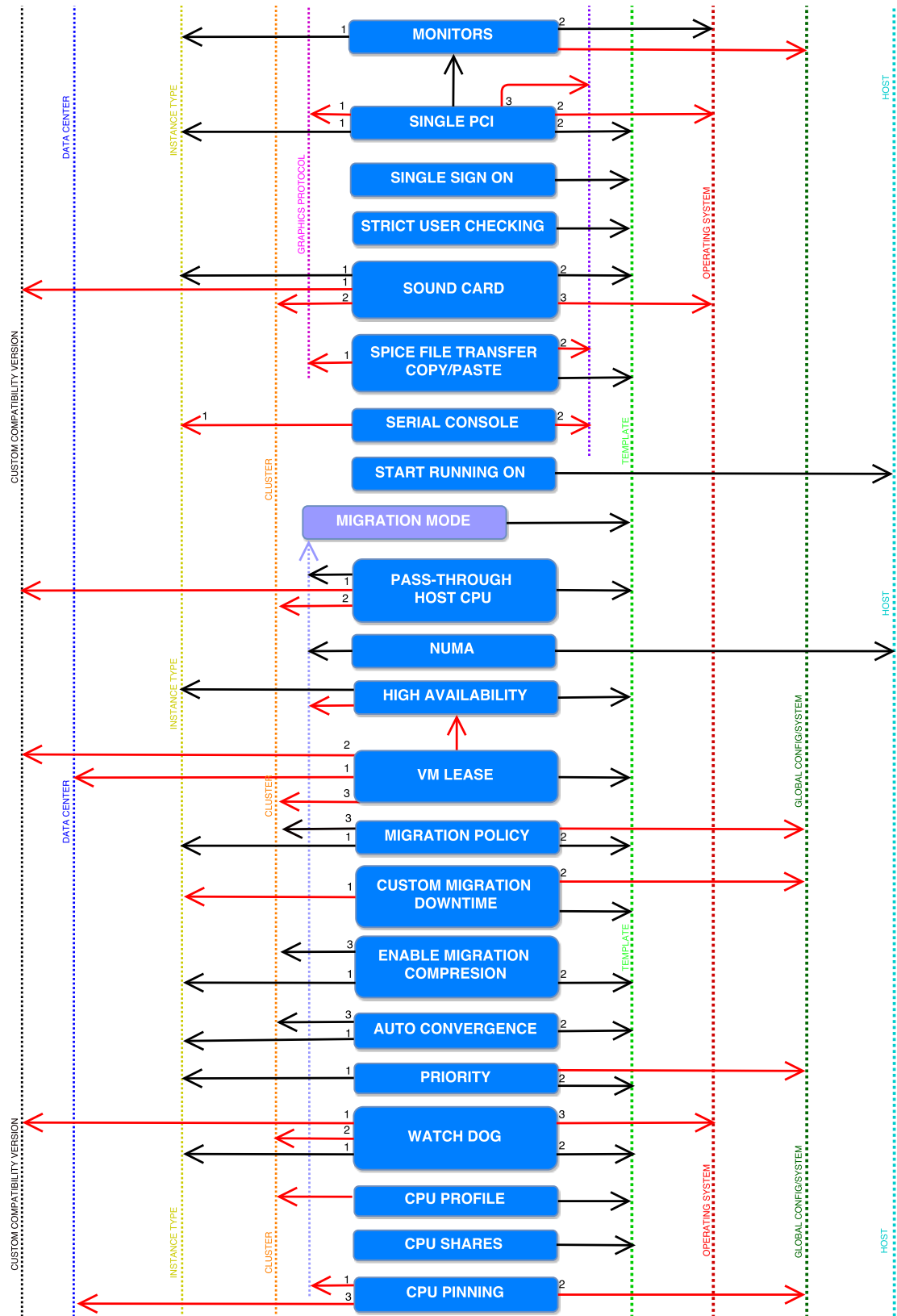
- [30] Jansen, G.: *Managing heterogeneous environments with ManageIQ*. [Online; visited 13.04.2017].
Retrieved from: <https://lwn.net/Articles/680060/>

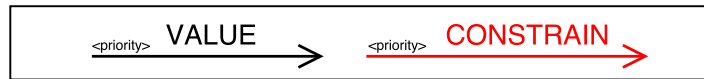
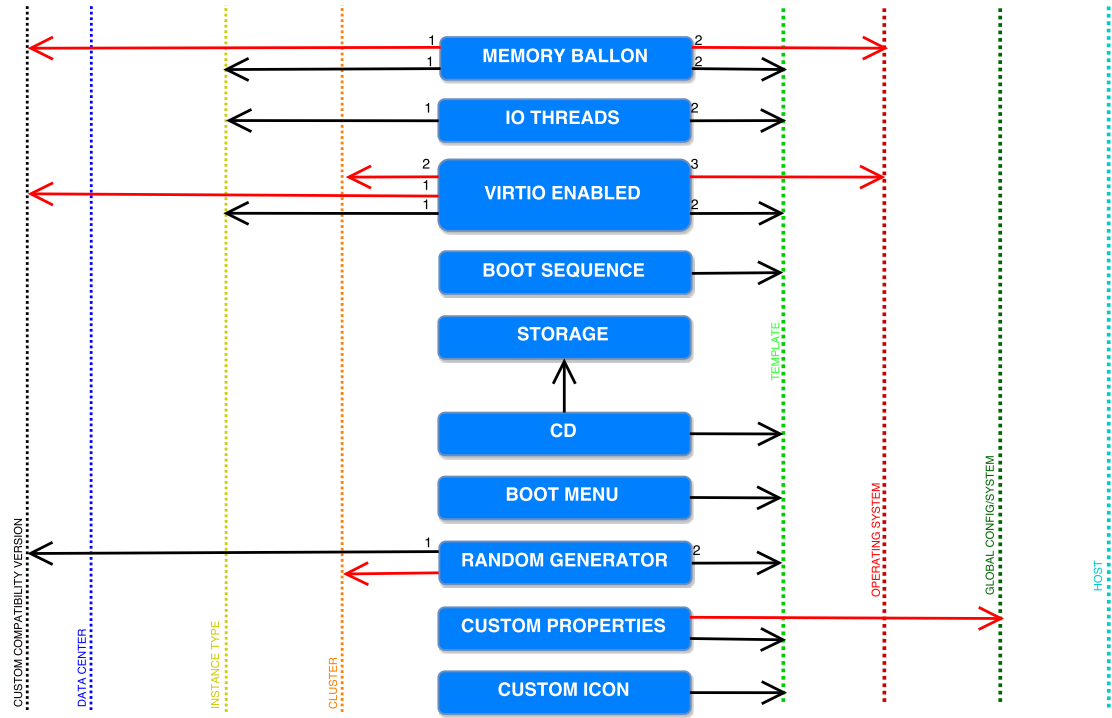
Appendices

Appendix A

Complete oVirt virtual machine dialog dependency graph







Appendix B

Installation guide

B.1 Prerequisites

Application should be started via development mode with the engine url as the parameter with following prerequisites:

1. A running oVirt engine, 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 - Linux

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

B.3 ManageIQ API variant installation - Linux

The ManageIQ variant is accessible only via Google Chrome and it is required to run it from terminal with parameters `-disable-web-security -user-data-dir`. These parameters will disable all the security.

1. Copy and unzip the contents of the attached CD
2. Open a terminal
3. Make sure that you are in correct folder (ManageIQ API): `cd ovirt-web-ui-miq`
4. Make sure npm is installed: `dnf install npm`
5. Install yarn: `npm install yarn`
6. Install and update all required project dependencies: `yarn install`
7. Run oVirt-web-ui: `ENGINE_URL=https://engine_address yarn start` (change the url to your oVirt engine)
8. From another terminal window/tab run `google-chrome -disable-web-security -user-data-dir` and type the url and port of the app (usually `localhost:3000`)

Appendix C

Contents of attached CD

1. **bachelor-thesis.zip** – folder containing the git repository with pdf, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ sources and images used in documentation
2. **ovirt-web-ui.zip** – folder containing the git repository with sources and implementation
3. **ovirt-web-ui-miq.zip** – folder containing the git repository with sources and implementation of ManageIQ backend