



Rapport de stage

Participation au développement
de Xen Orchestra, interface web
de gestion de machines virtuelles

Bois Cédric
Master WIC
Année 2016 - 2017

Remerciements

Je tiens à remercier Julien Fontanet qui s'est rendu disponible, m'a apporté son aide et m'a guidé tout au long de cette période, ainsi que Pierre Donias et Olivier Lambert qui m'ont également été de bon conseils.

Plus généralement, je remercie Vates de m'avoir accueilli au sein de son équipe.

Sommaire

Introduction.....	1
L'entreprise	2
A propos de Vates.....	2
Gestion de projet.....	2
Répartition du temps de travail	3
Technologies utilisées	4
NodeJS	4
Yarn	4
ReactJS.....	4
Babel	5
Redis.....	5
Travail réalisé.....	6
Prise en main des différents outils	6
ES6.....	6
Redux	6
Lodash.....	8
Développement du module withState	8
Objectifs	8
Réalisation.....	8
Tests	12
Développement de Xen Orchestra	13
Présentation de Xen Orchestra.....	13
Afficher une icône orange quand l'hôte est indisponible	15
Demander une confirmation quand une mise à jour est faite depuis la vue d'un pool.....	16
Afficher un message rappelant les fichiers de sauvegarde compatibles	18
Afficher depuis combien de temps une machine est éteinte	19
Rendre possible la création de groupes de machines virtuelles	20
Conclusion	29

Introduction

Dans le cadre du master Web Informatique et Connaissances, j'ai effectué un stage de cinq mois au sein de l'entreprise Vates. Ce stage s'est déroulé du 30 janvier au 30 juin 2017.

Le but de ce stage est de découvrir le monde professionnel, tout en développant de nouvelles compétences, ou en approfondissant des connaissances qui ont pu être vues en cours, ou lors de projets personnels. Pendant de ma recherche de stage, je souhaitais trouver un sujet usant d'une technologie utilisant le langage Javascript. Même en ayant été initié à ce langage de programmation durant ma formation, je ne me sentais pas à l'aise avec celui-ci. Cette technologie étant largement utilisée dans le monde professionnel aujourd'hui, il me paraissait important d'étendre mes connaissances à son sujet.

Le projet proposé par Vates répondait entièrement à mes attentes puisqu'il est de participer au développement du logiciel libre Xen Orchestra à l'aide de ReactJS.

Ce rapport commence par une présentation de l'entreprise et son mode de fonctionnement, pour situer l'environnement dans lequel j'ai pu évoluer au cours de cette période. Viens ensuite une présentation des différentes technologies que j'ai pu utiliser puis une présentation du travail que j'ai réalisé.

L'entreprise

A propos de Vates

Vates est une entreprise de développement open source située à Grenoble, au 17 rue Aimé Beret. Elle a été créée en 2012 par Olivier Lambert, Julien Fontanet et Nithida Vialle.

De 2012 à 2014 Vates faisait de la prestation de solutions open source pour les entreprises. Depuis 2014, Vates se consacre uniquement au développement, à la maintenance et au support du logiciel Xen Orchestra également open source. L'entreprise est composée de six salariés : un directeur technique, un directeur de projet, deux développeurs, un chef de projet web marketing, une responsable administrative et financière

Vates est également membre du réseau Minalogic, qui a pour but de regrouper et dynamiser les filières du numérique en région Rhône-Alpes. Elle regroupe 300 entreprises afin de les accompagner dans leurs projets d'innovation, et leur donne une meilleure visibilité

Durant ma période de stage, j'ai rejoint l'équipe développement.

Gestion de projet

Xen Orchestra est développé grâce essentiellement à trois outils de gestions :

- Mattermost qui est un chat sur lequel l'ensemble de l'équipe peut échanger. En effet, une partie de l'équipe effectuant son activité en télétravail, un outil de communication est nécessaire
- Git, Github et Gitlab: Vates utilise l'outils de gestion et de versions git qui permet de garder une traçabilité du développement du logiciel ainsi que de travailler en équipe sur un projet. Github et Gitlab sont des services web d'hébergement et de gestion du développement de projets utilisant git. Xen Orchestra est disponible dans un dépôt public de Github. Dans ces plateformes, chaque dépôt possède une section dans laquelle tout le monde peut ouvrir des tâches qui peuvent correspondre à un rapport de bug ou au développement de nouvelles fonctionnalités. Au sein de Vates, ces tâches sont ensuite assignées à un développeur par un chef de projet. De plus, git permet de travailler en équipe sur un projet. Ainsi, pour développer une fonctionnalité, il est possible de créer une branche sur laquelle il va être possible de coder de manière indépendante au projet. Lorsque le développeur estime qu'il a fini sa tâche, il va pouvoir faire une demande de fusion de sa branche de travail avec la branche principale. Le code produit va être relu par le directeur technique et éventuellement d'autres développeurs. Une discussion sur le code produit va alors démarrer, entraînant des modifications. Lorsqu'il n'y a plus aucun point à éclaircir, la demande de fusion est acceptée et les modifications seront alors ajoutées à la branche principale.
- Crisp, qui est également une plateforme de chat. Elle sert au service de support pour Xen Orchestra. Ainsi, l'équipe de développement y répond aux questions de clients, ce qui permet de faire ressortir des défauts dans la conception du logiciel.

Répartition du temps de travail

Durant ce stage, plusieurs missions m'ont été confiées. Dans un premier temps j'ai dû prendre en main la bibliothèque ReactJS, j'ai ensuite développé un décorateur de composant React, puis, j'ai contribué au développement de Xen Orchestra.

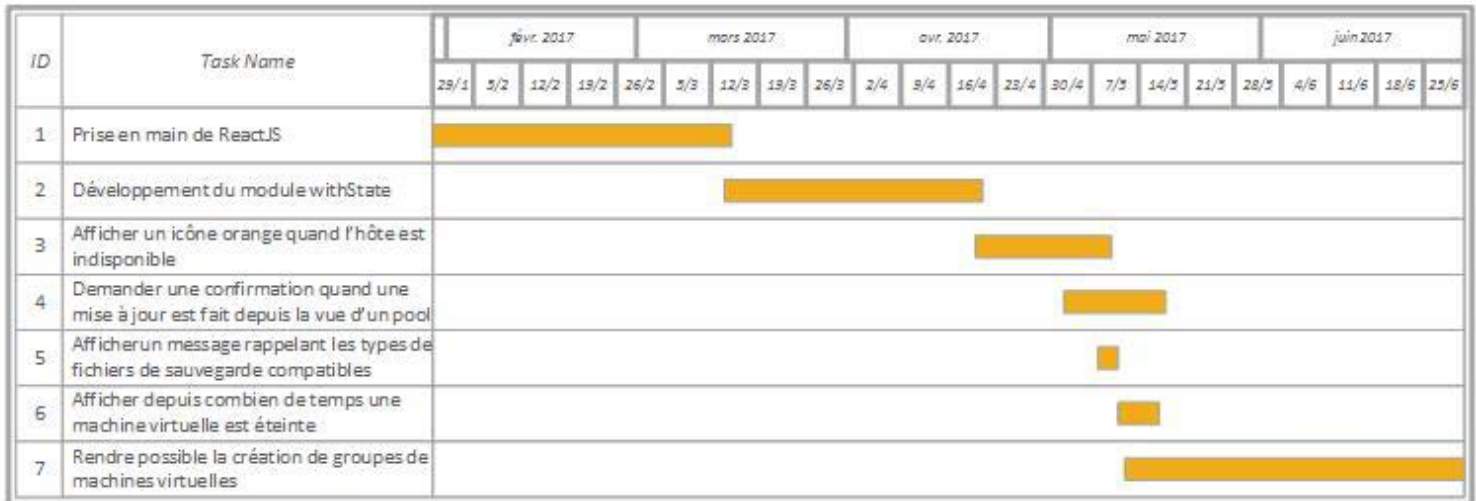


Figure 1: Diagramme de Gantt

Technologies utilisées

NodeJS

NodeJS est une plateforme Javascript possédant une librairie de serveur http, ce qui permet de faire tourner un serveur web écrit en Javascript.

Yarn

Yarn est un gestionnaire de paquets. Il permet donc d'installer des modules déjà développés par une communauté, et mis en libre accès, évitant ainsi de devoir développer des fonctionnalités qui existent déjà. De la même manière que npm, yarn utilise un fichier package.json situé à la racine du projet dans lequel sont définis les paquets desquels le projet dépend, ainsi que des scripts permettant notamment de travailler sur celui-ci tout en l'exécutant et donc de visualiser le rendu (start), de lancer une procédure de test (run test), ou encore de générer une version installable de l'application (build).

ReactJS

React est une bibliothèque Javascript développée par Facebook. Le but de React est de faciliter le développement d'interface web par la création de composants permettant de générer un morceau de page html.

Un composant React est constitué de :

- Un State qui un objet ¹ Javascript dans lequel vont être stocké les données nécessaires au fonctionnement du composant.
- Une variable props qui permet de recueillir des données transmises par un autre composant
- Une fonction render qui renvoie un objet représentant la vue du composant

La manière la plus simple de déclarer un composant React se fait sous la forme d'une fonction. Cette fonction prend en paramètre les props du composant, et retourne son rendu. Dans ce cas, on parle de composant stateless

```
1 function Welcome(props) {  
2   return <h1>Hello, {props.name}</h1>;  
3 }
```

Figure 2: Composant stateless

Un composant peut aussi être déclaré à l'aide d'une classe. Ci-contre, un exemple de composant React simple, déclaré de cette manière. C'est un compteur qui reçoit sa valeur initiale par la variable props et qui est ensuite stockée dans le state. Cette valeur est modifiée grâce aux fonctions d'incrément et de décrémentation, exécutées lorsqu'un clic est effectué sur les boutons correspondants. Ce type de

```
import React, { Component } from 'react';  
  
export default class Compteur extends Component {  
  constructor (props) {  
    super(props)  
    this.state = {cpt: props.cpt}  
  }  
  
  _incrément = () => this.setState({cpt: this.state.cpt + 1})  
  _décrément = () => this.setState({cpt: this.state.cpt - 1})  
  
  render () {  
    return (  
      <div>  
        <p>{this.cpt}</p>  
        <button onClick={this._incrément}>Incrément</button>  
        <button onClick={this._décrément}>Décrément</button>  
      </div>  
    )  
  }  
}
```

Figure 3: Composant stateful

¹ Ce qu'on appelle un objet en Javascript est en réalité un tableau associatif comportant à chaque clé unique, une valeur associée, déclaré entre accolades.

composant est dit *stateful*.

Concernant l'instanciation du module, celle-ci s'effectue en déclarant une balise portant le nom du module. On peut également voir sur l'exemple que la valeur initiale transmise à la variable `props` est passée dans un attribue de la balise.

Pour pouvoir modifier le state, chaque composant possède une méthode « `setState` » qui possède deux signatures :

- L'une prend en argument un objet. La fonction va alors mettre à jour les champs qu'il a en commun avec l'argument.
- L'autre prend une fonction en argument à laquelle est passé l'ancien state, et renvoie le nouveau

```
1 import Compteur from './Compteur'
2 import React, { Component } from 'react';
3
4 class App extends Component {
5   render() {
6     return (
7       <div>
8         <Compteur cpt={3} />
9       </div>
10    );
11  }
12 }
13
14 export default App;
```

Figure 4: Instanciation d'un composant

Babel

Les portions de code observables ci-contre sont écrites en EcmaScript 6 qui est un langage de script standardisé qui n'est pas supporté par les navigateurs, le code doit donc être transpilé en ES5 pour pouvoir être exécuté. Babel est le transpileur utilisé par Vates pour effectuer cette tâche.

Redis

Redis est un gestionnaire de bases de données noSQL utilisé par Xen Orchestra. Il conserve la totalité des données en RAM pour éviter des accès au disque dur et fournir les meilleures performances possibles.

Travail réalisé

Prise en main des différents outils

Pendant cette phase de prise en main, j'ai développé une petite application de type site vitrine. Elle contient différentes pages comme une authentification, permettant d'accéder à une page privée, ou encore un gestionnaire du thème de l'application.

ES6

Durant toute cette période, j'ai appris à manipuler le langage ES6. Je n'avais que très peu utilisé de langage de script que ce soit durant ma formation comme par mes projets personnels. Ces langages étaient, à mon sens, très particuliers de par leur syntaxe, notamment car ils permettent de déclarer une fonction dans une autre, ce qui est très utilisé et pratique, mais à mon avis difficilement lisible pour un non initié. De plus, la syntaxe de ES6 permet une simplification du code grâce des opérateurs qui existent dans peu de langages, si ce n'est le seul, qu'il m'a fallu apprendre à utiliser. Voici quelques exemples de ces simplifications de code :

Déclaration d'une fonction inline : De la même manière que dans d'autres langages où un "if" pourrait être écrit sans accolades si celui-ci ne comporte qu'une seule instruction, il est possible de déclarer une fonction de cette manière en ES6. De plus cette fonction renverra le résultat de l'instruction. Dans l'exemple ci-dessous, la fonction incrémentation prend en paramètre `cpt` et renvoie `cpt + 1`

Suppression des parenthèses des paramètres d'une fonction : Lorsqu'une fonction prend un seul paramètre, il est possible d'omettre ses parenthèses, comme dans l'exemple ci-dessous.

```
incrémentation = cpt => cpt + 1
```

Figure 5: Exemple de fonction

Le Destructuring : Lorsqu'il est nécessaire de déclarer une ou plusieurs variables qui contiendront la valeur d'un champ d'un objet, il est possible de l'écrire de la manière suivante.

```
const { foo: fooVar, bar } = this.props
```

Figure 6: Exemple de destructuring

Ainsi, la valeur contenue dans le champs `foo` de la variable `props` va être injectée dans la constante `fooVar`, et la valeur de `this.props.bar` va également être injecté dans une constante `bar`.

L'opérateur « ... » : Cet opérateur s'applique sur un objet. Il permet de copier tous les champs qu'il contient. Ainsi les deux variables suivantes possèdent des champs semblables mais ne sont pas égales, car une copie a été effectuée.

```
const a = {foo: 'foo', bar: {bar: 'bar'}}
const b = {...a}
```

Figure 7: Exemple d'utilisation de l'opérateur "..."

Redux

Redux est une librairie dont le but est la gestion de différents états de l'application, et donc de centraliser les données nécessaires au fonctionnement de l'application. Elle permet donc la gestion d'un state. Pour ce faire, redux met à disposition des fonctions de créations d'actions, de sélecteurs et de "reducers".

Ce qu'on appelle une action dans Redux est déclaré de la manière suivante :

```
5   const { changeTheme } = createActions({
6     CHANGE_THEME: (theme) => ({theme})
7   })
```

Figure 8: Déclaration d'une action

Elle permet de renvoyer la liste des paramètres qui vont être nécessaire au changement d'état. Ici, la méthode `createActions` fournie par Redux permet de déclarer plusieurs actions facilement. La fonction `changeTheme` prend donc en argument un nouveau thème qu'elle va renvoyer encapsulé dans un objet.

Lorsqu'une action est déclenchée, celle-ci va appeler un reducer associé à cette action. L'intérêt d'utiliser un reducer est de mettre à jour le state. Le reducer prend donc en argument le state et la liste des paramètres renvoyée par l'action et va renvoyer un nouveau state. Voici le reducer associé au changement de thème.

```
13  export const reducer = combineReducers(red, {
14    [changeTheme]: (state, {theme}) => theme
15  })
```

Figure 9: Déclaration d'un reducer

Redux introduit également la notion de sélecteurs. Un sélecteur prend en paramètre le state et renvoie un champ de celui-ci

```
export const getTheme = state => state.theme
```

Figure 10: Déclaration d'un sélecteur

Un composant va donc pouvoir utiliser les actions et sélecteurs, mais pas directement. En effet, si on observe, un sélecteur nécessite d'avoir accès au state "global", celui-ci n'étant pas disponible dans le contexte du composant. De plus, si on observe l'action ci-dessus, l'exécution de celle-ci ne permet de modifier directement le state. L'envoi du résultat d'une action au reducer se fait grâce à une fonction fournie par Redux appelée « dispatch », mais celle-ci n'est pas non plus disponible dans le contexte d'un composant. Pour ce faire, Redux fournit un décorateur² appelé `connect`. Cette fonction prend en argument une fonction à laquelle le state sera passé en argument, ce qui permet d'utiliser les sélecteurs, ainsi qu'un objet. Pour chacun des champs de cet objet, une nouvelle

```
const ThemeChooser = connect(
  state => ({
    theme: getTheme(state)
  }),
  { changeTheme }
)(({theme, changeTheme}) =>
  <div>
    <form>
      <select value={lodash.findKey(themes, (t) => t === theme)} onChange={(e) => changeTheme(themes[e.target.value])}>
        {lodash.map(themes, (_, key) => <option value={key} key={key}>{key}</option>)}
      </select>
    </form>
  </div>
)
```

Figure 11: Utilisation du décorateur `connect`

fonction exécutant `dispatch` sur celui-ci est générée, permettant de modifier le state via ces actions. `Connect` retourne une autre fonction qui prend en paramètre un composant en lui injectant les

² Un décorateur est une fonction qui prend un composant en argument et en renvoie un nouveau

valeurs calculées précédemment et retourne un nouveau composant. Voici un exemple de l'utilisation du décorateur connect.

Lodash

Lodash est une bibliothèque d'utilitaire Javascript que j'ai eu l'occasion d'utiliser très fréquemment. Elle contient des fonctions de calculs mathématiques et de traitement de collections ou de fonctions.

Développement du module withState

Objectifs

Le module withState³ est un décorateur de composant. Son but est d'abstraire la gestion du state propre au composant qu'il décore. Dans un premier temps, l'objectif était de manager un state à un seul niveau⁴, pouvant posséder plusieurs sous-states⁵. Ensuite s'est posé la question de générer automatiquement une fonction de remise à la valeur initiale pour chacun des champs de l'objet, ainsi que la gestion d'un state avec un nombre de niveau non définit.

Réalisation

```
const Connection = connect(
  state => ({
    user: getUser(state)
  }),
  authActions
)(withState(
  {
    credentials: {
      onLoginChange: (_1, _2, {target}) => ({ loginField: target.value}),
      onPasswordChange: (_1, _2, {target}) => ({password: target.value}),
      onSubmit: ({loginField, password}, login, e) => {
        e.preventDefault()
        login(loginField, password)
      }
    },
    {
      credentials: {loginField: ({ initLogin }) => initLogin, password: ''}
    }
  }
)(class extends PureComponent {
  render () {
    return componentView
  }
}))
```

Figure 12: Utilisation de la première version de withState

state nommé « credentials » comprenant des champs « loginField » et « password ». De plus la variable props du composant contiendra des fonctions « onLoginChange » et « onPasswordChange » renvoyant une nouvelle valeur du champ concerné pour chacun d'eux, respectivement « loginField » et « password », et une fonction « onSubmit » qui traite ces deux champs.

Comme expliqué précédemment, withState est un décorateur. Il renvoie donc une fonction qui en renvoie une autre. Cette dernière prend en argument un composant react.

Il est d'abord nécessaire de penser à la structure de notre module. Elle présente ci-contre.

Voici à quoi devrait ressembler la déclaration d'un composant utilisant withState

Dans cet exemple, le premier argument est défini par un objet comprenant des fonctions permettant de modifier les valeurs qui seront dans le champs « credentials » du state du composant généré. Chacune de ces fonctions prend trois arguments, dans l'ordre : le state associé à la fonction (ici, state.credentials), la variable props du composant, ainsi qu'un argument passé lors de l'appel de la fonction.

Le second argument de withState représente la structure initiale qu'aura le state.

Ici le composant résultant du traitement de withState contiendra un state avec un sous-

```
const withState = (actions, initialValue) => Component =>
class extends PureComponent {
  constructor (props) {
    super(props)
  }

  render () {
    return <Component
      {...this.props}
      {...this._actions}
      {...this.state}
    />
  }
}
```

Figure 13: Structure du décorateur withState

³ Adresses des dépôts git de développement de withState : https://gitlab.com/vates/www-xo-2/merge_requests/10/, https://gitlab.com/vates/www-xo-2/merge_requests/12/

⁴ On entend ici par un seul niveau que chaque champ du state représente un sous-state qui lui, contient des valeurs.

⁵ Dans les exemples suivant, le state n'est composé que d'un seul sous-state.

L'objet contenant les actions et la structure du state sont reçus, respectivement, dans les arguments « actions » et « initValue », et le composant qui sera décoré, dans la variable Component. La fonction déclare et retourne un composant qui aura été décoré. Il possède un constructeur dans lequel les traitements permettront de construire le state final, ainsi que la liste de fonctions autorisant sa gestion. Ces fonctions seront stockées dans une variable « _actions ». Le composant que l'on décore est ensuite instancié en lui transmettant les variables props, _actions et state. Elles seront reçues dans la variable « props » de Component.

```

this._actions = {}
forIn(actions, (groupActions, key) => {
  forIn(groupActions, (action, actionName) => {
    this._actions[actionName] = (...args) => {
      const res = action(this.state, this.props, ...args)
      this.setState(res !== undefined ? res : this.state)
    }
  })
})

```

Figure 14 Construction de la liste d'actions dans la première implémentation de withState

Il est ensuite nécessaire de construire le state contenu dans « initValue ». Celui-ci pourrait-être directement assigné au state, mais nous autorisons qu'une valeur provenant de « initValue » puisse être une fonction qui initialise une valeur du futur state via les propriétés passées au composant. Pour ce faire, il est nécessaire de parcourir les sous-states de l'objet « initValue », on initialise la valeur du nouveau state via la fonction si s'en est une, sinon avec le champs directement.

La construction de l'objet contenant les actions se fait en parcourant l'objet « actions ». Pour chacune des fonctions qui seront enregistrées dans « _actions » (ici, « onLoginChange », « onPasswordChange » ou « onSubmit »), lorsqu'elle seront exécutées, elles renverront une valeur et le state sera mis à jour si la valeur de retour est définie, car, par exemple, ici onSubmit n'a pas de valeur de retour et renvoie donc « undefined ».

```

const initState = {}
forIn(initValue, (reducer) => {
  forIn(reducer, (init, name) => {
    initState[name] = typeof init === 'function' ? init(props) : init
  })
})
this.state = initState

```

Figure 15: Construction du state du composant décoré

De cette manière, le composant withState effectue les fonctionnalités voulue à l'origine, mais il prend en paramètres deux objets, ce qui implique de devoir les parcourir tous les deux. Le module pourrait donc être plus efficace en lui passant un seul objet. De plus, d'autres fonctionnalités comme la possibilité de remettre un champs à sa valeur initiale, ou alors de pouvoir générer un state à plusieurs niveau, ce qui rendrait possible d'intégrer un sous-state dans un autre sous-state, seraient intéressantes à ajouter à withState. Pour pouvoir se rendre compte des différentes fonctionnalités nécessaires à notre module, une liste de choses à faire est implémentée, car ce genre de petite application permet de se rendre compte des différentes fonctions et données nécessaires à son fonctionnement.

De l'implémentation du composant TodoList resulte l'utilisation du withState présenté en annexe III. Il prend encore deux arguments :

- Le premier est un objet ayant la structure du future state à gérer. Dans cet objet, chaque champ est composé soit d'un sous-state, soit d'un ensemble de valeurs dont l'une d'elles est une valeur init qui permet d'initialiser la valeur du sous-state, les autres sont des fonctions permettant de mettre à jour cette valeur. On définit donc qu'un sous-state qui contient une valeur "init" ou des fonctions, ne peut définir un autre sous-state. Ces fonctions ont la signature suivante :

```

(sous-state, props, actions, init) => (...args) {}

```

Figure 16 : Signature d'une action traitée par withState

Elles prennent en argument le sous state dans lequel elle est déclaré, les props passées lors de l'instanciation du composant retourné par `withState`, la liste complète des fonctions déclarées dans la structure, ainsi qu'une fonction « `init` » qui renvoie la valeur initiale du sous-state. Cette fonction en renvoie une autre qui prend en argument la liste des arguments passés lors de l'appelle de la fonction.

Le state qui sera construit aura la forme ci-contre. Le champs « `list` » contiendra la liste de choses à faire, `filteringPredicate`, un filtre permettant de filtrer la liste, et « `title` », le titre d'une chose à faire.

```
{
  todos: {
    list: Map(...),
    filteringPredicate: list => list
  },
  creationForm: {
    title: ''
  }
}
```

Figure 17: Structure du state généré par `withState`

- Durant l'implémentation du composant, on s'aperçoit que l'on a besoin d'une fonction qui applique le filtre sur la liste et renvoie la nouvelle liste, ainsi que d'une fonction qui donne le nombre de choses qui restent à faire. Nous ne voulons pas stocker les valeurs renvoyées par ces fonctions car elles résultent d'un calcul effectué sur des valeurs du state. De plus on ne peut pas stocker ces fonctions dans le premier argument car le résultat des fonctions qui y sont présentes sont directement injectés dans le state du composant qui sera généré. Il n'est donc pas possible de recueillir leur valeur de retour. On choisit donc de passer un second argument qui contient des sélecteurs. Un sélecteur prendra en paramètre le state du futur composant, et les props passées au composant pour pouvoir renvoyer la valeur souhaitée.

La nouvelle structure du module `withState` est la même que précédemment mis à part que les deux arguments qui lui sont passés s'appellent maintenant « `states` » et « `selectors` ».

Dans le constructeur du composant généré par `withState`, il est d'abord nécessaire de déclarer une fonction « `setAction` » qui va permettre de déclarer une action de mise à jour du state. Cette fonction est la suivante :

```
const setAction = (index, substate, path, func) => {
  if (index in this.actions) {
    throw new Error(`there is already an action ${index}`)
  }
  const substateCopy = typeof substate === 'object' ? cloneDeep(substate) : substate
  this.actions[index] = (...args) => {
    const res = func(path.length === 0 ? this.state : get(this.state, path), props, this.actions, () => substateCopy)(...args)
    this.setState((prevState) => {
      const stateCopy = cloneDeep(prevState)
      return res !== undefined
        ? path.length === 0 ? {...prevState, ...res} : set(stateCopy, path, res)
        : prevState
    })
  }
}
```

Figure 18 : Implémentation de la fonction `setAction`

Elle prend en argument :

- L'index de la fonction, qui est aussi son nom. C'est donc une chaîne de caractères.
- Le sous state dans lequel elle est déclarée
- Le chemin permettant d'atteindre ce sous-state, sous forme d'un tableau. Ainsi, si la fonction qui est traitée est « `create` », ce tableau contiendra ['`todos`', '`list`']
- La fonction qui est traitée

D'abord il est testé si une action portant ce nom existe déjà, si c'est le cas, une erreur est générée. Ensuite, si le sous-state est un objet, une copie est créée qui sera nécessaire pour réinitialiser la

valeur du sous-state si besoin est. Une action est alors définie dans l'objet « this.actions » à l'indice portant son nom. Cette action peut être exécuté avec une liste de paramètres qui lui est passée en arguments lors de son appelle. Lorsqu'une action sera exécutée, sa valeur de retour sera ensuite utilisée pour mettre le sous-state à jour (si elle est définie).

```
const constructState = (obj, path) => {
  let substate = {}
  let pathCopy
  if (obj.init !== undefined) {
    substate = typeof obj.init === 'function' ? obj.init(props) : obj.init
  }
  forEach(pickBy(obj, val => typeof val !== 'function'), (value, key) => {
    pathCopy = path.slice()
    pathCopy.push(key)
    if (typeof value === 'object') {
      if (typeof substate !== 'undefined' && typeof substate !== 'object') {
        throw new Error('cannot provide an init value and substates')
      }
      substate[key] = constructState(value, pathCopy)
    } else if (typeof value === 'string' && key !== 'init') {
      const func = stateField => event => event.target.value
      substate[key] = ''
      setAction(value, substate, pathCopy, func)
    }
  })
  forEach(pickBy(obj, val => typeof val === 'function' && !path.includes('init')), (value, key) => {
    setAction(key, substate, path, value)
  })
  return substate
}
```

Figure 19: Implémentation de la fonction *constructState*

Il est ensuite nécessaire de construire le state. Celui-ci comprenant un nombre de niveaux non défini, le state va être construit de manière récursive. La fonction « *constructState* » prend en paramètres un objet « *obj* » représentant le sous-state, ainsi que le chemin pour atteindre celui-ci. Cette fonction commence par déclarer une variable « *substate* » qui contiendra la valeur de retour, et une variable « *pathCopy* » qui permettra de stocker une copie de l'argument « *path* ». On regarde ensuite si l'objet contient un champs « *init* », si c'est le cas, on initialise sa valeur, sinon on va parcourir ses champs. On va d'abord enlever toutes les fonctions de l'objet pour qu'il ne reste plus que les trois types de valeur suivant :

- Une valeurs d'initialisation. La valeur d'initialisation étant déjà traitée, elle sera ignorée si la valeur courante est de ce type.
- Un éventuel sous-state, dans ce cas, le sous-state est construit en rappelant la fonction « *constructState* » dessus.
- Un champ contenant une chaine de caractères comme le champs « *title* » dans la *todoList*. Ce type de champs sert à reduire la déclaration d'une chaine de caractères ayant une chaine vide comme valeur initiale. Un sous état sera donc créé portant le nom du champ courant et ayant pour valeur une chaine vide, ainsi qu'une action ayant pour nom le contenu de la chaine contenu dans le champs (ici « *updateTitle* ») permettant de modifier le *substate* à l'aide d'un évènement.

Nous allons devoir créer la liste des actions grâce la fonction « *setAction* » décrite plus haut. Pour ce faire, on parcourt l'objet dans lequel on a gardé uniquement les fonctions.

Il est essentiel de faire ces deux parcours dans cet ordre car le premier parcours permet de construire le state qui peut ensuite être passé comme valeur de réinitialisation aux différentes fonctions. Des actions sont définies lors du premier parcours mais celles-ci ne nécessitent pas d'avoir accès à la valeur initiale

Par la suite, les sélecteurs sont parcourus grâce à la fonction « setSelectors » suivante :

```
const setSelectors = selectors => mapValues(selectors, value => () => value(this.state, this.props))
```

Figure 20 : Implémentation de la fonction setSelectors

La fonction mapValues est issue de Lodash. Elle permet de parcourir un objet en modifiant ses champs. Ainsi, un selecteurs sera une fonction qui exécute le sélecteur déclaré dans l'argument « selectors » de withState.

Pour valider le bon fonctionnement du module, il a été nécessaire d'écrire des tests unitaires.

Tests

withState doit pouvoir supporter les fonctionnalités suivantes :

- Décorer un composant stateful en paramètre.
- Décorer un composant stateless en paramètre.
- Utiliser une fonction pour initialiser la valeur d'un sous-state
- Remettre la valeur d'un sous-state à sa valeur initiale
- Une action peut ne rien retourner
- L'initialisation d'une chaîne de caractères vide doit pouvoir se faire en lui donnant pour valeur le nom de la fonction qui pourra la modifier grâce à un événement
- Déclencher une erreur si plus d'une valeur sont déclarées dans un sous-state

Un test est écrit pour chacune de ces fonctionnalités. Les tests sont écrits avec la librairie Jest. Les fonctions fournies par Jest que nous utilisons sont les suivantes :

- « toEqual » qui permet de tester l'égalité de deux objets en profondeur.
- « toBe » qui permet de tester l'égalité entre deux valeurs.
- « fn » qui permet de mettre un espion sur une fonction. Nous l'utilisons pour vérifier les paramètres passés aux fonctions testées.

Tous les tests sont visibles en annexe V. Ceux-ci étant validés, le module « withState » remplit ses fonctions. Cependant, il reste toutefois couteux. En effet, puisqu'il parcourt quatre fois l'objet « states » passé en paramètre du module (pour retirer les fonctions, traiter l'objet sans fonction, supprimer les champs qui ne sont pas des fonctions, traiter l'objet contenant uniquement des fonctions). L'avantage de ce décorateur est qu'il permet de déclarer les fonctions et données nécessaires au fonctionnement d'un composant de manière organisée et, je pense, de manière facilement lisible.

Développement de Xen Orchestra

Présentation de Xen Orchestra

Xen Orchestra⁶ est un gestionnaire de machines virtuelles. Il permet d'administrer des serveurs tournant sous XenServer. XenServer est un logiciel développé par Citrix, qui utilise Xen qui est un hyperviseur de machines virtuelles. XenServer l'exécution de machines virtuelles. Pour pouvoir interroger un XenServer, une API⁷ du nom de Xapi est utilisée par Xen Orchestra.

Il est composé de trois grands modules :

- xo-server qui représente le cœur de l'application.
- xo-CLI qui est un client de xo-server en ligne de commandes
- xo-web qui est également un client de xo-server, cette fois grâce à une interface web

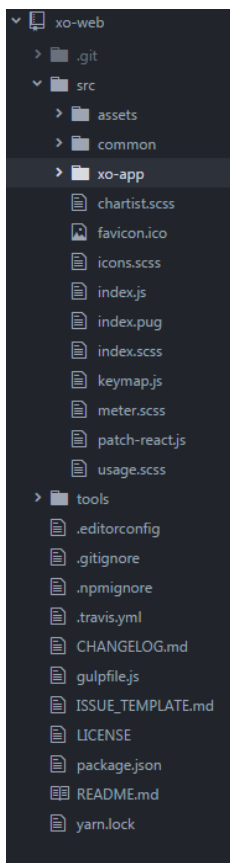


Figure 21 :
Architecture de xo-web

Xo-web, comporte un dossier « src » contenant les fichiers source de l'application. Il inclut le dossier « common » dans lequel sont présents une multitude de composants nécessaires au fonctionnement de l'interface graphique, comme des composants graphiques génériques (boutons, liste triées, barre de navigation...), une liste de sélecteurs ou encore une liste de requête pouvant être envoyées à xo-server. Il inclut également le dossier xo-app. Celui-ci comporte les composants permettant de construire l'interface graphique. Par exemple, Sur la vue détaillée d'un hôte présente en annexe VII, le menu à gauche est défini dans le composant Menu du fichier « menu/index.js », l'en-tête par le composant Host du fichier « host/index.js » et le centre de la page avec les graphiques par un composant TabGeneral du fichier « host/tab-general.js ». Chacun de ces composants peut être confectionné grâce à d'autres sous-composants situés dans d'autres sous-dossiers.

Xo-server contient un dossier src comportant un dossier « api » qui inclut les fichiers définissant les fonctions qui pourront être appelées par xo-web et appelleront des fonctions de Xapi., « collection » contient le fichier permettant la gestion de la base de données Redis, ainsi qu'une multitude d'autres dossiers que je n'ai pas eu l'occasion d'explorer.

Xen Orchestra est composé de différentes entités dont les principales sont :

- VM : Cet objet représente une machine virtuelle
- Host : Un hôte est une machine physique sur laquelle est installé XenServer. Elle sert à héberger l'exécution d'une VM
- Pool : Un pool est un amas d'appareils de stockage sur lesquels, les données nécessaire au fonctionnement de Xen Orchestra, vont

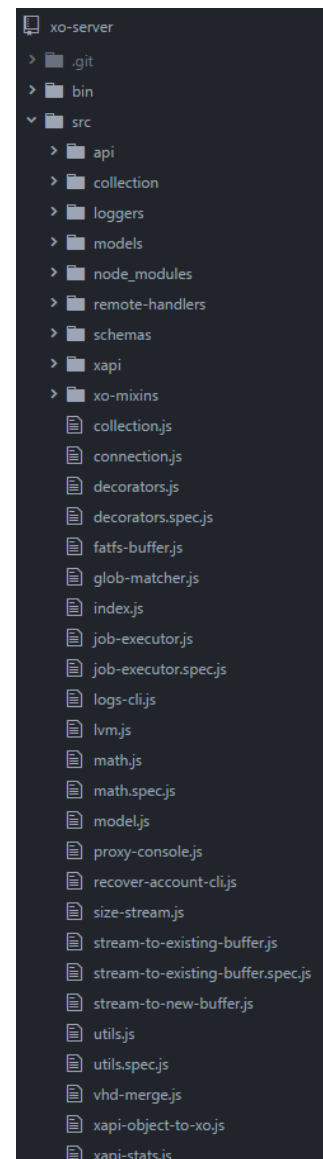


Figure 10 : Architecture de xo-server

⁶ Adresses des dépôts de Xen Orchestra: <https://github.com/vatesfr/xo-web> et <https://github.com/vatesfr/xo-server>

⁷ Une API (Application Programming Interface) est une interface de programmation par laquelle un logiciel offre des services.

être sauvées.

Dans Xen Orchestra, des décorateurs sont utilisés sur la plupart de des composants. Chacun d'eux s'utilise soit de la même manière que `withState`⁸ ou en plaçant le caractère « @ » devant et s'applique sur le composant défini sur la ligne suivante. Les principaux sont :

- `connectStore` : Ce module permet d'insérer dans les props du composant décoré, le résultat de sélecteurs. Il prend en paramètre une fonction à laquelle seront passés le state et les props de ce même composant et qui seront passées en argument du sélecteur. Dans l'exemple suivant, le composant `TabLogs` du dossier « `xo-app/vm` ». Une machine virtuelle est associée à des messages appelés « logs ». Le champ « log » d'une VM contient donc la liste des identifiants des messages qui la concerne. La fonction « `createGetObjectMessages` » permet de créer un sélecteur renvoyant, ici, les logs associés à la machine virtuelle reçu dans les props. Une fois décoré, le composant possèdera dans ses props un champ « logs » contenant le résultat de l'exécution du sélecteur « logs ».

```
@connectStore(() => {
  const logs = createGetObjectMessages(
    (_, props) => props.vm
  )

  return (state, props) => ({
    logs: logs(state, props)
  })
})
```

Figure 23 : Utilisation de `connectStore`

- `addSubscription` : Ce décorateur prend en argument un objet. Chacune de ces valeurs est une fonction qui exécute une requête sur `xo-server`. Il s'utilise donc de la manière ci-contre

Il va permettre de générer un champ du nom de la clé (ici `resourceSets`) dans les props du composant généré par le décorateur, qui sera mis à jour à intervalles de temps réguliers.

```
@addSubscriptions({
  resourceSets: subscribeResourceSets
})
```

Figure 24 : Utilisation de `addSubscriptions`

- `routes` : Le décorateur « routes » permet de déclarer des routes. Le but d'une route est d'afficher un contenu différent en fonction de l'URL⁹. Voici, ci-contre l'utilisation de ce décorateur dans le fichier « `xo-app/pool/index.js` » sur le composant « Pool ».

Il est défini dans un autre composant, via ce même décorateur, que, ce composant `Pool` est affiché lorsque l'adresse « `/pool/<poolId>` » est demandée. Ainsi, si l'adresse « `/pool/<poolId>/logs` » est demandée, le composant « Pool » intègrera le composant « `TabLogs` » dans son rendu. Le premier argument définit le composant affiché par défaut (ici donc sur à l'adresse « `/pool/<poolId>` »). Ce contenu qui diffère selon l'adresse est instancié grâce au composant « Page ».

```
@routes('general', {
  advanced: TabAdvanced,
  general: TabGeneral,
  logs: TabLogs,
  network: TabNetwork,
  patches: TabPatches,
  stats: TabStats
})
```

Figure 125 : Utilisation de routes

```
render () {
  return <Page header={this.header()} title={pool.name_label}>
    {cloneElement(this.props.children, childProps)}
  </Page>
}
```

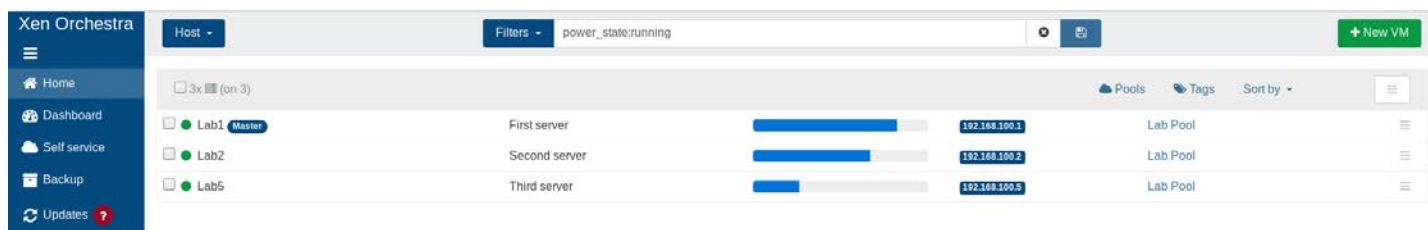
Figure 11 : Instanciation d'un composant transmis par routes

⁸ `Decorateur(args)(composant à décorer,`

⁹ Adresse web

Afficher une icône orange quand l'hôte est indisponible

Les vues de liste des hôtes et générale d'un hôte ressemblent à ceci :



Host	Role	Status	IP	Pool
Lab1	Master	Running	192.168.100.1	Lab Pool
Lab2	Second server	Running	192.168.100.2	Lab Pool
Lab5	Third server	Running	192.168.100.5	Lab Pool

Figure 27 : Affichage de la liste des hôtes

Ici, le point situé en début de chaque ligne sur la liste, ainsi que l'icône représentant un serveur sur la vue générale en haut à gauche (Annexe VIII), sont vertes car l'hôte est démarré. Elles pourraient être également rouges lorsque l'hôte est éteint. La tâche qui m'a été assignée consistait à afficher ces icônes en orange lorsque que l'hôte est démarré, mais qu'il est indisponible.

Un hôte est caractérisé par différents attributs dont « power_state » qui est une chaîne de caractères qui représente son état. Ses valeurs possibles sont « Halted » (éteinte) ou « Running » (allumé). Un hôte contient aussi un champ « enable » qui est un booléen qui définit s'il est disponible, ainsi qu'un champ « current_operations » qui est un tableau de chaînes de caractères représentant des opérations en cours d'exécution. On définit qu'un hôte est indisponible s'il est allumé, que son champ « enable » est à « false » et qu'aucune opération n'est en cours.

Concernant le point, celui-ci est défini par le composant HostItem qui se situe dans le fichier « src/xo-app/home/host-item.js ». Il définit une ligne représentant un hôte dans le tableau présent sur la capture d'écran ci-dessus. Sa fonction « render » contient le code suivant :

```
{isEmpty(host.current_operations)
  ? <Icon icon={` ${host.power_state.toLowerCase()} `} />
  : <Icon icon='busy' />
}
```

Figure 13 Condition d'affichage de l'icône représentant l'état d'un hôte

Le point est représenté par le composant Icon qui a une propriété « icon ». On passe à cette propriété l'état de l'hôte. En fonction de cette valeur, le composant va choisir un style qui va modifier la couleur du point. On peut voir que lorsqu'une opération est en cours, un icône de type « busy » est affiché. Un tel icône s'affiche bien en orange, mais son appellation ne correspondant pas à l'état d'un hôte indisponible, nous allons utiliser ce composant avec son attribut « icon » à une valeur « disabled ». C'est dans le fichier « icons.scss » situé dans le dossier xo-app que sont définis les différents styles qu'un icône peut prendre. On y définit donc un style disabled :

```
.xo-icon {
  &-disabled {
    @extend .fa;
    @extend .fa-circle;
    @extend .xo-status-busy;
  }
}
```

Figure 14 : règles de style s'appliquant sur les icônes de VMs indisponibles

Les règles de style définies se trouvent encapsulées dans une balise nommée « .xo-icon », ce qui signifie que, quand la page HTML sera générée ; elles s'appliqueront sur les éléments de classe « xo-icon-disabled ». Les règles de styles, ici, sont importées. Les deux premières viennent d'un package appelé « font-awesome » et la dernière est définie dans le fichier « index.scss » situé à la racine de l'application.

Il va ensuite falloir revoir la condition d'affichage de l'icône « disabled ». S'il y a des opérations en cours, ce cas ne change pas,

et l'icône « busy » est affiché. Sinon, dans le cas où l'hôte est démarré et qu'il n'est pas disponible, c'est l'icône « disabled », sinon, on affiche l'icône correspondant au champ « power_state ». Ce qui donne le code suivant :

```
{!isEmpty(host.current_operations)
? <Icon icon='busy' />
: (host.power_state === 'Running' && !host.enabled)
? <Icon icon='disabled' />
: <Icon icon={` ${host.power_state.toLowerCase()} `} />
}
```

Figure 15 : Condition d'affichage d'un icône "disabled"

Pour ce qui est de l'icône présent dans la vue générale de l'hôte, il définit dans le composant Host du fichier « src/xo-app/host/index.js ». Il est défini par une fonction « header » qui renvoie le rendu de l'entête de la page. Le composant qui doit être modifié est instancié dans cette fonction. Il est modifié avec la même condition que précédemment.

Demander une confirmation quand une mise à jour est faite depuis la vue d'un pool

La seconde tâche concernant Xen Orchestra qui m'a été confiée était d'afficher un message de mise en garde lorsqu'une mise à jour d'un hôte est installée depuis la vue de l'hôte. Il y a plusieurs manières de mettre à jour un hôte

- A partir de la vue d'un pool, une mise à jour globale des hôtes qu'il héberge. Dans ce cas un algorithme intelligent est exécuté et va choisir les mises à jour nécessaires.
- A partir de la vue d'un hôte, l'utilisateur peut choisir les mises à jour qu'il veut installer. Seulement cette méthode peut engendrer des conflits. C'est pourquoi il est nécessaire d'en avertir l'utilisateur.

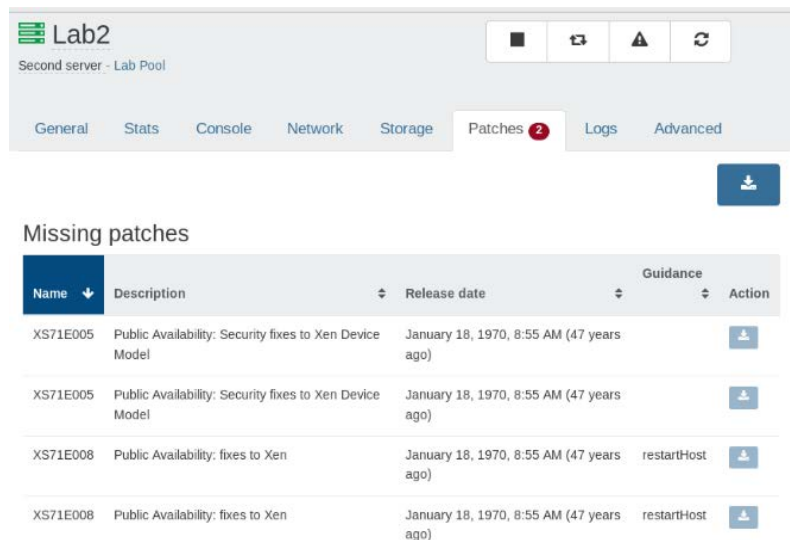


Figure 31 : Vue de mise à jour d'un hôte

C'est en appuyant sur un des boutons bleus à droite de la vue ci-contre que l'utilisateur peut installer les mises à jour. C'est à ce moment-là que devra apparaître le message. Pour l'afficher, il m'a été conseillé d'utiliser un composant de type Modal. Ce type de composant représente une fenêtre d'informations qui s'affiche au premier plan, avec au moins un bouton permettant de fermer la fenêtre. Dans un premier temps, la fenêtre devait contenir deux boutons : l'un pour effectuer l'installation, et l'autre pour l'annuler. Le message devait

également contenir un lien menant à la vue du pool qui héberge l'hôte, ce pour effectuer la mise à jour globale. Puis dans un second temps, ce lien est devenu un bouton intégré au message d'information, pour que l'utilisateur soit interpellé par celui-ci. Finalement il a été décidé que ce bouton n'avait pas sa place dans cette fenêtre. En effet la logique d'une telle fenêtre est que seulement deux actions soient possibles : l'adoption ou le rejet du message proposé. Dans cette

dernière solution, les labels des boutons devraient indiquer l'action qui sera effectuée si un clic est effectué dessus. Le bouton de validation permettra donc d'installer la mise à jour, et le bouton d'annulation redirigera l'utilisateur sur la page de mise à jour du pool.

Le fichier modal.js situé dans src/common définit deux composants, Confirm et Alert, de type Modal qui utilisent le patron donné par React-Bootstrap¹⁰. Alert affiche une fenêtre d'information avec un seul bouton pour la fermer. Celui qui nous intéresse est donc Confirm car il génère une fenêtre de confirmation avec deux choix possibles. Il reçoit en attribut le contenu du message, son titre ainsi que les fonctions qui seront exécutées lorsque la proposition sera acceptée ou refusée. Une fonction « confirm » est également définie dans ce fichier. Cette fonction permet d'instancier le composant « Confirm ». Elle prend en paramètre le contenu du message, son titre ainsi qu'une chaîne de caractères représentant une icône. Elle crée et retourne une promesse¹¹. L'exécuteur de cette promesse prend en arguments les fonctions d'installation du patch, et de redirection vers les mises à jour du pool concerné. La fonction « modal » permet d'afficher la fenêtre avec l'instance du composant « Confirm » comme contenu.

On se rend compte que le composant Confirm ne permet pas de modifier l'intitulé de ses boutons. Il a donc été modifié ainsi que la fonction qui lui est associée pour rendre ceci possible, en leur rajoutant les labels en paramètre (voir image ci-contre).

La mise à jour d'un hôte s'effectue dans le fichier « src/xo-app/host/tab-patches.js ». Lorsque un clic est effectué sur un bouton de mise à jour d'un patch, une fonction est exécutée. Elle va être remplacée par la fonction suivante :

```
export const confirm = ({
  body,
  title,
  okLabel,
  cancelLabel,
  icon = 'alarm'
}) => {
  return new Promise((resolve, reject) => {
    modal(
      <Confirm
        title={title}
        resolve={resolve}
        reject={reject}
        icon={icon}
        okLabel={okLabel}
        cancelLabel={cancelLabel}
      >
        {body}
      </Confirm>,
      reject
    )
  })
}
```

Figure 32 : Fonction confirm

```
_installPatchWarning = (patch, installPatch) => confirm({
  title: _('installPatchWarningTitle'),
  body: <p>{_('installPatchWarningContent')}</p>,
  okLabel: _('installPatchWarningResolve'),
  cancelLabel: _('installPatchWarningReject')
}).then(
  () => installPatch(patch),
  () => this.context.router.push(`/pools/${this.props.host.$pool}/patches`)
)
```

Figure 16 : Fonction d'installation d'un patch

Elle prend en argument le patch qui doit être installé et une fonction permettant de l'installer. Elle va exécuter la fonction « confirm » en lui donnant le contenu nécessaire et retourner son résultat qui est la promesse que nous avons vue précédemment. L'exécuteur de la promesse est déclenchée lors de l'appel à « then », qui lui passe les fonctions de rejet ou d'adoption. Ainsi, si l'utilisateur veut poursuivre l'installation, la première fonction qui lui est passée en argument sera exécutée, et s'il

¹⁰ React-Bootstrap est une librairie permettant d'utiliser le framework css Bootstrap dans un projet React à travers des composants React.

¹¹ Une promesse est un objet Javascript qui est utilisé pour réaliser des traitements de façon asynchrone. Il prend un exécuteur en paramètre. Cet exécuteur est une fonction qui prend en arguments deux fonctions qui seront exécutées, l'une si la promesse est résolue, l'autre si elle est rejetée.

décline, la seconde sera appelée. Voici le rendu final de la demande de confirmation.

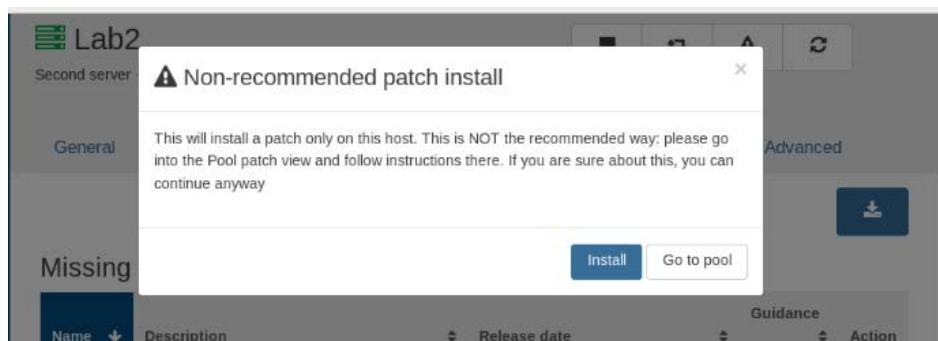


Figure 34 : Affichage d'une fenêtre de confirmation

Afficher un message rappelant les fichiers de sauvegarde compatibles

Dans Xen Orchestra, il est possible de faire des sauvegardes des machines virtuelles sous différents types :

- full backup
- rolling snapshot
- delta backup
- disaster recovery
- continuous replication
- file level restore

De plus il est possible de stocker ces sauvegardes de manière local, ou distante via les protocoles SMB¹² ou NFS¹³.

Xen Orchestra permet de restaurer des machines uniquement grâce à des fichiers de type delta backup hébergés en local ou via NFS. Cette tâche qui m'a été confiée avait pour but d'en informer l'utilisateur grâce à un message d'information.

Le formulaire permettant de restaurer une machine se trouve dans le rendu du composant FileRestore. Celui-ci contenant déjà un message d'information, une liste a été créée incluant les deux messages d'information, de la manière suivante :

```
<ul>
  <li><em><Icon icon='info' /> {_('restoreBackupsInfo')}</em></li>
  <li><em><Icon icon='info' /> {_('restoreDeltaBackupsInfo')}</em></li>
</ul>
```

Figure 17: Structure des messages d'informations concernant la restauration par fichier

Ici, la fonction « _ » fait référence à un module d'internationalisation de l'application. Ainsi pour chaque label comme « restoreBackupsInfo » ou « restoreDeltaBackupsInfo », un texte correspond pour chacune des langues proposées par Xen Orchestra. Il a donc fallu ajouter cette correspondance dans le fichier « message.js » situé dans « src/common/intl ». L'interface définitive du formulaire est donc la suivante :

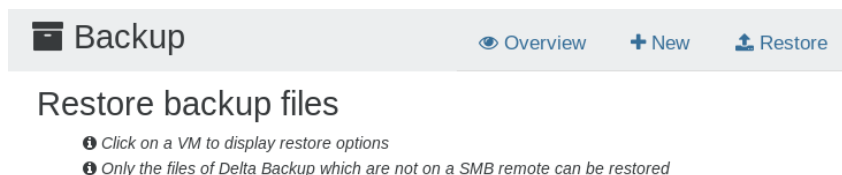


Figure 36 : Affichage des messages d'informations concernant la restauration par fichier

¹² Server Message Block

¹³ Network File System

Afficher depuis combien de temps une machine est éteinte

Sur la vue générale d'une machine virtuelle, si celle-ci est allumée, il est affiché depuis combien de temps elle a été démarrée.

Cette tâche a pour but d'afficher depuis combien de temps une machine est éteinte. Une machine virtuelle ne possède pas de champ gardant en mémoire la dernière date d'extinction. Cependant, lorsqu'une opération sur une machine est effectuée, comme l'allumage, l'extinction ou la création d'un snapshot, un message appelé « log » associé à la machine est généré. L'objectif

était donc de rechercher le dernier message de type « VM_SHUTDOWN » associé à la machine virtuelle voulue. Il a donc été nécessaire d'écrire un sélecteur pour obtenir cet objet. Si une machine virtuelle n'a jamais été éteinte, ou alors que ses logs ont été effacés, il peut arriver que ce message n'existe pas, dans ce cas il sera juste indiqué que la machine est éteinte. Tous les sélecteurs utilisés par Xen Orchestra sont centralisés dans le fichier « selectors.js » du dossier « common ». La fonction suivante a donc été implémentée dans ce fichier.

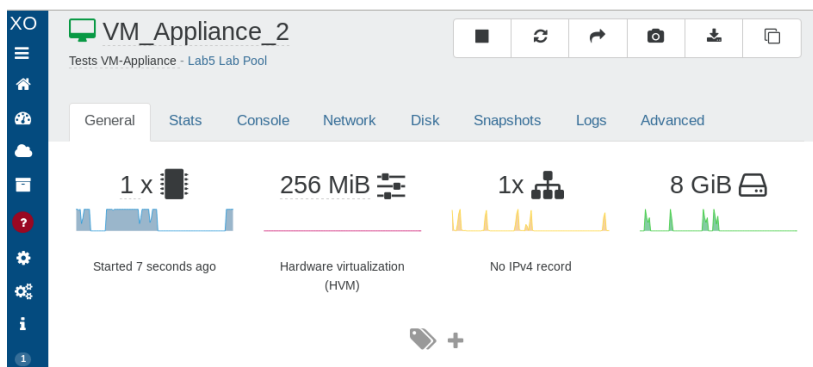


Figure 18 : Vue générale d'une machine virtuelle

```
export const createGetVmLastShutdownTime = (getVmId = (_, {vm}) => vm != null ? vm.id : undefined) => create(
  getVmId,
  createGetObjectsOfType('message'),
  (vmId, messages) => {
    let max = null
    forEach(messages, message => {
      if (
        message.$object === vmId &&
        message.name === 'VM_SHUTDOWN' &&
        (max === null || message.time > max)
      ) {
        max = message.time
      }
    })
    return max
  }
)
```

Figure 19 Implémentation du créateur de sélecteur createGetVmLastShutdown

Cette fonction renvoie un sélecteur, c'est-à-dire qu'elle fournit une fonction qui, elle, renverra la valeur souhaitée. Elle prend en argument une fonction qui renvoie l'identifiant de la machine virtuelle courante et renvoie le résultat d'une fonction « create ». Cette fonction « create » est issue du module « Reselect ». Elle reçoit en argument des fonctions appelées « input selectors » ainsi que le sélecteur qui est le dernier argument. Chacun des inputs selectors va être exécuté et va passer son résultat en argument du sélecteur. « create » va exécuter le sélecteur et créer une fonction qui renvoie son résultat. Ainsi, le sélecteur recevra l'identifiant de la machine virtuelle et la liste de tous les messages de l'application et renverra le message voulu.

Le résultat du sélecteur est injecté dans le composant TabGeneral du dossier « src/xo-app/vm » à l'aide du décorateur connectStore.

La donnée voulue étant disponible dans TabGeneral, il est nécessaire de l’afficher dans le rendu du composant.

```
{vm.power_state === 'Running'  
  ? <div>  
    <p className='text-xs-center'>{__('started', { ago: <FormattedRelative value={vm.startTime * 1000} /> })}</p>  
  </div>  
  : <p className='text-xs-center'>  
    { lastShutdownTime  
      ? __('vmNotRunningHaltedFor', {ago: <FormattedRelative value={lastShutdownTime * 1000} />})  
      : __('vmNotRunning')  
    }  
  </p>
```

Figure 20 : condition d’affichage de la valeur “lastShutdownTime”

Ainsi, si la machine virtuelle est éteinte et que la valeur « lastShutdownTime » est définie, on affiche la valeur formatée par un composant « FormatedRelative » et englobée dans message défini dans le module d’internationalisation. Si elle n’est pas définie, le message d’origine est affiché.

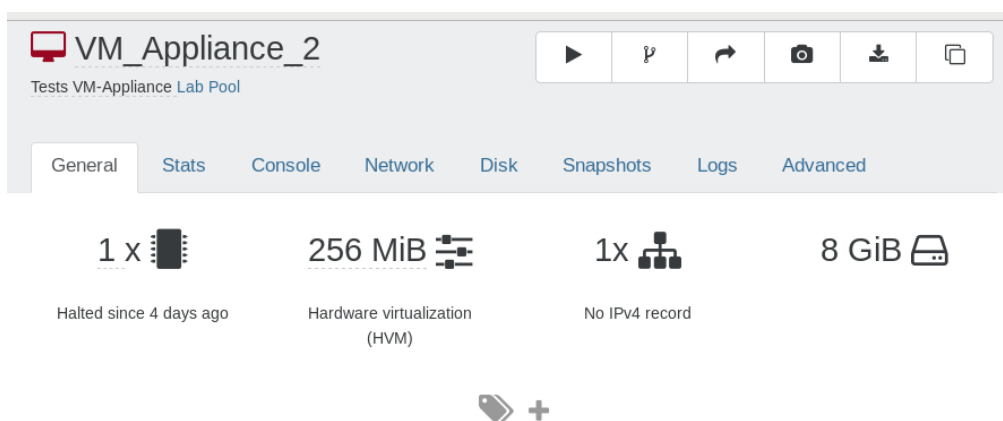


Figure 21 : Affichage du temps d’extinction d’une VM

Rendre possible la création de groupes de machines virtuelles

Cette dernière tâche a été celle qui m’a demandé le plus de temps. En effet, son but était d’implémenter une nouvelle fonctionnalité de Xen Orchestra : permettre la création de groupe de machines virtuelles, appelées VM_Appliance par Xapi. L’intérêt de tels groupes est de pouvoir démarrer ou arrêter les machines du groupe en même temps, ou dans un ordre prédéfini. Xapi définit ces fonctionnalités, elles doivent donc être testées dans un premier temps pour comprendre leur fonctionnement pour ensuite les intégrer à Xen Orchestra. Une VM_Appliance est caractérisée par un nom et une description.

Test des fonctionnalités via xe

Xe est une interface en ligne de commandes permettant l’utilisation de Xapi dans un environnement XenServer. J’ai donc pu tester les commandes permettant la gestion d’une VM_Appliance grâce à une connexion SSH sur un XenServer. A résulter de cette étape le document présent en annexe VI qui décrit chacune de ces commandes.

On peut voir dans ce document que les fonctions d’ajout, de suppression d’une machine virtuelle, et de gestions de l’ordre de démarrage et d’arrêt des machines font intervenir l’argument « vm- param-set ». Ce sont donc des fonctionnalités de gestion d’une machine virtuelle et non d’un groupe de machines.

Implémentation dans xo-server

Comme expliqué précédemment xo-server est le cœur de Xen Orchestra. Pour utiliser les objets gérés par XenServer dans un client Xen Orchestra, il est nécessaire de les exposer dans xo-server. Pour ce faire, xo-server va traiter des demandes envoyées par le client, et va interagir avec un XenServer via Xapi. Pour assurer le dialogue entre xo-server et le client, une API est créée dans le dossier « src/api » de xo-server. Ce dossier contient un fichier Javascript par entité. Un fichier vm-groupe.js¹⁴ y a donc été créé.

```
export async function start ({ vmGroup }) {
  await this.getXapi(vmGroup).call('VM_appliance.start', vmGroup._xapiRef, false)
}

export async function shutdown ({ vmGroup }) {
  await this.getXapi(vmGroup).call('VM_appliance.shutdown', vmGroup._xapiRef)
}

export async function set ({ vmGroup, name_label: nameLabel, name_description: nameDescription }) {
  if (nameDescription) {
    await this.getXapi(vmGroup).call('VM_appliance.set_name_description', vmGroup._xapiRef, nameDescription)
  }
  if (nameLabel) {
    await this.getXapi(vmGroup).call('VM_appliance.set_name_label', vmGroup._xapiRef, nameLabel)
  }
}

export async function destroy ({vmGroup}) {
  await this.getXapi(vmGroup).call('VM_appliance.destroy', vmGroup._xapiRef)
}

export async function create ({pool, name_label, name_description}) {
  await this.getXapi(pool).call('VM_appliance.create', {name_label, name_description})
}
```

Figure 22: Fonction d'appel à Xapi

Dans ce fichier sont définies les fonctions ci-dessus. Elles utilisent Xapi grâce à la méthode « call ». On peut voir que les fonctions de gestion de VMs¹⁵ au sein du groupe ne sont pas définies ici. En effet, elle utilise les champs d'une machine virtuelle, « order » qui donne une priorité au démarrage de la machine, et « appliance » qui définit l'appartenance de la machine à un groupe. En effet, ces attributs sont paramétrés grâce à la fonction « set » définies dans le fichier « vm.coffee ». Chacune

```
start.params = {
  id: { type: 'string' }
}

start.resolve = {
  vmGroup: ['id', 'VmGroup', 'operate']
}
```

Figure 232: paramètres de la fonction start

de ces fonctions est également caractérisée par la liste de ces paramètres, ainsi qu'un objet « resolve » qui a pour valeur un objet incluant des tableau, contenant le nom d'un paramètre passé à la fonction, le type de l'objet qui va être rechercher, ainsi que le niveau de droit que doit avoir l'utilisateur pour utiliser la fonction. La clé de cette

valeur est le nom d'un nouveau paramètre.

Lors de l'appelle à la fonction « start », un identifiant de type string lui sera passé en paramètre. Elle recevra en argument un objet de type « VmGroup » du nom de « vmGroup ».

Xo-server récupère des objets grâce à Xapi. Pour que ces objets puissent être personnalisés, il est nécessaire de les exposer. Ceci est le but des fonctions écrites dans le fichier « xapi-object-to-xo.js ». La fonction « vm_appliance » est donc ajoutée à ce fichier.

```
vm_appliance (obj) {
  return {
    type: 'VmGroup',
    name_description: obj.name_description,
    name_label: obj.name_label,
    allowed_operations: obj.allowed_operations,
    $VMs: link(obj, 'VMs'),
    current_operations: obj.current_operations
  }
},
```

Figure 24: Exposition d'un vmGroup

¹⁴ Le nom vm-appliance ne correspondant pas vraiment à la fonctionnalité qu'il représente, il a été renommé par vm-group dans Xen Orchestra.

¹⁵ Machine virtuelle

Elle prend un objet issue de Xapi en argument, et renvoie un objet représentant un groupe de machines virtuelles.

Permettre la gestion des VmGroupes dans xo-web

Connexion avec xo-server

Pour gérer les groupes dans le client web, il a donc été indispensable de créer des fonctions de dialogue avec xo-server. Toutes ces fonctions sont écrites dans le fichier « common/xo/index.js ».

```
export const startVmGroup = vmGroup => {
  _call('vmGroup.start', { id: resolveId(vmGroup) })
}
export const shutdownVmGroup = vmGroup => _call('vmGroup.shutdown', { id: resolveId(vmGroup) })
export const rebootVmGroup = async vmGroup => {
  await shutdownVmGroup(vmGroup)
  await startVmGroup(vmGroup)
}
export const editVmGroup = (vmGroup, props) => _call('vmGroup.set', { id: resolveId(vmGroup), ...props })
export const deleteVmGroup = (vmGroup, vms) =>
  confirm({
    title: _('deleteVmGroupModalTitle'),
    body: _('deleteVmGroupModalMessage')
  }).then(() => {
    forEach(vms, vm => editVm(vm, { appliance: null, order: 0 }))
    _call('vmGroup.destroy', { id: resolveId(vmGroup) })
  }, Promise.reject())
export const createVmGroup = ({ pool, name_label, name_description }) =>
  _call('vmGroup.create', { id: resolveId(pool), name_label, name_description })
```

Figure 25: Fonctions d'appel à Xapi

Affichage d'une liste de VmGroupes

Ensuite, le but a été d'afficher une liste de groupes sur le même modèle que la liste d'hôte située sur la figure 27.

Le composant permettant d'afficher une telle liste est Home situé dans le fichier « index.js » du dossier « xo-app/home » quel que soit le type des objets à afficher. Pour définir ce type, un paramètre « t » est passé au composant par l'URL. Il récupère la liste des objets à afficher grâce à un sélecteur « getObjectByType » et à ce paramètre « t ». Le résultat est inséré dans les props grâce au décorateur « connectStore ». Home utilise un objet « OPTIONS » qui définit les types des différents composants qu'il permet d'afficher.

```
'VmGroup': {
  defaultFilter: '',
  filters: homeFilters.vmGroup,
  mainActions: [
    { handler: shutdownVmGroups, icon: 'vm-stop', tooltip: _('stopVmLabel') },
    { handler: startVmGroups, icon: 'vm-start', tooltip: _('startVmLabel') },
    { handler: rebootVmGroups, icon: 'vm-reboot', tooltip: _('rebootVmLabel') }
  ],
  Item: VmGroupItem,
  sortOptions: [
    { labelId: 'homeSortByName', sortBy: 'name_label', sortOrder: 'asc' },
    { labelId: 'homeSortByPowerstate', sortBy: 'power_state', sortOrder: 'desc' }
  ]
},
```

Figure 26: Objet définissant différentes options d'affichage d'une liste de groupes

Le type VmGroup est donc ajouté à cet objet. La clé représente donc le type, et la valeur est un objet définissant, dans l'ordre, le filtre appliqué à la liste par défaut, les filtres disponibles pour ce type, importés depuis un autre fichier, les actions qui pourront être exécutées à partir de la liste, le composant qui va permettre d'afficher un

vmGroup dans la liste, ainsi que des paramètres pour modifier l'ordre de tri de la liste.

Le composant `VmGroupItem` est implémenté dans le fichier « `vm-group-item.js` » du dossier « `home` ». Chaque groupe est affiché sous forme d'une ligne dans la liste, comportant une case à



Figure 27 : Affichage d'une liste de groupes

cocher, une icône représentant l'état du groupe (allumé ou éteint) et le nom et la description du groupe. Il permet également de modifier ces deux dernières propriétés, ainsi que d'afficher des boutons correspondant aux actions précédemment définies lorsqu'une case est cochée, ainsi que d'afficher la vue détaillée du groupe en cliquant sur la ligne.

Implémentation de la vue détaillée d'un groupe de machines virtuelles

La vue détaillée d'un `VmGroup` devait se faire en utilisant le même modèle que celle des autres composants (VM, hôte ou pool). Cette vue devait contenir un en-tête avec le nom du groupe, sa description, une icône de visualisation de l'état du groupe, des boutons permettant d'effectuer des actions sur celui-ci ainsi que des onglets permettant d'afficher différents contenus. Ils sont les suivants :

- « `General` » qui montre le nombre de machines contenues dans le groupe, le nombre de processeurs utilisés par ces machines, la mémoire vive totale, ainsi que l'espace utilisé sur les disques.
- L'onglet « `Stats` » permettant d'afficher des graphiques montrant le pourcentage total d'utilisation des processeurs, la RAM total utilisée, ainsi que des graphiques montrant des échanges de données sur le réseau de la machine et sur son disque dur, en temps réel. De plus, cette vue devra inclure un bouton permettant d'afficher les statistiques cumulées.
- Un onglet « `Management` » permettant d'afficher la liste des machines virtuelles contenues dans le groupe, ainsi que d'ajouter ou supprimer des machines, et gérer l'ordre de démarrage et d'extinction des VMs.
- « `Advanced` » contiendra les informations relatives au groupe. Dans premier temps, seul l'identifiant du groupe sera affiché. Un bouton permettant de supprimer le groupe sera aussi disponible dans cette vue.

Pour ce faire, un dossier « `vm-groupe` » est créé dans « `xo-app` ». Ce dossier contient un fichier « `index.js` » déclarant un composant « `VmGroup` » permettant d'afficher l'entête, et qui instancie le composant nécessaire correspondant à l'onglet choisi. Ainsi, pour chacun des onglets, un fichier détermine son contenu.

Il est défini dans le composant « `XoApp` » que « `VmGroup` » sera affiché à l'adresse « `/vm-group/<VmGroupId>` ». Ce dernier définit les différentes routes, ci-contre, pour afficher les différents contenus des différents onglets en fonction de l'adresse. De plus, il nécessite un accès à l'objet que nous voulons afficher. Il est rendu disponible avec le décorateur « `connectStore` » qui injecte une propriété « `vmGroup` », ce grâce au selecteur « `createGetObject` » qui renvoie l'objet correspondant à l'identifiant passé dans l'URL. Comme le

```
@routes('general', {
  advanced: TabAdvanced,
  general: TabGeneral,
  management: TabManagement,
  stats: TabStats
})
@connectStore(() => {
  const getVmGroup = createGetObject()

  return (state, props) => {
    const vmGroup = getVmGroup(state, props)
    if (!vmGroup) {
      return {}
    }
    const vms = {}
    forEach(vmGroup.$VMS, vmId => {
      const getVM = createGetObject(() => vmId)
      vms[vmId] = getVM(state, props)
    })
    return {
      vmGroup,
      vms
    }
  }
})
```

Figure 28: `connectStore` appliqué au composant `VmGroup`

VmGroup récupéré précédemment ne contient que les identifiants de ces machines dans son champs « \$VMs », la liste des machines va être générée et intégrée au composant grâce à ce même décorateur. Ce composant n'exige pas un accès à cette liste, mais la totalité des onglets l'utilise, ainsi, pour éviter la redondance de code, l'objet contenant les VMs est défini dans ce composant, et sera passé à l'instanciation de l'onglet.

```
<TabButton
  btnStyle='danger'
  handler={() => this._deleteVmGroup(vmGroup, vms)}
  icon='vm-delete'
  labelId='vmRemoveButton'
/>
```

Figure 48: Instanciation du bouton de suppression d'un groupe

Le fichier « tab-advanced » définit l'affichage de l'onglet « Advanced ». Pour la suppression du groupe, il utilise un bouton de type « TabButton ». Il prend en argument un style qui va ici, l'afficher en rouge, une icône qui sera affichée sur le bouton, un label faisant référence à une chaîne de caractère dans le

module d'internationalisation, ainsi qu'une fonction qui sera exécutée lorsqu'un clic sera effectué sur le bouton. Cette fonction va appeler la fonction deleteVmGroup (figure 44) qui va demander une confirmation à l'aide du composant « Confirm ». Si la promesse est acceptée, l'utilisateur sera redirigé sur la liste des VmGroups et le groupe sera supprimé, sinon, rien ne se passera, car la fonction « noop » est une fonction n'exécutant aucune instruction. L'identifiant du groupe est affiché dans une table HTML utilisant les balises « table », « tbody », « tr », et « th ».

```
_deleteVmGroup = (vmGroup, vms) => {
  deleteVmGroup(vmGroup, vms).then(
    () => this.context.router.push('home?s=&t=VmGroup'),
    noop
  )
}
```

Figure 49 : Fonction de suppression d'un VmGroup

C'est le composant « TabGeneral » qui définit le rendu de l'onglet « General ». Pour afficher le nombre de machines contenues dans le groupe, on utilise une fonction « size » issue de la librairie Lodash sur la variable « vms » transmis par le composant « VmGroup ». « vms » est un objet associant à chaque identifiant de machine, l'objet représentant la machine. Les objets en Javascript ne possédant pas de fonction renvoyant la taille de l'objet, il est donc nécessaire d'utiliser une fonction tierce.

Concernant les autres critères à afficher dans cette page, leur valeur est injectée dans les props du composant via le décorateur « connectStore ». Chaque VM possède un champ « CPUs » contenant un champs « number » qui est le nombre de processeurs utilisés par la machine virtuelle. Une somme est alors effectuée sur ce champ de chacune des machines virtuelles. Une VM inclut également la taille de la mémoire vive qu'elle peut utiliser dans le champ « memory.dynamic ». La valeur contenue à cet emplacement est un tableau incluant deux valeurs : les mémoires minimums et maximum pouvant être utilisées par la machine. Une somme est effectuée sur la seconde valeur qui est la mémoire maximum dont peut avoir besoin la VM. Ces valeurs étant exprimées en octet, il est nécessaire de les convertir dans une unité qui aura plus de sens. Ceci est fait grâce à une fonction « formatSize » qui utilise un module « humanFormat » qui va convertir la valeur dans l'unité voulue, et lui associe un préfixe. Ici nous récupérons donc des valeurs en gigabits.

La taille de la mémoire statique se calcule différemment. En effet une machine virtuelle est associée à un ou plusieurs disques virtuels appelés VDIs¹⁶ et ce via un objet VBD¹⁷. Chaque VDI peut être associé à plusieurs VBDs, et donc plusieurs machines virtuelles. Ainsi, ce que nous souhaitons calculer est la somme des tailles de chacun des disques utilisés par les Vms du groupe. Mais, si plusieurs machines utilisent le même disque virtuel, celui-ci ne doit être compté qu'une seule fois.

¹⁶ virtual disk image

¹⁷ virtual block device

Un ensemble (sans doublons) contenant les identifiants des VDIs utilisés par les machines du groupe est donc créé. C'est grâce à cette liste d'identifiant que l'on calcule la mémoire statique totale utilisée par le groupe. La vue obtenue est la suivante :



Figure 29 : Vue générale d'un groupe de machines virtuelles

Le rendu du composant « TabManagement » défini dans le fichier « tab-management » montre, à son initialisation deux boutons et la liste des machines virtuelles. La liste est affichée grâce à un composant « SortedList ». Cette liste prend en deux arguments : la liste des objets à afficher qui est ici la variable « vms » qui a été passé en argument à « TabManagement » depuis le composant « VmGroup », et un argument « columns » qui définit les colonnes à afficher dans la table. Ici, on passe un tableau « VM_COLUMNS » qui est un tableau d'objets. Chaque objet contient un champ « name » qui contient le nom de la colonne, un champ « itemRenderer » qui est une fonction qui prend en paramètre un objet de la liste et retourne son affichage dans la case du tableau, ainsi

```
<SortedList collection={vms} columns={VM_COLUMNS} />
```

Figure 30: Instanciation du composant SortedTable

qu'un champ « sortCriteria » qui est une fonction qui prend également un objet à afficher et retourne un critère de trie dans la liste. Ce tableau contient donc une colonne affichant le nom des machines virtuelles, une, montrant sa description, et une dernière affichant un bouton permettant de supprimer la VM du groupe. On obtient donc le rendu ci-contre :

Label	Description	Actions
VM_Appliance_1	Created by XO	
VM_Appliance_3	Created by XO	

« < 1 > » 2 / 2 🔍

Figure 31: Affichage de la liste des VMs du groupe

Concernant les deux boutons bleus situés au-dessus du tableau, ils permettent de montrer les composants d'ajout de machines au groupe, et de gestion de l'ordre de démarrage. Ainsi, une valeur « attachVm » associée au bouton « New VM » dans le state de « TabManagement » est initialisée à « false ». Lorsqu'un clic est effectué sur ce bouton, sa valeur associée sera changée par son opposé. C'est quand cette valeur est vraie que le composant permettant d'ajouter des VMs au groupe sera affiché. Il en est de même avec le bouton « Boot » order qui est associé à un booléen « bootOrder » et permettra la gestion de l'ordre de démarrage.

L'ajout de machines virtuelles se fait avec un formulaire contenant un champ autorisant la sélection d'une ou plusieurs VMs, et ce grâce à un composant « SelectVm » qui prend en argument une fonction « onChange » qui sera exécutée quand la valeur du champ sera changée. Ici, elle est associée à un champ « vmsToAdd » du state, qui contient la liste des machines sélectionnées. Il prend également en argument un prédicat qui nous permet de mettre à disposition, une liste contenant les machines voulues. En effet, une VM ne peut être associée qu'à un seul groupe. Ainsi on définit un prédicat qui renvoie vrai si le champs « appliance » de la machine est « null », et faux sinon. La liste affiche donc uniquement les machines ne faisant pas parti d'un groupe.

Dans l'onglet « Disk » de la vue d'une machine virtuelle situé dans le fichier « xo-app/vm/tab-disk.js », un composant permet la gestion de l'ordre de démarrage parmi différents composants de la machine. Ce composant graphique de gestion d'ordre correspond exactement à ce que nous voulons faire. Cependant, ce composant « BootOrder » est déclaré dans le fichier tab-disk et n'est pas générique. Il sera donc copié dans le fichier « tab-management », et adapté à l'utilisation voulue. Etant conscient que la duplication de code entraîne des problèmes de maintien de celui-ci, il pourra être envisagé par la suite de créer un composant générique satisfaisant ces utilisations. Ceci permet également d'identifier les fonctionnalités à implémenter dans le futur composant générique.

Ce composant est affiché de la manière suivante :

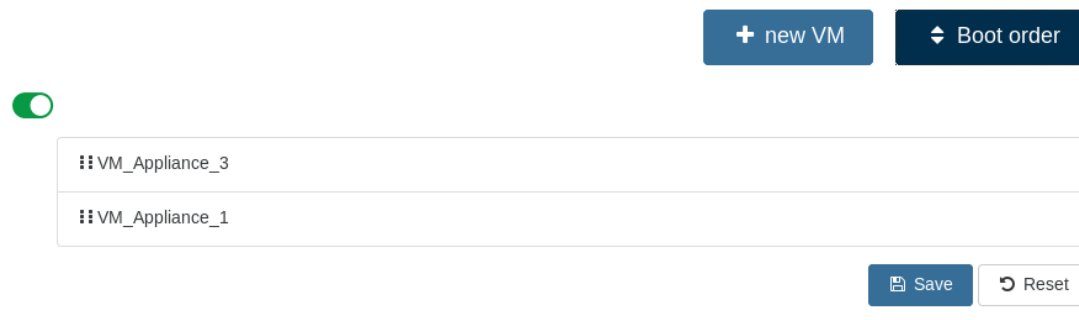


Figure 54 : Affichage du composant de gestion de l'ordre de démarrage

Dans cette interface, le bouton en haut à gauche permet d'activer ou non la gestion de l'ordre de démarrage des machines. La gestion de l'ordre se fait à partir de la liste en dessous, en drag n drop¹⁸. Le bouton « Save » permet de sauvegarder la configuration, et le bouton « Reset » permet de restituer la dernière configuration sauvee. Le paramètre qui définit la priorité de démarrage d'une machine virtuelle est son champ « order ». Plus il est bas, plus la priorité est élevée.

Ce composant BootOrder utilise une fonction parseBootOrder qui parcourt la liste des VMs reçu en attribut du composant TabManagement et établie un ordre initial. Si toutes les valeurs « order » des machines virtuelles sont identiques, aucun ordre ne peut être déterminé, il sera donc désactivé.

Une fonction setVmBootOrder permet d'enregistrer la configuration. Si l'interrupteur est activé, elle va donner à la première machine une valeur 0 à son champ « order », puis 1 à la seconde, etc... et si l'interrupteur est désactivé, toutes les machines auront une valeur 0 dans leur champs « order ».

La vue d'une machine virtuelle permet déjà d'afficher des statistiques. Ce seront ces mêmes données qui seront affichées dans l'onglet « Stats », superposées pour chacune des machines incluent dans le groupe. De plus, les statistiques consternant un pool affichent également des données de plusieurs appareils. Il est donc nécessaire de faire un mélange entre ces deux méthodes pour obtenir les graphiques voulus. Les composants effectuant ces traitements se trouvent dans le fichier « common/xo-line-chart/index.js ». Le rendu graphique de ces données est donné grâce à un composant CharistGraph qui reçoit une liste d'options, ainsi qu'un tableau dans un champ « data », contenant un autre tableau pour chaque série à afficher. Pour chaque graphique, un composant est créé dans ce même fichier. On déclare donc les composants VmGroupCpuLineChart, VmGroupMemoryLineChart, VmGroupXvdLineChart, VmGroupVifLineChart. Chacun d'eux reçoit en argument une liste de données « stats » transmises par xo-server et un booléen « addSumSeries » qui définit si on affiche les statistiques cumulées et d'éventuelles options. Si on

```
const series = map(data, ({ vm, stats }) => ({
  name: vm,
  data: computeArraysSum(stats.cpus)
}))
```

prend l'exemple du composant affichant les statistiques relatives à l'utilisation des processeurs de chaque machine, ces VMs peuvent utiliser plusieurs processeurs, l'objet « stats » recevra donc dans son

Figure 55 : Calcul des différentes séries de statistiques

¹⁸ Glisser, déposer

champ «cpus» un tableau pouvant inclure plusieurs tableaux de données de taille égale. L’affichage de l’utilisation de ces processeurs va être représenté par une somme sur chacun d’eux à chaque instant. Ceci est le but de la fonction «computeArraysSum». De plus si les statistiques cumulées sont demandées, une autre liste de données est ajoutée au tableau «series» faisant une seconde somme à chaque instant sur chacune de ces collections. Il est ensuite transmis au composant ChartistGraph. Le rendu est le suivant :



Figure 56: Statistiques d'un groupe de VMs

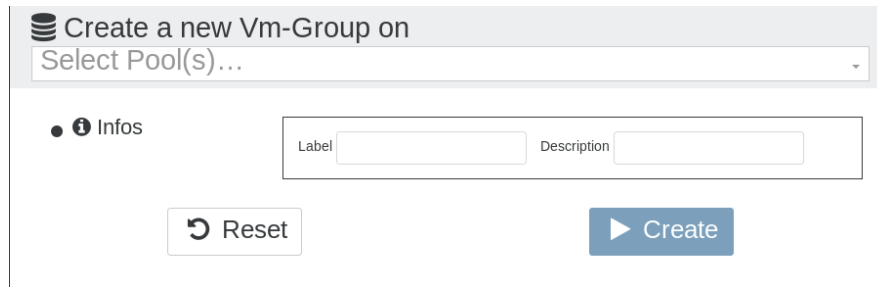
Tous ces traitements sont très similaires, mais tout de même différent, il n’est donc pas forcément judicieux de créer un composant commun pour tous.

Création d’un groupe de machines virtuelles

La gestion et l’affichage des informations concernant un VmGroupe étant terminé, il est nécessaire de pouvoir créer des groupes depuis xo-web. Pour ce faire un composant « VmGroup » définissant un formulaire est créé dans le dossier «xo-app/new/vm-group». Le fichier index situé dans le dossier « new » définit que ce composant sera affiché à l’adresse « new/vm-group ».

Ce formulaire est très simple, il comporte trois champs : un champ de sélection du pool sur lequel sera installé le VmGroup, ainsi qu’un champ permettant de lui donner un label et un dernier, pour une éventuelle description. Le choix du pool se fait grâce à un composant « selectPool » qui propose une liste de pool, de manière similaire au composant « selectVm ».

Lorsque la valeur d'un de ces champs est modifiée, celle-ci est mise à jour dans le state du formulaire « VmGroup ». Quand il est validé, un groupe est créé grâce à la fonction « createVmGroup » qui appelle la fonction de creation d'un VmGroup de xo-server. Après validation du formulaire, l'utilisateur est redirigé vers la liste des groupes de machines virtuelles.



Create a new Vm-Group on

Select Pool(s)...

Infos

Label Description

Reset Create

Figure 57 : Formulaire de création d'un groupe de machines virtuelles

Conclusion

Cette période de stage aura été très enrichissante. En effet, j'ai appris à manier le langage ES6 qui aujourd'hui ne me pose plus aucun problème de compréhension. De plus l'apprentissage de l'utilisation de la librairie ReactJS est pour moi un atout car c'est une technologie très répandue dans le monde professionnel, et que j'ai trouvé, au final, très intuitive et qui a l'avantage d'être rapide à prendre en main.

Ce stage m'a aussi apporté l'occasion d'utiliser git de manière conventionnelle. En effet, j'avais déjà eu l'occasion de m'en servir, dans le but d'avoir accès à mes projets depuis n'importe quel poste de travail, mais jamais de l'utiliser en équipe.

En ce qui concerne la langue anglaise, qui constitue la totalité des documentations que j'ai pu consulter, j'ai pu améliorer ma compréhension écrite de celle-ci en lisant quotidiennement.

Concernant le travail sur le logiciel Xen Orchestra, j'ai trouvé très formateur de participer au développement d'un projet de cette envergure. En effet, l'exercice de compréhension de code, que je n'avais que très peu produit jusqu'ici, ne m'était pas familier. De plus n'ayant encore jamais eu l'occasion de développer un serveur, les notions de client et serveur étaient encore un peu abstraites au début de ce stage car elles étaient, pour moi, uniquement théoriques.

Bibliographie

- Xen Orchestra : <https://xen-orchestra.com>
- Dépot git de Vates : <https://github.com/vatesfr>
- Babel: <https://babeljs.io>
- Node: <https://nodejs.org/>
- Yarn : <https://yarnpkg.com/>
- React : <https://facebook.github.io/react/>
- Lodash : <https://lodash.com>
- Jest : <https://facebook.github.io/jest/>
- Redis-server : <https://redis.io/>