|  |  |  |
| --- | --- | --- |
| [Tapez le titre du document] | [Sélectionnez la date]  [Année] | |
| [Tapez le résumé du document ici. Il s'agit généralement d'une courte synthèse du document. Tapez le résumé du document ici. Il s'agit généralement d'une courte synthèse du document.] | | [Tapez le sous-titre du document] |

Sommaire

[Introduction 1](#_Toc484440069)

[L'entreprise 2](#_Toc484440070)

[Gestion de projet 2](#_Toc484440071)

[Cahier des charges 3](#_Toc484440072)

[Technologies utilisées 4](#_Toc484440073)

[NodeJS 4](#_Toc484440074)

[Yarn 4](#_Toc484440075)

[ReactJS 4](#_Toc484440076)

[Babel 5](#_Toc484440077)

[Travail réalisé 6](#_Toc484440078)

[Prise en main des différents outils 6](#_Toc484440079)

[Développement du module withState 7](#_Toc484440080)

[Développement de Xen Orchestra 12](#_Toc484440081)

[Réalisation et tests 15](#_Toc484440082)

[Liens 15](#_Toc484440083)

# Introduction

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

Le but de ce stage est découvrir le monde professionnel, tout en approfondissant des connaissances acquises qui ont pu être vu en cours, comme lors de projets personnels. Lors 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 la formation, je ne me sentais pas à l'aise avec celui-ci. Cette technologie étant largement utilisée dans le monde professionnel, il me paraissait important d'étendre mes connaissances à son sujet.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/!\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Le projet proposé par Studio Goliath répondait entièrement à mes attentes puisqu’il est de développer une application WEB à l’aide Symfony2.

Ce rapport commence par une présentation de l’entreprise, pour situer l’environnement dans lequel j’ai pu évoluer au cours cette période. Viens ensuite une présentation du projet, puis une présentation du travail que j’ai réalisé, et pour finir, une partie concernant l’avenir de ce projet.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **L'entreprise**

Vates, qui du latin signifie "devin" ou "oracle", est également une entreprise de développement open source. Situé à Grenoble, au 3 rue Pierre Termier, Vates a été créé en 2006 par Olivier Lambert, Julien Fontanet et Nithida Keophilavong.

De 2006 à 2012 Vates développait des solutions open source....  
Depuis 2012, Vates se consacre uniquement au développement, à la maintenance et au support du logiciel Xen Orchestra également open source.

L'entreprise est composée de cinq salariés : un directeur technique, nit, deux développeurs,

# Gestion de projet

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

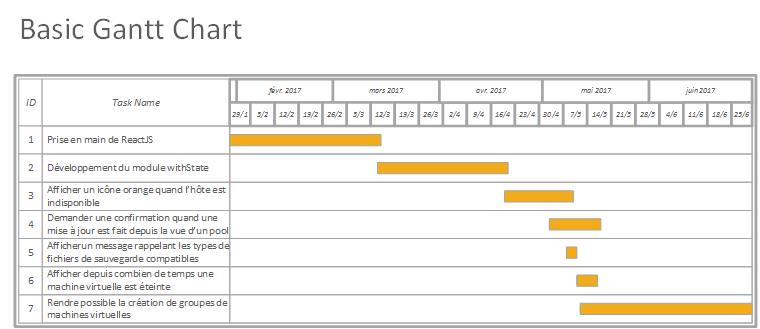
- Mattermost qui est un chat sur lequel l'ensemble de l'équipe peut échanger. En effet, une partie de l'équipe effectuant son activité en télétravail, un outil de communication est nécessaire

-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 projet utilisant git sur lequel Xen Orchestra est disponible dans un dépôt public. Dans ces plateformes, chaque dépôt possède une section dans laquelle tout le monde peut ouvrir des rapports des bugs. Ces bugs sont ensuite assignés à un développeur par un directeur technique

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. Un directeur technique va ensuite relire le travail qui a été fait et accepter la demande de fusion ou faire des commentaires sur le code produit par le développeur. Ce dernier pourra alors soumettre une nouvelle version

# 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.



# **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

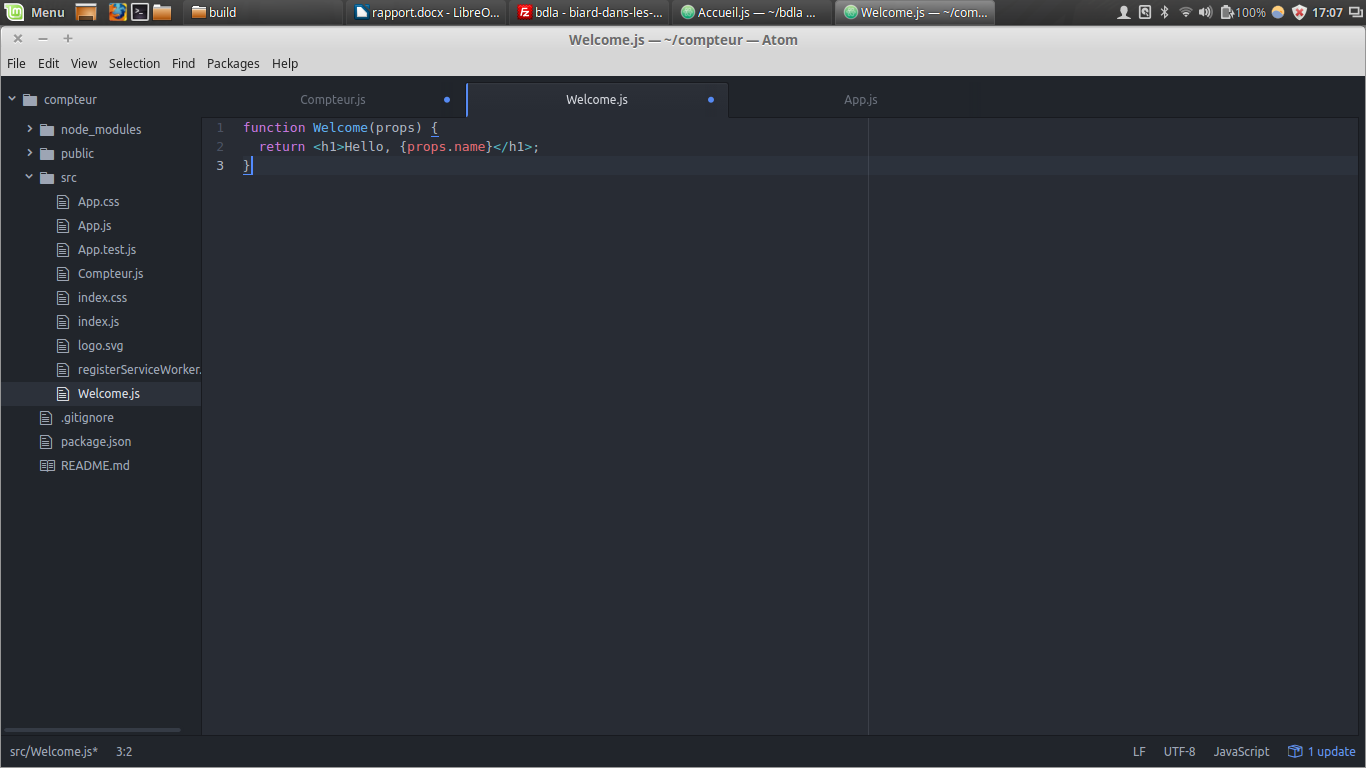
Yarn est un gestionnaire de paquets. Il permet donc d'installer des modules déjà développés par une communauté, et mis en libre accès, évitant ainsi de devoir développer des fonctionnalités qui existent déjà. De la même manière que npm, yarn utilise un fichier package.json situé à la racine du projet dans lequel sont définis les paquets desquels dépend le projet 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).

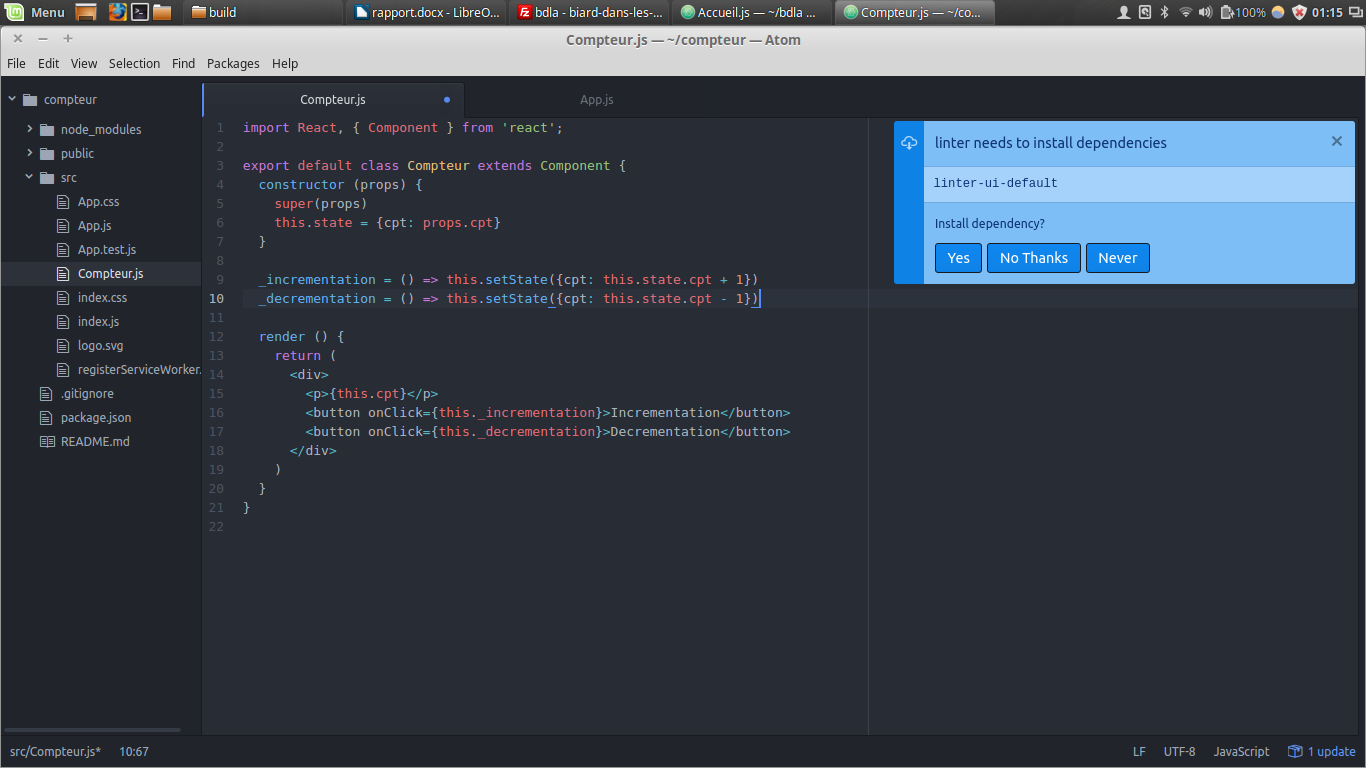
## ReactJS

React est une bibliothèque Javascript développé 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.

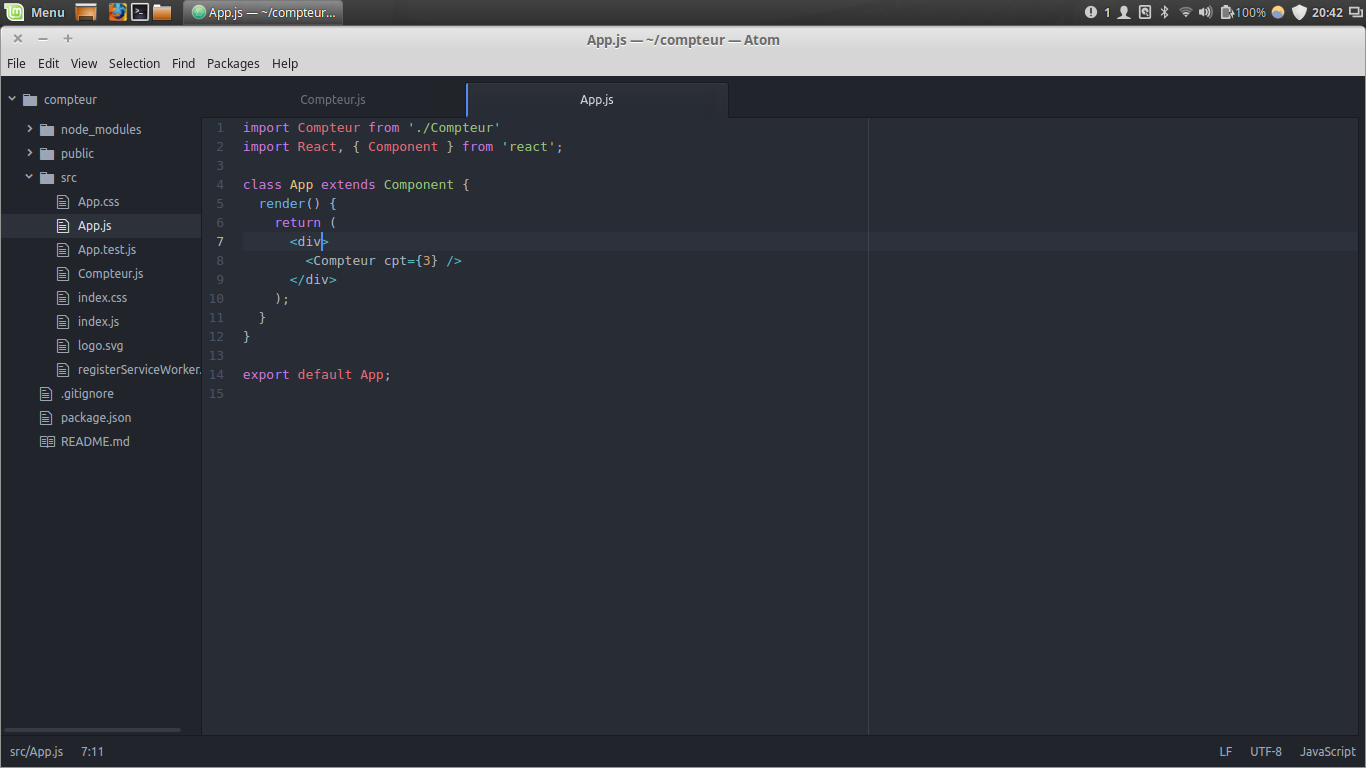
Un composant React est constitué de :

* Un State qui un objet [[1]](#footnote-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

Un composant peut aussi être déclaré à l'aide d'une classe, voici 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'incrémentation et de décrémentation, exécutées lorsqu’un clic est effectué sur les boutons correspondants. Ce type de 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

## 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 étape.

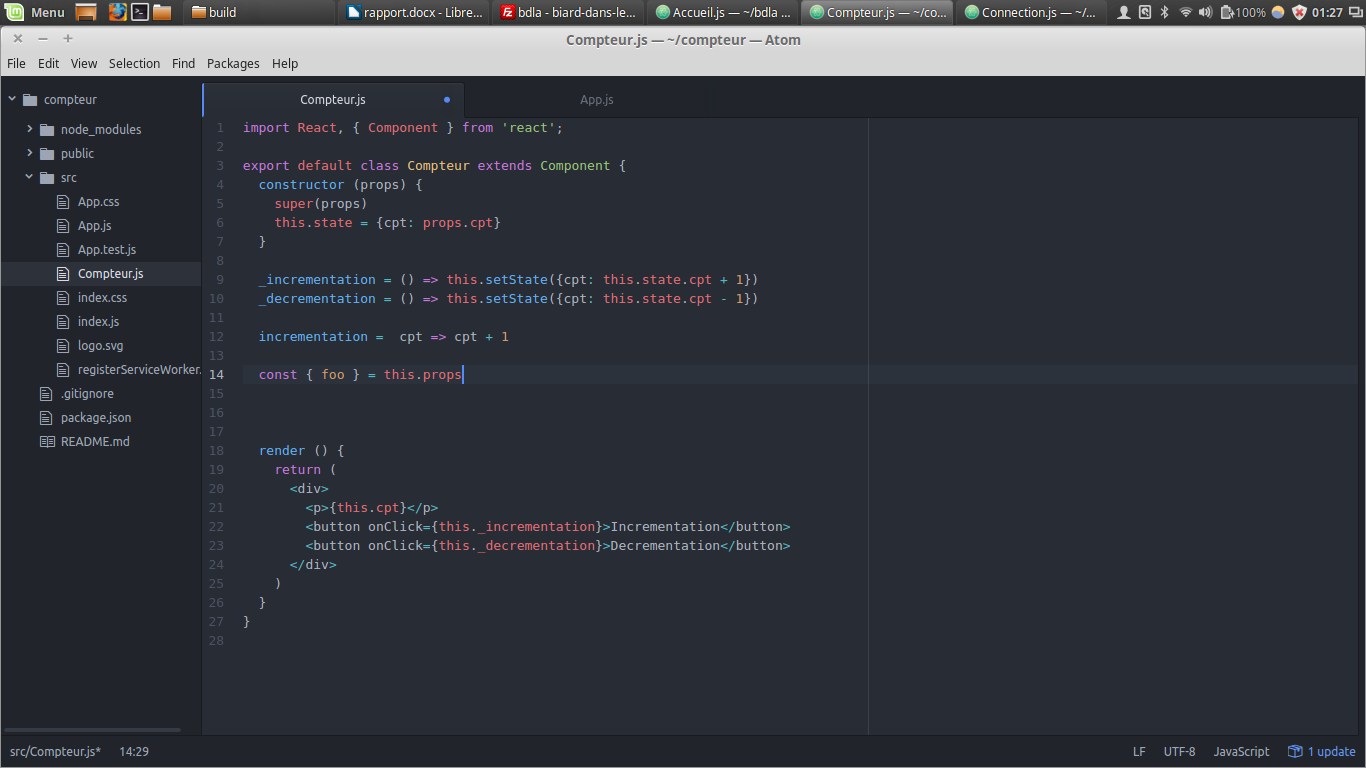
# Travailréalisé

## Prise en main des différents outils

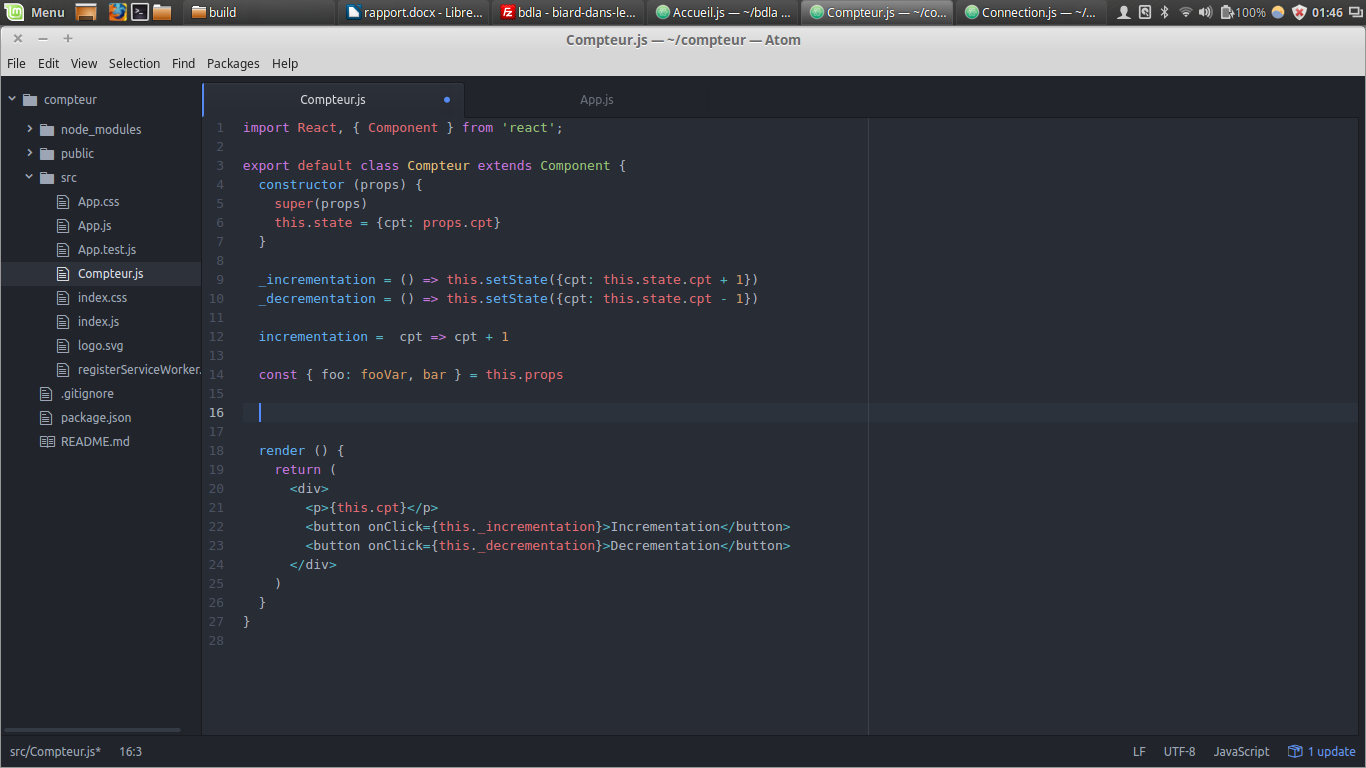
Pendant cette phase de prise en main, j’ai développé une petite application de type site vitrine. Elle contient différente pages comme une authentification, permettant d’accéder à une page privée, ou encore un gestionnaire du thème de l’application.

### 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 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

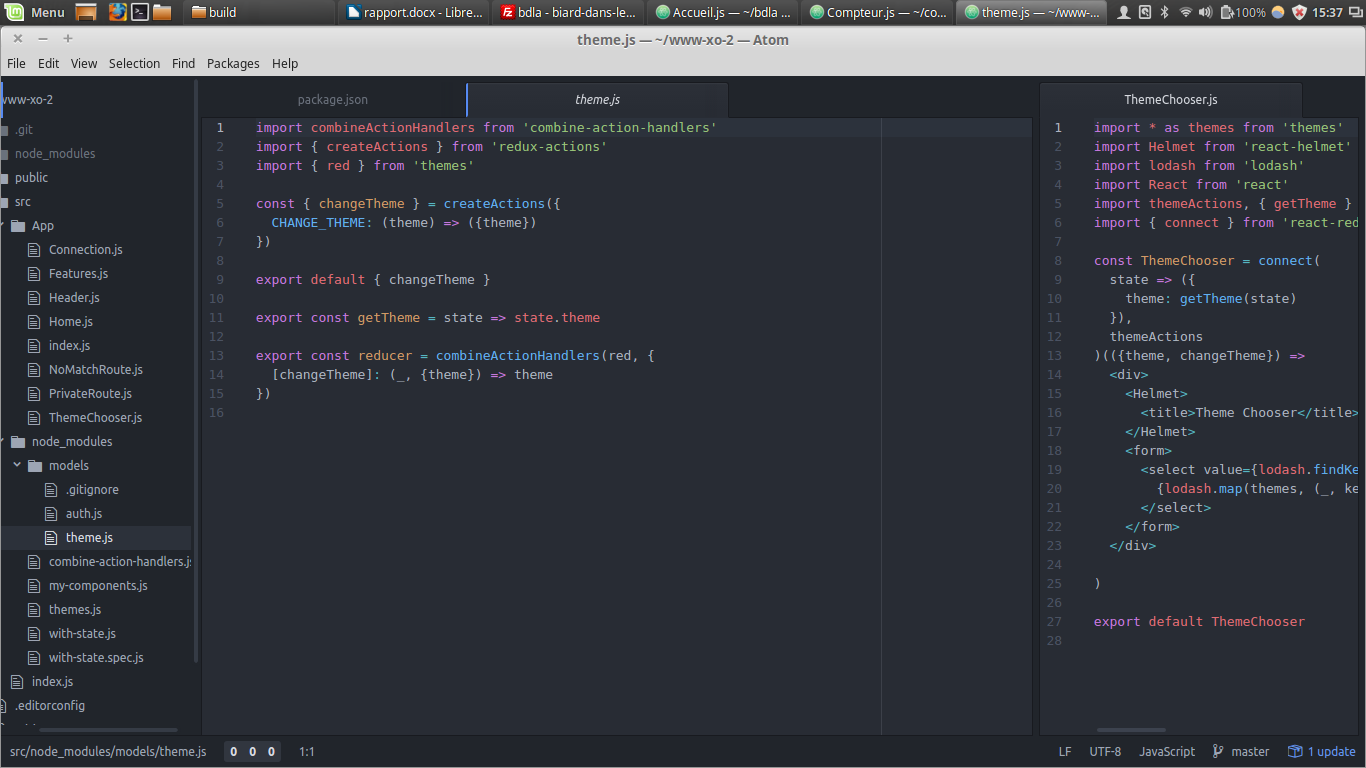
Suppression des parenthèses des paramètre d'une fonction : Lorsqu'une fonction prend un seul paramètre, il possible d'omettre celles-ci, comme dans l'exemple ci-dessus.

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

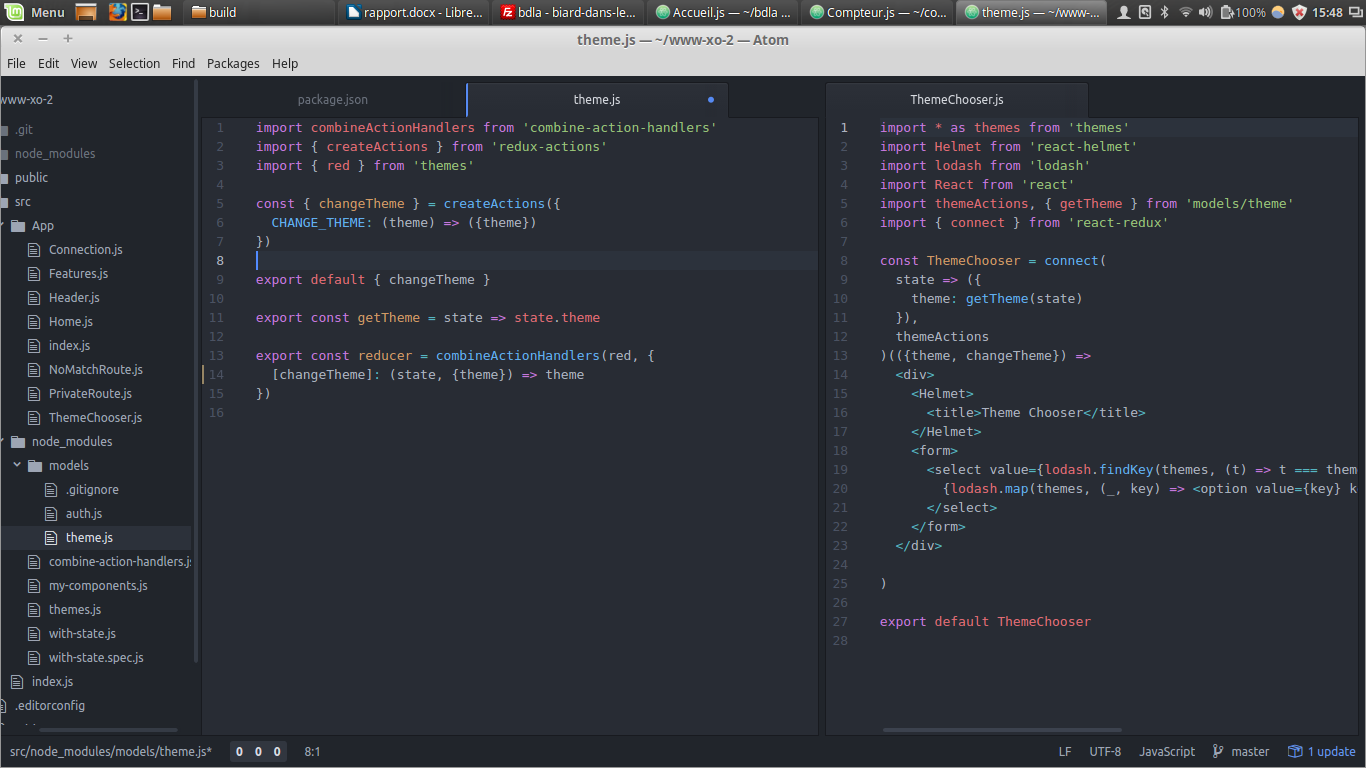
L’opérateur « … » : Cet opérateur s’applique sur un objet. Il permet de copier tous les champs qu’il contient. Ainsi les deux variables suivantes possèdent des champs semblables mais ne sont pas égales, car une copie a été effectuée.

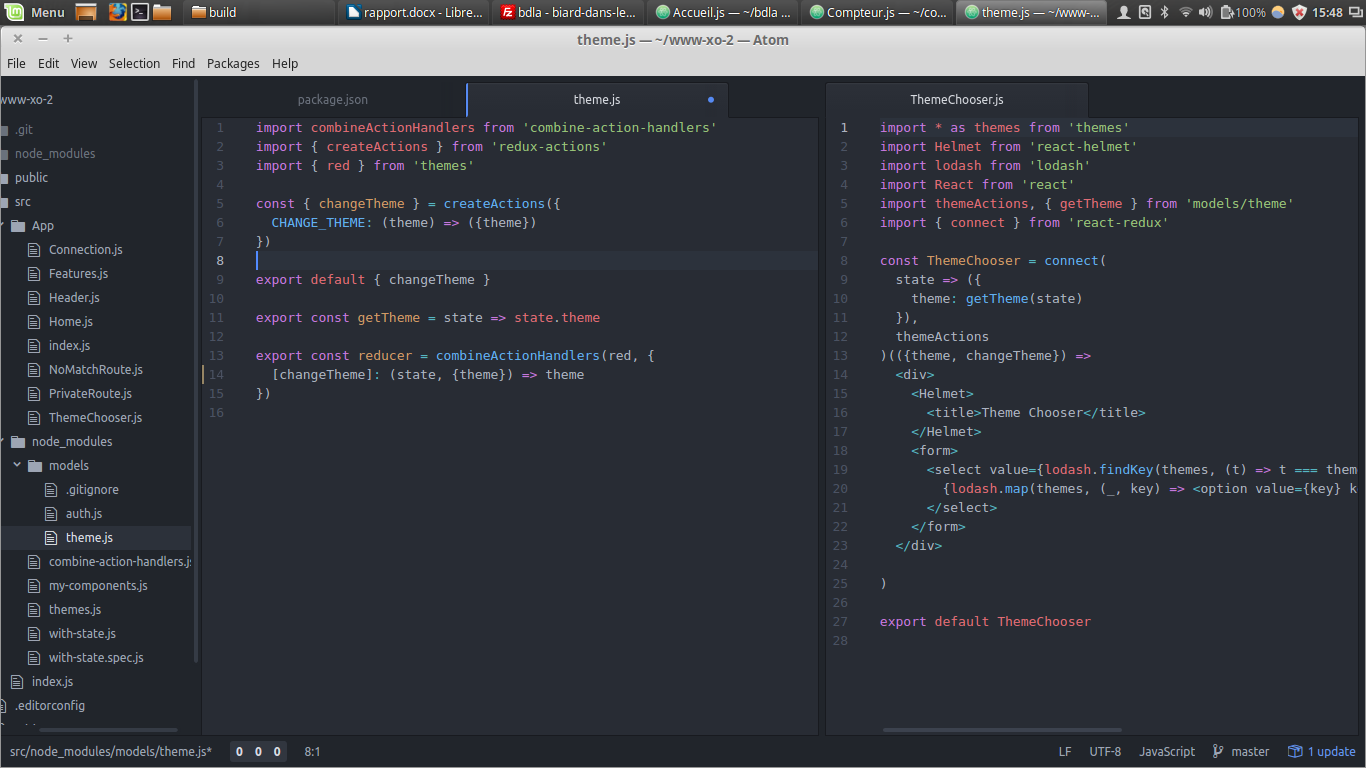
### Redux

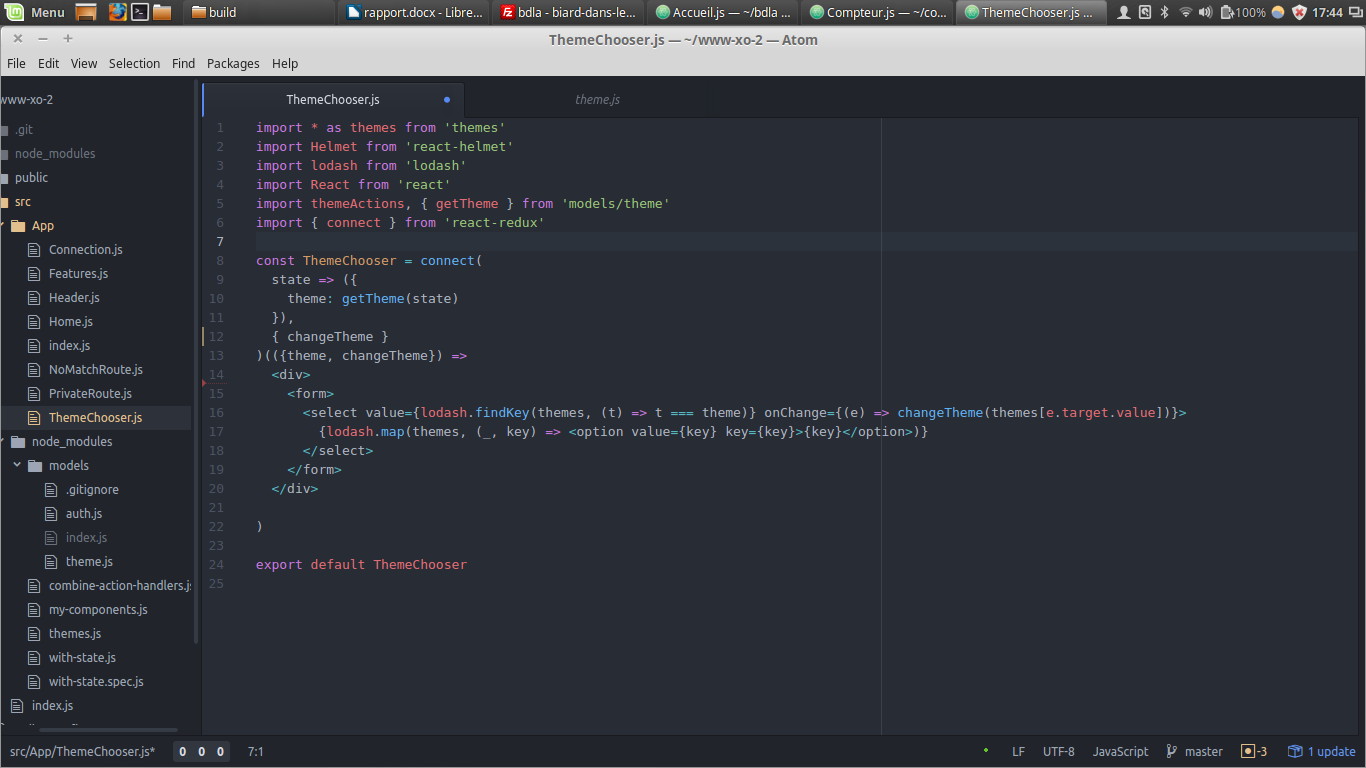
Redux est une librairie dont le but est la gestion de différents états de l'application, et donc de centraliser les données nécessaires au fonctionnement de l'application. Elle permet donc la gestion d'un state. Pour ce faire, redux met à disposition des fonctions de créations d'actions, de sélecteurs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

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

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. Le reducer prend 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.

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

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 le state. L'envoie du résultat d'une action au reducer se fait grâce à une fonction fournie par Redux appelé dispatch, mais celle-ci n'est pas non plus disponible dans le contexte d'un composant. Pour ce faire, Redux fourni un décorateur [[2]](#footnote-2) appelé connect. Cette fonction prend en argument une fonction à laquelle le state est passé en argument, ce qui permet d'utiliser les selecteurs, ainsi qu'un objet. Pour chacun des champs de cet objet, une nouvelle 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 valeurs calculées précédemment et retourne un nouveau composant. Voici un exemple de l'utilisation du décorateur connect.

react

github

react router

tests

## Développement du module withState

### Objectifs

Le module withState 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 niveau[[3]](#footnote-3) mais pouvant posséder plusieurs sous-states[[4]](#footnote-4). 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

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

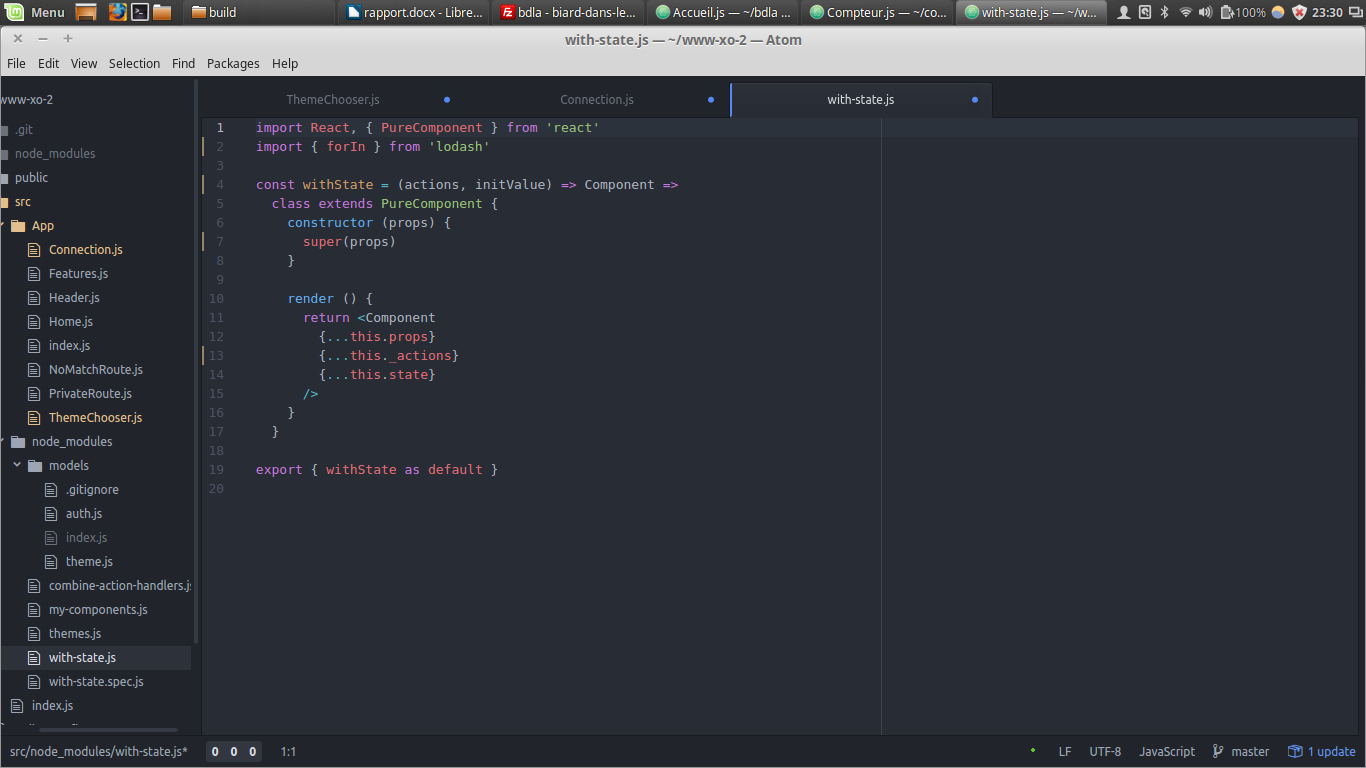
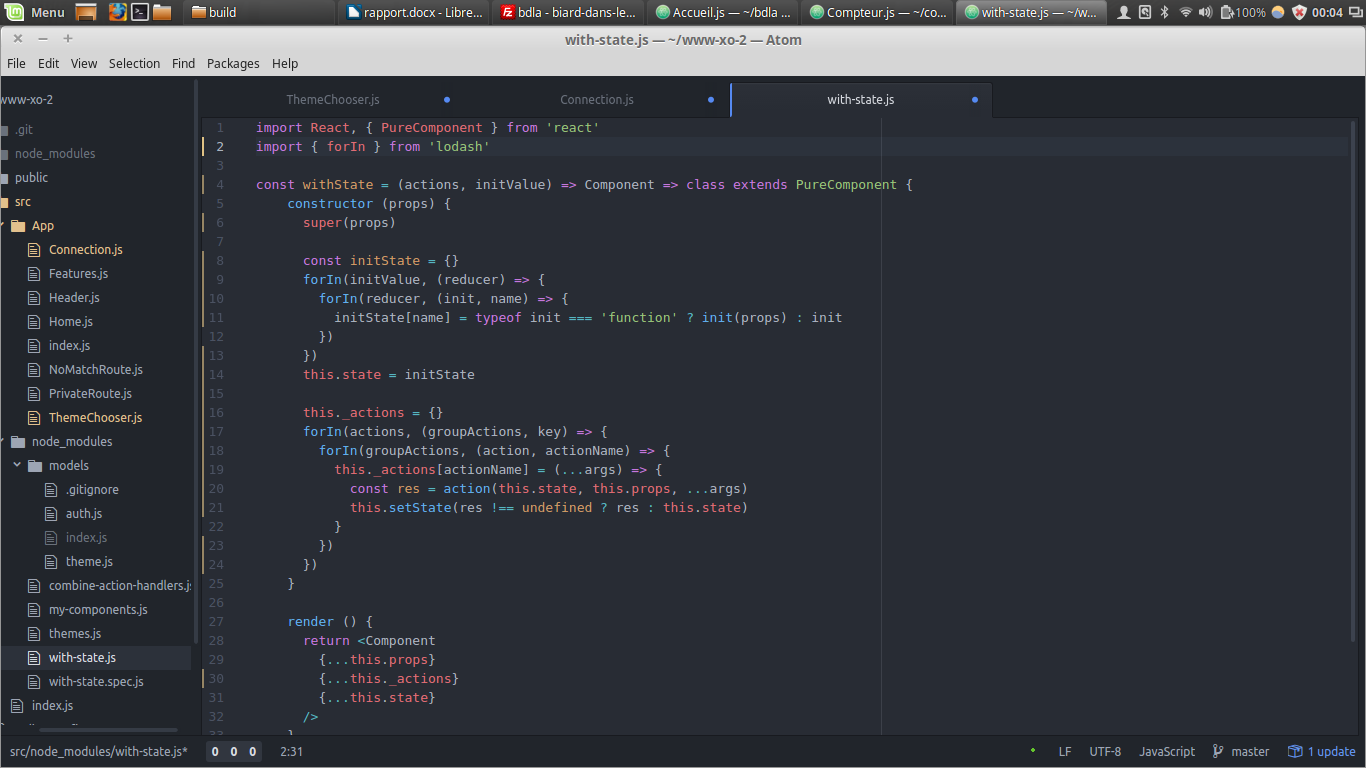
Dans cet exemple, le premier argument est définit par un objet comprenant des fonctions permettant de modifier les valeurs qui seront dans le champs « credentials » du state du composant final. Chacune de ces fonctions prend trois arguments, dans l’ordre : le state associé à la fonction (ici, state.credentials),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

Le second argument de withState représente la valeur initiale qu’aura le state.

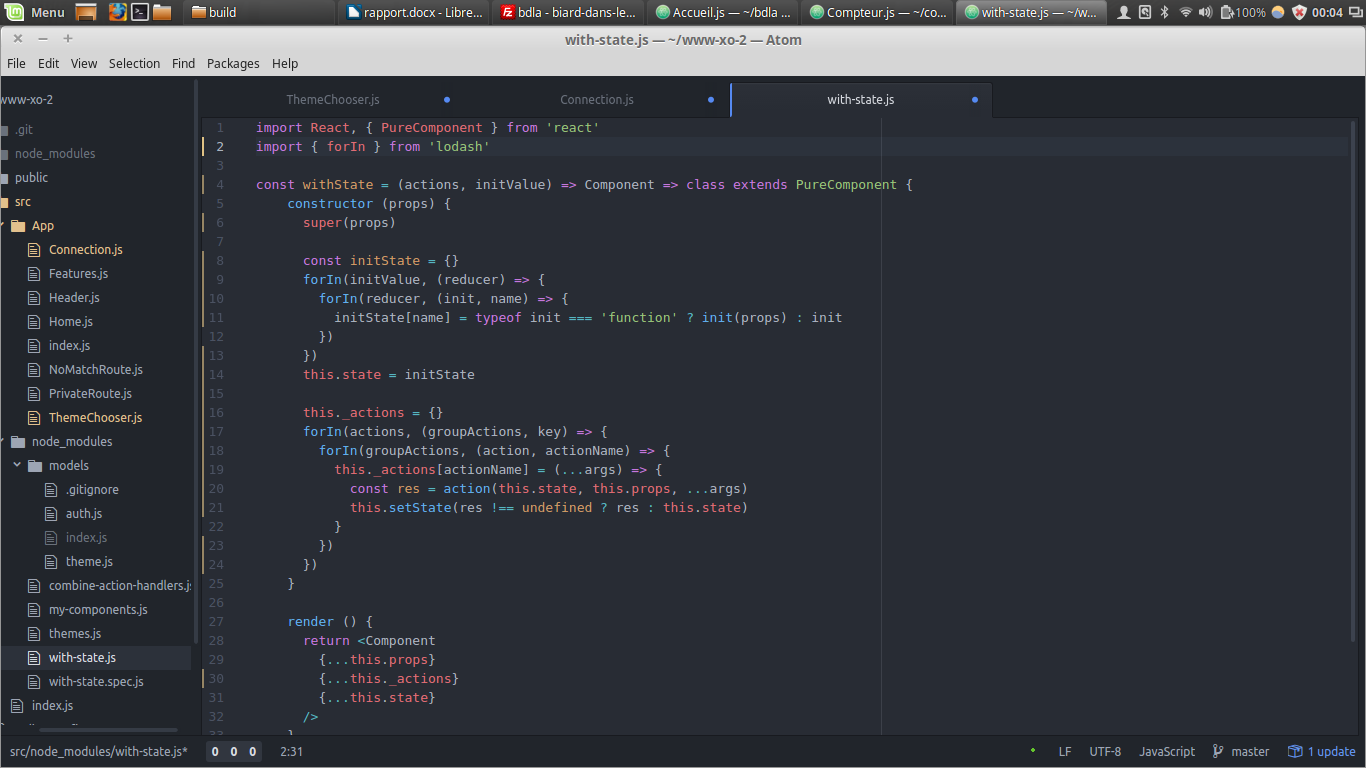
Ici le composant résultant du traitement de withState contiendra un state avec un sous-state nommé « credentials » comprenant des champs « username » et « password ». De plus la variable props du composant contiendra une fonction renvoyant la nouvelle valeur du champ concerné pour chacun d’eux, respectivement « onLoginChange » et « onPasswordChange » et une fonction « onSubmit ».

Comme on peut le voir ci-dessus, withState est une fonction qui en renvoit une autre qui prend en argument un composant react.

Il est d’abord nécessaire de penser à la structure de notre module. Le module a donc la structure suivante :

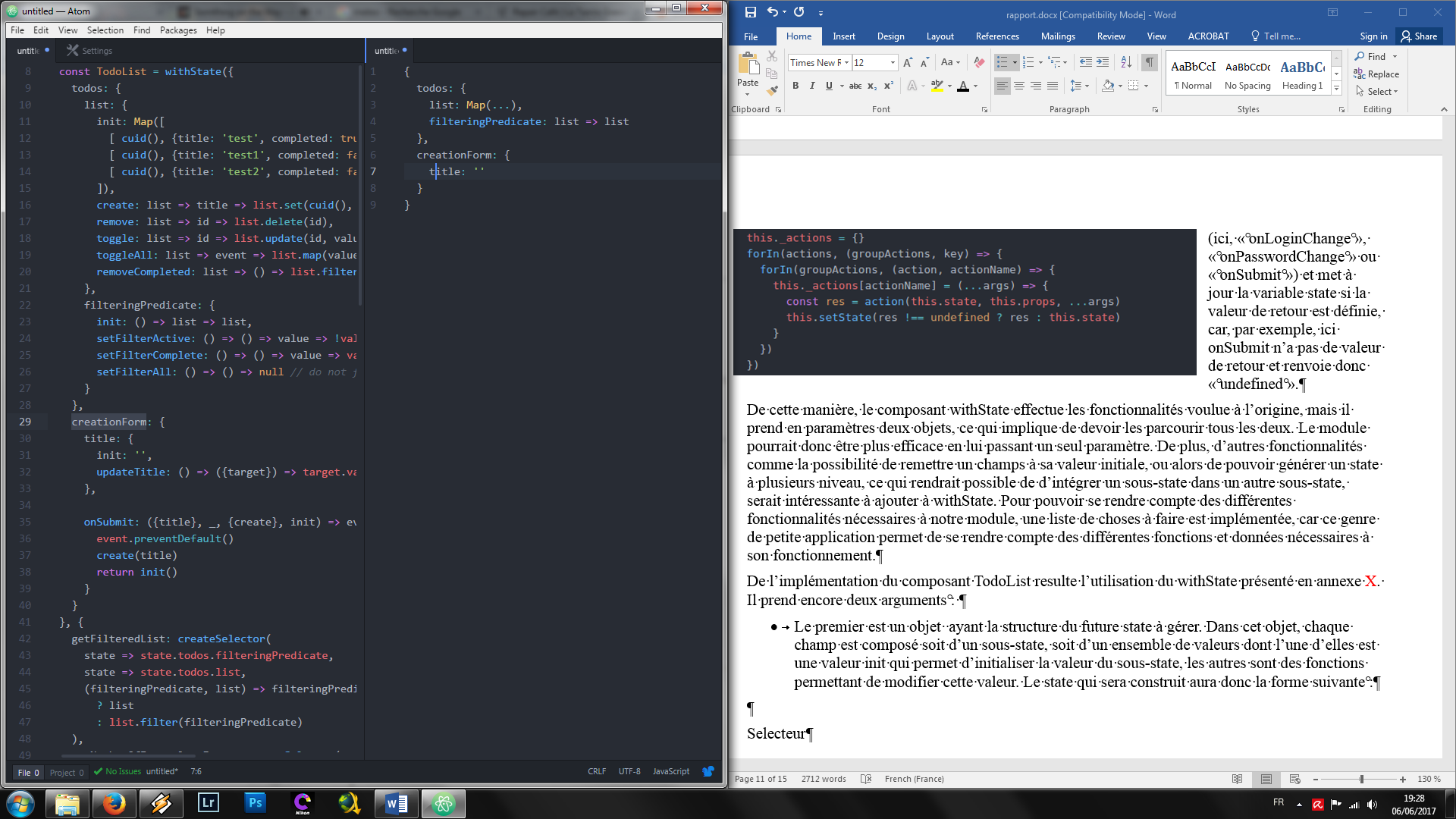
L’objet contenant les actions et la structure du state sont reçus dans les arguments « actions » et « initValue », et le composant dans la variable Component. La fonction déclare et retourne un composant qui sera décoré. Il possède un constructeur dans lequel les traitements permettant de construire le state final. Les fonctions passées dans la variable « actions » seront stockées dans la variable this.\_actions. Le composant que l’on décore est ensuite instancié en lui transmettant les variables props, \_actions et state, dans son champs props.

Il est ensuite nécessaire de construire le state contenu dans initValue. En effet, nous autorisons qu’une valeur puisse être une fonction qui initialise la valeur via les propriétés passées au composant. Pour ce faire, il est nécessaire de parcourir les sous-states de l’objet « initValue », puis de parcourir les différents champs de celui-ci. Dans chacun de ces champs, on initialise la valeur du nouveau state via la fonction si s’en est une, sinon avec le champs directement.

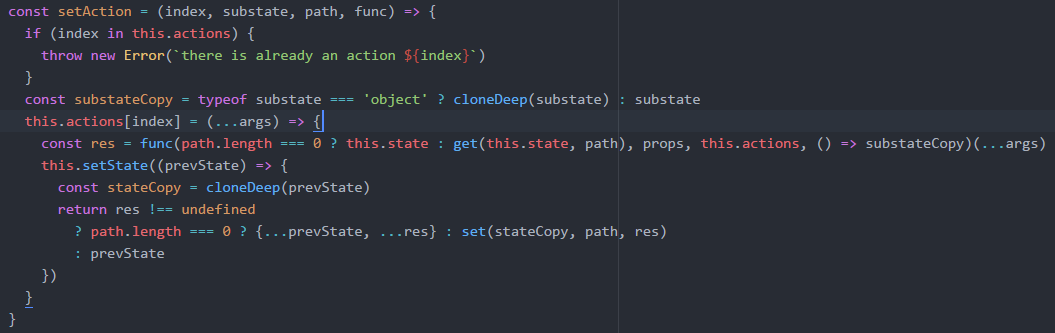
La construction de l’objet contenant les actions se fait en parcourant l’objet « actions » de la même manière que précédement. Chacune des fonctions qui sont enregistrées dans « this.\_actions », lorsqu’elle sont éxecutées, récupère la valeur de retour d’une fonction passée dans l’objet « action » (ici, « onLoginChange », « onPasswordChange » ou « onSubmit ») et met à jour la variable state si la valeur de retour est définie, car, par exemple, ici onSubmit n’a pas de valeur de retour et renvoie donc « undefined ».

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 paramètre. 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 de d’intégrer un sous-state dans un autre sous-state, serait intéressante 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 X. 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. Ces fonctions ont la signature suivante :   
  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 déclaré dans la structure, ainsi qu’une fonction « init » qui renvoie sa valeur initiale. Cette fonction en renvoie une autre qui prend en argument la liste des arguments passés lors de l’exécution de la fonction.  
  Le state qui sera construit aura la forme ci-contre.  
  Le champs « list » la liste de choses à faire, filteringPredicate, un filtre permettant d’afficher toutes les choses, celles qui restent à faire ou celles faites, et « title », le titre d’une chose à faire.
* Durant l’implémentation du composant, on s’aperçoit que l’on a besoin de’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 la fonction dans le premier argument car le résultat des fonctions qui y sont présentes sont directement injectés dans le state, il n’est donc pas possible de recueillir leur valeur de retour. On choisit donc de passer un second argument qui contient les sélécteurs décris précédement. Un sélécteur prendra en paramètre le futur state, et les props passées au composant pour pouvoir renvoyer la valeur souhaitée.

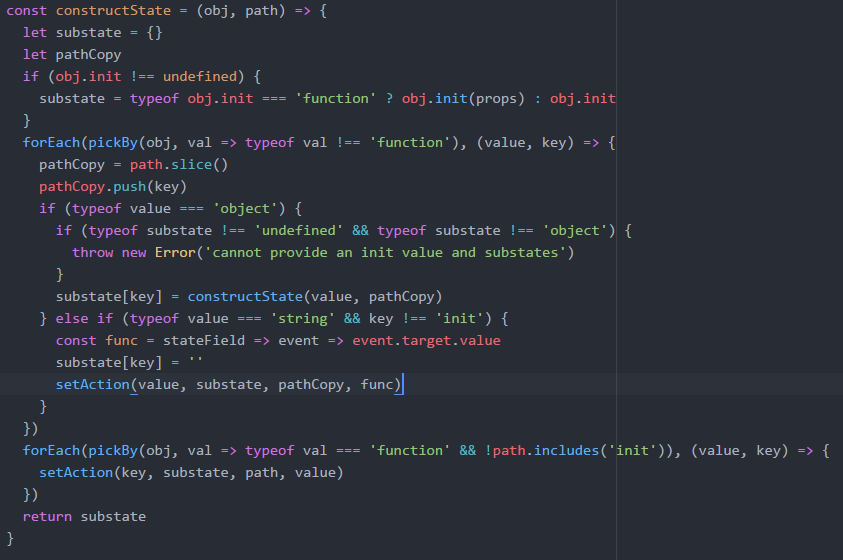
La nouvelle structure du module withState est la même que précédement mis à part que les deux arguments qui lui sont passés s’appellent maintenant « states » et « selectors ».

Dans le constructeur du composant générer 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 :

Elle prend en argument :

* L’index de la fonction, qui est aussi son nom. C’est donc une chaine de caractères.
* Le sous state dans lequel dans lequel elle est déclarée
* Le chemin permettant d’atteindre ce sous-state, sous forme d’un tableau. Ainsi, si la fonction qui est traitées 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 liste de paramètres qui lui est passée en arguments. L’action sera éxecutée, elle va ensuite éxecuter en prenant les paramètres décris précédemment, la fonction qui lui est passée en argument. Sa valeur de retour sera ensuite utilisée pour mettre le sous-state à jour.

Il est ensuite nécessaire de construire le state. Celui-ci comprenant un nombre de niveaux non définit, le state va être construit de manière récursive. La fonction constructState qui prend en paramètre 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 valeur modifié 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 traité, elle sera ignorée si la valeur courante est de ce type.
* Un éventuel sous-state, dans ce cas, le sous-state est construit en rappelent 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 caractère ayant une chaine vide comme valeur initiale. Un sous état est 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 initiale aux différentes fonctions. Des actions sont définits lors du premier parcours mais celles-ci ne nécessitent pas d’avoir accès à la valeur initiale

Par la suite, les selecteurs sont parcourus grâce à la fonction « setSelectors » suivante :  
La fonction mapValues permet de parcourir un objet en modifiant ses champs. Ainsi, un selecteurs sera une fonction qui éxecute le sélécteur déclaré dans l’argument de « states » withState, en lui passant les paramètres nécessaires.

### Tests

withState doit pouvoir supporter les fonctionnalités suivante :

* 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 chaine de caractères vide doit pouvoir se faire en lui donnant pour valeur le nom de la fonction qui pourra la modifier grâce à un évènement
* 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 fournis 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 X. Ceux-ci étant validés, le module « withState » remplie 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 le 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 nécessaire au fonctionnement d’un composant de manière organisée et, je pense, de manière plus facilement lisible.

## Développement de Xen Orchestra

### Présentation de Xen Orchestra

Xen Orchestra est un gestionnaire de machines virtuelles. Il permet d'administrer des serveurs tournant sous XenServer. XenServer est un logiciel basé sur Xen qui est un hyperviseur de machines virtuelles. XenServer permet la gestion de machines virtuelles hébergée sur des serveurs.

Il est composé de trois grand 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

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

* VM :
* Host
* Pool :
* Disk :

Gestionnaire de machines virtuelles

Redis

Internationalisation

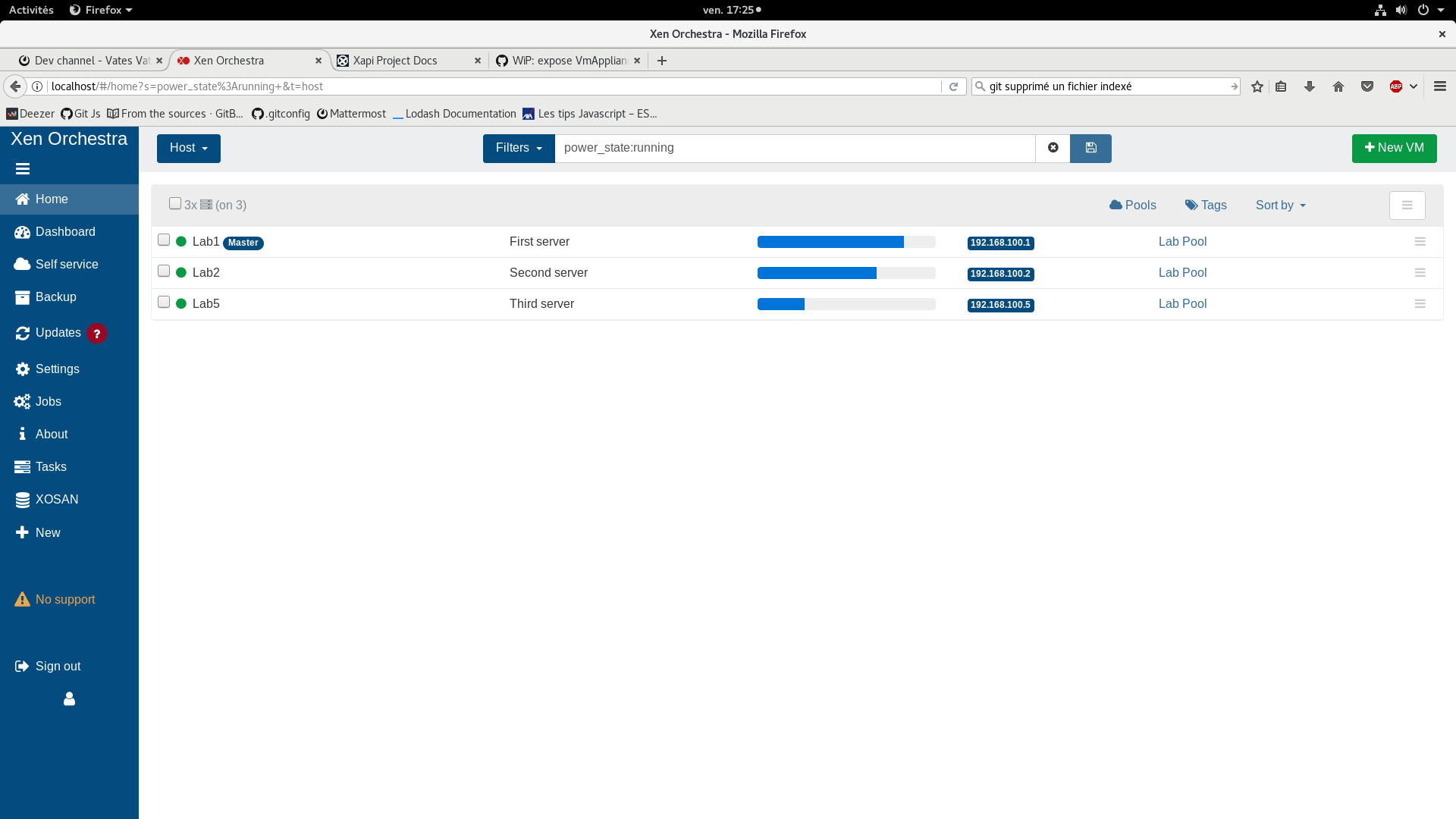
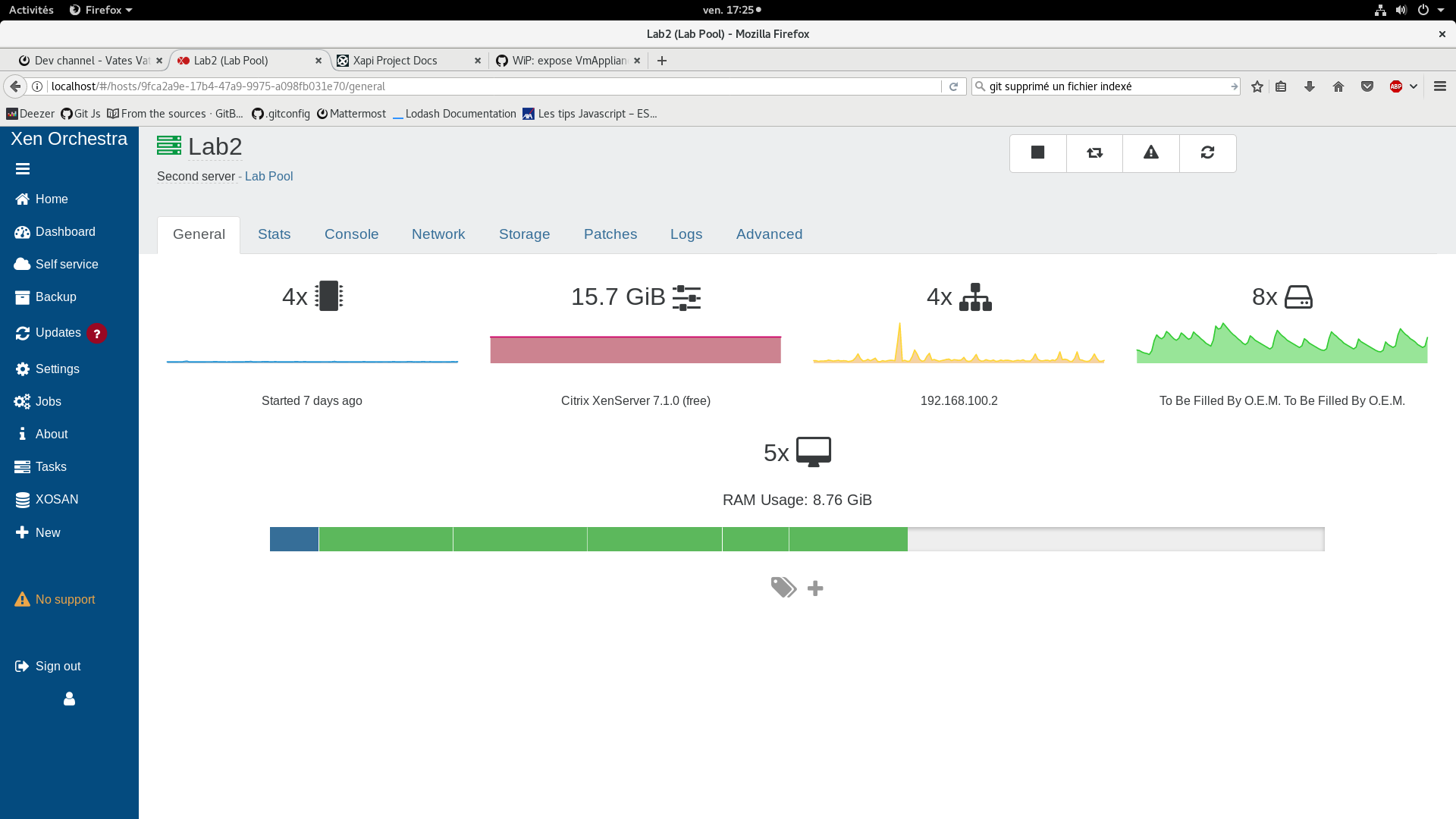
prix ???

Xen Server

Architecture

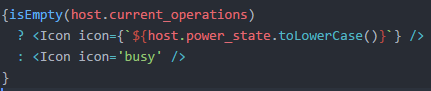
Démo

### Afficher une icône orange quand l'hôte est indisponible

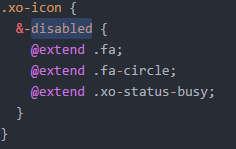
Les vues de liste des hôtes et détaillée d’un hôte ressemblent à ceci :

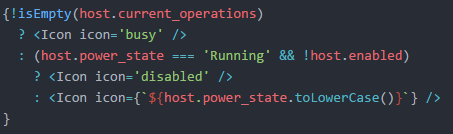
Ici, le point situé en début de chaque ligne sur la liste, ainsi que l’icône représentant un serveur sur la vue détaillée en haut à gauche, sont vert car l’hôte est démarré. Ils pourraient être également rouge lorsque l’hôte est éteint. Le bug qui m’a été assigné consistait à afficher ces icônes en orange lorsque que l’hôte est démarré, mais qu’il est indisponible.

Un hôte est caractérisé par différents attributs dont « power\_state » qui est une chaine de caractères qui représente son état. Ses valeurs possibles sont « halted » (éteinte) ou « Running » (allumé). Un hôte contient aussi un champs « enable » qui est un booléen qui définit s’il est disponible, ainsi qu’un champs « current\_operations » qui est tableau de chaines de caractères représentant des opérations en court d’exécution. On définit qu’un hôte est indisponible s’il est allumé, que son champs « enable » est à « false » et qu’aucune opération n’est en court.

Concernant le point, celui-ci est définit 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 contenant à l’origine le code suivant :

Le point est représente 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 court, un icône de type « busy » est affiché. Un tel icône s’affiche bien en orange, mais son appelation 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 definis les différents style qu’un icône peut prendre. On y définit donc un style disabled :

Les règles de style définies se trouve 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 ». .fa ???

Il va ensuite falloir revoir la condition d’affichage de l’icône « disabled ». S’il y a des opérations en court, 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 champs « power\_state ». Ce qui donne le code suivant :

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

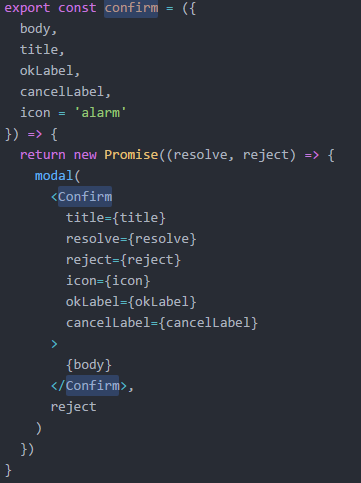
### Demander une confirmation quand une mise à jour est faite depuis la vue d'un pool

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

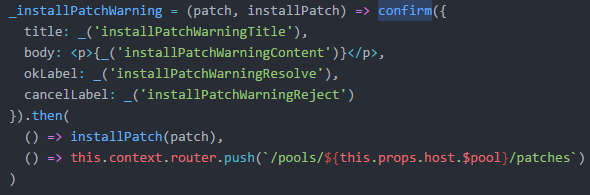
* A partir de la vue d’un pool, une mise à jour globale des hôtes qu’il héberge. Dans ce cas un algorithme intelligent est exécuter et va choisir les mises à jour nécessaire.
* A partir de la vue d’un hôte, l’utilisateur peut choisir les mises à jour qu’il veut installer. Seulement cette méthode peut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

Ajouter screenshot

C’est en appuyant sur un des boutons bleus à droite 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 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 interpelé 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 possible : 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 composant, « Confirm » et « alert » de type Modal qui utilisent le patron donné par React-Bootstrap[[5]](#footnote-5). « 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 exécutée lorsque la proposition sera acceptée ou refusée.  
Une fonction « confirm » est également définit dans ce fichier. Cette fonction permet d’instancier le composant « Comfirm ». Elle prend en paramètre le contenu du message, son titre ainsi qu’une chaine de caractère représentant une icône. Elle crée et retourne une promesse[[6]](#footnote-6). 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 associé 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 :

Elle prend en argument le patch qui doit être installé et une fonction permettant de l’installer. Elle va exécuter la fonction « confirm » est lui donnant le contenu nécessaire et retourner son resultat qui est la promesse que nous avons vue précédement. L’exécuteur de la promesse est exécuté 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 passé en argument sera exécutée, et s’il décline, la seconde sera exécutée. Voici le rendu final de la demande de confirmation.

Screenshot modal

### Afficher un message rappelant les fichiers de sauvegarde compatibles

Dans Xen Orchestra, il est possible de faire des sauvegardes des machines virtuel

### Afficher depuis combien de temps une machine est éteinte

### Rendre possible la création de groupes de machines virtuelles

# Bibliographie

babel

node

yarn

react

xo

Dépot Gitlab

Dépot Vates

Lodash

Jest

Annexes :

State v1 : <https://gitlab.com/vates/www-xo-2/blob/2d54b2d25dcf78e0cb9703074c59b84565ba6a52/src/node_modules/with-state.js>

TodoListV2 : <https://gitlab.com/vates/www-xo-2/blob/0ae51c7019abe00102407e27b90aa550a8822921/src/App/TodoList.js>

withState v2 : <https://gitlab.com/vates/www-xo-2/blob/0ae51c7019abe00102407e27b90aa550a8822921/src/node_modules/with-state.js>

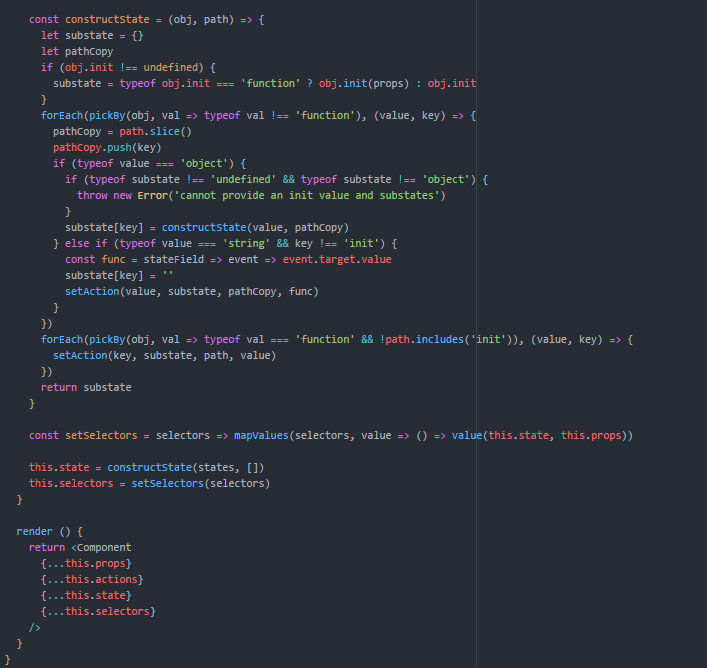
Tests withState : <https://gitlab.com/vates/www-xo-2/blob/0ae51c7019abe00102407e27b90aa550a8822921/src/node_modules/with-state.spec.js>

# Utilisation de la première version de withState

# Première version du module withState

# Composant TodoList utilisant la seconde version de withState

# Implémentation de la deuxième version du module withState



# Fichier de tests permettant de valider le module withState

## Décoration d’un composant stateful

## Décoration d’un composant stateless

## Utilisation d’une fonction pour initialiser la valeur d’un sous-state

## Remise de la valeur d’un sous-state à sa valeur initiale

## 



## Test d’une action qui ne retourne rien

## Initialisation d’une chaine de caractères vide et création d’une fonction de mise à jour de celle-ci

## Déclenchement d’une erreur lorsque plus d’une valeur est déclarée dans un sous-state

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. [↑](#footnote-ref-1)
2. Un décorateur est une fonction qui prend un composant en argument et en renvoie un nouveau [↑](#footnote-ref-2)
3. On entend ici par un seul niveau que chaque champ du state représente un sous-state qui lui, contient des valeurs. [↑](#footnote-ref-3)
4. Dans les exemples suivant, le state n’est composé que d’un seul sous-state. [↑](#footnote-ref-4)
5. React-Bootstrap est librairie permettant d’utiliser le framework css Bootstrap dans un projet React à travers des composant React. [↑](#footnote-ref-5)
6. 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. [↑](#footnote-ref-6)