

|  |
| --- |
| Capitalisation  **Single-page webapp avec Backbone** |
|  |
|  |
| Référence :  Version 0.1 – 23/07/13 |

Sommaire

I. Généralités 4

II. Outils de développement 5

2.1. NodeJS – NPM 5

2.2. Yeoman 5

2.3. Grunt 6

2.4. Bower 6

III. Librairies et frameworks 7

3.1. RequireJS 7

3.2. Backbone.js 7

3.2.1. Les Modèles 8

3.2.2. Les Collections 9

3.2.3. Les vues 10

3.2.4. Les routeurs 11

3.3. Le templating 11

3.4. Backbone.Marionette 12

3.4.1. Pourquoi Marionette ? 12

3.4.2. Les principaux ajouts de Marionette 13

IV. Exemple d’un projet 20

4.1. Récupération et démarrage de l’application 20

4.1.1. Récupération des sources 20

4.1.2. Installation de l’application 20

4.1.3. Lancement de l’application 22

4.2. Explications sur le code Front-end 23

4.2.1. Les fichiers qui initialisent l’application 23

4.2.2. Les contrôleurs de l’application 23

4.2.3. Les vues 25

Historique des révisions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vers. | Date | Modifications | Auteur | Date de validation | Approbateur |
|  |  |  |  |  |  |

Indiquer manuellement les différentes versions du document, ainsi que leur validation dans le cadre de l’usage de ce document dans un projet.

Documents de référence

|  |  |  |  |
| --- | --- | --- | --- |
| Intitulé | Version | Date | Commentaire |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Points restant à traiter

|  |  |
| --- | --- |
| Chapitre | Description élément à traiter |
|  |  |
|  |  |

# Généralités

Les single-page webapps sont des applications web qui ne contiennent qu’une seule page HTML et dont le contenu est entièrement géré par le code JavaScript, HTML et CSS. Ces applications nécessitent donc une façon de coder différente de celle utilisée dans des applications dites « classiques ».

Créer une seule page HTML dont le contenu est géré par le code coté front-end permet de ne pas avoir à charger de nouveau entièrement une nouvelle page avec certains éléments déjà affichés auparavant. Ainsi, l’objectif est d’avoir une application web légère à charger, ce qui est très intéressant pour les mobiles dont la bande passante n’est pas forcément importante et dont la connexion est parfois valable sur une courte durée.

De telles applications tournent sur des navigateurs plutôt récents, c’est-à-dire sur Google Chrome, Safari (5 et plus), Firefox, Opéra et sur Internet Explorer 9 et supérieurs. Tous les navigateurs mobiles sur smartphones sont compatibles avec ce type d’applications.

Ce document présente les différents outils utilisés pour écrire de telles applications ainsi qu’un framework JavaScript appelé Backbone qui facilite le développement de ce type de page web. Il se termine par des explications concernant un projet réalisé avec Backbone.

# Outils de développement

Pour faciliter le développement de single-page webapp, il est conseillé d’installer les outils décrits dans cette partie.

## NodeJS – NPM

Dans un premier temps, il faut installer NodeJS sur le poste qui sera utilisé pour le développement. NodeJS est une plateforme basée sur le moteur JavaScript de Google Chrome et qui permet d’écrire des applications JavaScript côté serveur. Cette plateforme servira aux autres outils que nous allons installer par la suite.

L’installation de la plateforme se fait en se rendant sur le site de [NodeJS](http://nodejs.org/) et en cliquant sur le bouton installer. Il ne reste plus qu’à ouvrir le document téléchargé et à suivre les différentes étapes.

Pour vérifier que NodeJS s’est bien installé, on ouvre une console dans laquelle on tape « **node --version**»

NPM est un gestionnaire de dépendances qui est installé automatiquement avec NodeJS. Il permet de télécharger et d’installer un très grand nombre de librairies et d’outils différents en lignes de commande. C’est par le biais de NPM que nous allons installer les autres outils de développement.

## Yeoman

Cette utilitaire va nous permettre de développer plus facilement et rapidement des applications en nous faisant suivre de bonnes pratiques.

Pour l’installer, il faut que le poste de développement ait Ruby, Git, NodeJS et Compass d’installés. Si vous ne savez si c’est le cas, en tapant la commande, « **npm install yo –g** », qui doit installer Yeoman sur tout le poste, il vous sera indiqué si des éléments manquent et si oui comment les installer. Si tout est déjà présent, vous avez désormais Yeoman d’installé, ou plutôt l’utilitaire « yo » dans la console.

Lors de l’installation de Yeoman, les outils Grunt et Bower (détaillés dans la suite) sont également ajoutés.

Yeoman fonctionne avec des générateurs d’applications qu’il faut installer. Ces générateurs permettent en une ligne de commande de générer les dossiers et les fichiers selon une architecture appropriée. Pour connaître la liste des générateurs, tapez « **npm search generator** » dans la console. Pour en installer un, tapez « **npm install [nom-du-generateur] –g** ». Il sera alors installé globalement, c’est-à-dire disponible depuis n’importe quel dossier dans l’OS.

Pour plus d’informations concernant Yeoman, rendez vous sur le site [yeoman.io](http://yeoman.io/) ou sur leur [dépôt Github](https://github.com/yeoman) avec la liste des générateurs officiels.

Dans notre cas, il est conseillé d’installer le générateur Backbone en tapant la commande « **npm install generator-backbone –g** », puis de créer et de se placer dans le dossier dans lequel nous allons développer notre application. Pour initialiser le projet, il suffit alors d’écrire « **yo backbone** **–template=mustache**». Il est possible d’ajouter d‘autres options comme précisé sur le [dépôt Github du générateur](https://github.com/yeoman/generator-backbone#options). Une fois la commande exécutée, des questions sont posées concernant l’ajout de certains composants. Il est conseillé de répondre oui à la question concernant l’installation de RequireJS.

## Grunt

Grunt est un exécuteur de tâches basé sur JavaScript et qui tourne sur NodeJS. Il permet d’automatiser les tâches de build notamment. De plus, grâce à la génération automatique faite par Yeoman, un fichier Gruntfile qui décrit les différentes tâches est déjà prêt à être utilisé.

Les différentes tâches programmées par défaut sont celles de développement, de test et de build.

La tâche de développement est lancée en tapant « **grunt server** » dans la console (la commande étant lancée depuis le dossier de l’application). Une fenêtre ou un nouvel onglet de votre navigateur préféré s’ouvre et la page que vous avez codée s’affiche. La page par défaut affiche ce qui a été installé et téléchargé lors de l’initialisation du projet. Cela permet de vérifier que tout s’est bien déroulé.

Pour lancer un build, la commande est « **grunt build** » et pour les tests, il s’agit de « **grunt test**». Lors d’une phase de build, les fichiers générés sont placés dans le dossier courant dans un dossier nommé « dist ». Il est possible de changer le nom de ce dossier dans le Gruntfile.

Si vous voulez en savoir plus sur Grunt et comment modifier le fichier Gruntfile, il faut se rendre sur ce [site](http://gruntjs.com/)

## Bower

Bower sert à gérer les paquets, les librairies nécessaires pour coder l’application. Plutôt que de devoir aller chercher à la main toutes les librairies nécessaires telles que jQuery, Backbone, RequireJS, il suffit de taper « **bower search [nom-de-librairie]** » pour chercher le paquet voulu et « **bower install [nom-de-librairie] –save(-dev)** » pour télécharger le paquet et ses dépendances et enregistrer celui-ci dans le fichier bower.json qui enregistre toutes les librairies nécessaires au bon fonctionnement du projet. Grâce à cette sauvegarde, il n’est plus nécessaire d’ajouter les librairies au gestionnaire de sources car il suffit de taper « **bower install** » pour récupérer toutes celles utilisées par le projet.

Dans la façon dont est générée l’application par Yeoman, les librairies seront placées dans le dossier bower\_components sous le dossier app.

Pour plus d’informations concernant cet utilitaire, consultez le [site de Bower](http://bower.io/).

# Librairies et frameworks

## RequireJS

RequireJS est une librairie JavaScript qui permet de faire de l’AMD (Asynchronous Module Definition). Elle permet donc de charger les modules JavaScript requis par un fichier de manière asynchrone.

Avec cette librairie, au lieu de charger un à un tous les fichiers JavaScripts à la fin du fichier HTML, on utilise la ligne :

<script data-main="scripts/config" src="bower\_components/requirejs/require.js"></script>

où src pointe vers le la librairie requirejs et data-main pointe vers le fichier de configuration de RequireJS pour l’application.

Le fichier de configuration doit contenir tous les chemins vers les modules qui seront utilisés dans l’application (les librairies souvent), les dépendances entre chaque librairie, et parfois le lien vers le fichier principal. Vous trouverez un exemple de fichier de configuration sur le site de RequireJS à cette [adresse](http://requirejs.org/docs/jquery.html) ou dans l’application de capitalisation de SQLI.

Pour utiliser un module dans un fichier, on utilise la méthode « define » de l’API de RequireJS. Cette méthode est une fonction qui prend en paramètre un tableau de String (nom des modules ou chemins vers les modules) et une fonction callback (avec pour arguments les noms donnés aux modules). Par exemple :

require(['jquery'], **function**( $ ) {

console.log( $ ) *// OK*

});

L’avantage d’utiliser RequireJS dans notre projet est que l’étape de build compressera également les librairies et les modules utilisés.

## Backbone.js

Backbone est une librairie JavaScript qui permet de structurer une application web sous le format MVC (Modèle-Vue-Contrôleur).

Cette librairie repose sur jQuery (ou Zepto) et sur Underscore (ou Lodash, librairie allégée de Underscore). Toutes ces dépendances ont été téléchargées et configurées lors de l’initialisation avec le générateur Backbone.

### Les Modèles

Un modèle représente une entité qui peut être stockée en back-end. C’est un objet JavaScript auquel on a ajouté des méthodes pour faciliter les modifications et la gestion de la vie de l’objet.

Toutes les méthodes sont listées sur le site officiel de [Backbone](http://backbonejs.org/#Model). Il est ainsi possible de modifier des attributs ou de les modifier, de mettre en place une url pour indiquer comment communiquer avec le serveur pour la gestion d’un objet de ce type, de sauvegarder ou détruire l’objet côté front et côté back-end, de valider les attributs lors de la création ou lors des modifications de l’objet, etc.

Pour créer un modèle, on étend l’objet Backbone.Model de la manière suivante :

var myModel = Backbone.Model.extend({

//url avec laquelle communiquer pour gérer en REST le modèle

urlRoot : ‘/myModel’,

//valeurs par défaut lors de la création de l’objet

defaults : {

name : ‘un objet’,

date : new Date()

},

//création d’une méthode

sayHello: function(argument) {

console.log(‘Hello ‘ + argument) ;

}

}) ;

L’url indiquée permet de communiquer avec le serveur pour les opérations de récupération (GET), création (POST), modification (PUT) et de suppression (DELETE). Par défaut, Backbone suppose que l’API sur laquelle on repose est une architecture REST. Donc pour les opérations de GET, PUT et DELETE, un id sera ajouté à l’url. Ceci est à prendre en compte lors de l’écriture du back-end. Il reste toutefois possible de changer ce comportement par défaut en ajoutant des méthodes spécifiques qui modifient l’url à appeler ou alors en émulant l’http.

Les opérations de modifications et de créations sont effectuées lorsque l’on appelle la méthode « save » sur un objet déjà existant ou sur une nouvelle instance de la classe. Backbone détecte automatiquement si cet objet existait auparavant ou pas.

Pour faire un DELETE en http, il faut appeler la méthode « remove » sur l’objet. Et pour en récupérer un en particulier, il faut soit indiquer l’id de l’objet après l’avoir initialisé, soit définir une nouvelle url dans l’objet et faire le traitement côté serveur.

De plus, lors de changements sur ces modèles, des événements pouvant être interceptés par les vues sont émis. Il sera ainsi plus facile d’afficher les changements du modèle dans une vue grâce à ces événements. On écrira par exemple :

myModel.on(‘change:attribute, doSomething) ;

Lorsque l’attribut « attribute » de « myModel » change, la méthode « doSomething » est appelée. Il est possible d’ajouter un contexte à cette méthode en ajoutant « this » en paramètre de « on »

Il est possible d’ajouter des méthodes propres à chaque modèle comme par exemple des getters et des setters.

getName : function() {

return this.get(‘name’) ;

}

setName : function(name) {

this.set(‘name’, name) ; //ou this.set({name : name}) ;

}

Pour générer facilement un modèle avec le générateur installé, il faut taper « **yo backbone:model [nom-du-modèle]** » et le code concernant ce modèle sera automatiquement placé sous le dossier app/scripts/models avec la structure nécessaire pour l’utiliser avec RequireJS.

### Les Collections

Les collections sont des ensembles de modèles. Elles sont générées par exemple lorsque l’on veut récupérer une liste d’utilisateurs dans une base de données.

Pour structurer ces collections, on indique quel est le modèle qui sert de base à cette collection, l’url avec l’application doit communiquer pour récupérer l’ensemble des modèles. Il est également possible d’ajouter des méthodes personnelles pour faciliter la manipulation des données.

De la même manière que pour le modèle, on étend la collection Backbone pour créer notre propre collection :

var myCollection = Backbone.Collection.extend({

//modèle de base de la collection

model : myModel

//url avec laquelle communiquer pour récupérer la collection

url : ‘/myModels’,

//création d’une méthode

aMethod: function() {

/\*Itération sur l’ensemble des modèles présents dans la collection\*/

this.each(function(model) {

console.log(model.get(‘name’) ;

}) ;

}

}) ;

### Les vues

Les vues sont en quelque sorte les contrôleurs de l’application. Les « véritables » vues (au sens classique du terme) sont en fait les templates utilisés pour afficher nos contenus.

Les vues permettent de manipuler le DOM, d’intercepter les événements et de les traiter, d’afficher ou de masquer du contenu etc. Elles vont donc servir à présenter à l’utilisateur les collections et les modèles et à les modifier en fonctions des actions faites par celui-ci.

var myView = Backbone.View.extend({

//Endroit où doit être insérée la vue

el : ‘#aDivId’

//Template de la vue

/\*Il peut faire référence à un nom de fichier, une méthode, une div selon la façon dont est structurée l’application \*/

template : ‘myTemplate’

//balise dans laquelle la vue doit être encapsulée

tagName : ‘li’

//Classe (DOM) que l’on donne à la vue

className : ‘aClassName’,

initialize : function() {

//Autre façon d’écouter les changements su un modèle

/\*A chaque fois que le modèle de la vue change, on actualise l’affichage\*/

this.listenTo(this.model, ‘change’, this.render) ;

},

//Méthode appelée pour afficher la vue

render : function() {

var content= Mustache.render(this.template, this.model.toJSON());

this.$el.html(content) ;

},

//Méthode pour faire une petite manipulation sur le DOM

aMethod : function(text) {

var $myDiv = this.$el.find(‘.aDiv’) ;

$myDiv.text(text) ;

}

}) ;

Pour gérer les événements du DOM, on peut également les lister dans l’attribut « events » de la vue et on leur associe une fonction comme ceci :

events : {

‘click button’ : ‘sayHello’,

‘change input’ : ‘inputChanged’,

‘click .myClass’ : ‘classClicked’

}

### Les routeurs

Les routeurs permettent d’intercepter les urls du navigateur et de les traiter en fonction. Avec l’API History de l’HTML5, il est possible d’afficher des url différentes tout en ayant un seul et unique fichier HTML.

Les fonctions liées à ces urls permettent souvent de changer le contenu affiché dans la page. Les routeurs secondent donc en quelque sorte les vues pour le « contrôle » de l’application.

var myRouter = Backbone.Router.extend({

routes : {

‘hello’ : ‘sayHello’,

‘hello/:name’ : ‘sayHello’

‘\*’ : ‘default’

},

//sayHello récupère le paramètre name indiqué par :name dans l’url

sayHello : function(name) {

if(name) {

console.log(‘Hello ‘ + name + ’ !’) ;

} else {

console.log(‘Hello anonymous !’) ;

}

}

}) ;

Pour que les urls soient interceptées par le routeur Backbone, il faut, une fois que le routeur a été instancié, lancer l’historique de Backbone avec Backbone.history.start() ; ou Backbone.history.start({pushState : true} ) ;

## Le templating

Le templating consiste à créer des bouts de code HTML qui seront utilisés par les vues Backbone. Ces fichiers contiennent donc du code HTML mais aussi des annotations spécifiques pour afficher du contenu envoyé par les vues Backbone (typiquement des objets au format JSON représentant des modèles).

Il existe plusieurs outils de templating. Celui par défaut est EJS. Pour afficher un élement du JSON fourni au template, il faut le mettre entre les balises <%= et %>. Et pour « compiler » un tel template avec des données, il faut appeler la fonction \_.template( leCodeDuTemplate, lesDonnéesEnJSON)

L’inconvénient de l’EJS est qu’il a tendance à alourdir le code dès que l’on doit itérer sur une liste ou lorsque l’on doit faire des tests car il faut injecter du code JavaScript dans le fichier, ce qui engendre des avertissements lors de la compilation.

C’est pour cela qu’ont été créés Mustache et Handlebars (ce dernier repose sur Mustache et lui ajoute quelques méthodes). Au lieu d’avoir des balises <%= %> on utilise des accolades {{ }}, mais surtout, il est possible d’itérer sur un tableau en faisant seulement {{#tableau}} {{uneValeur}} {{/tableau}}. Les tests fonctionnent sur le même principe du # et /.

Pour générer le template avec le contenu, il faut écrire Mustache.render(texteDuTemplate, lesDonnéesEnJSON) ;

Pour plus d’informations sur Mustache en JavaScript, vous pouvez vous rendre sur le [dépôt Github du projet](https://github.com/janl/mustache.js/).

Une bonne pratique concernant les templates consiste à créer un objet global JST qui contiendra tous les templates compilés. Cet objet JST est créé par défaut par Grunt que ce soit en développement ou lors du build. On définit alors le template d’une vue Backbone par :

template : JST[‘nom-du-template’]

où ‘nom-du-template’ est le nom du fichier du template en règle général (c’est le cas dans le projet de capitalisation)

La méthode render n’a alors pas à être changée.

Pour mieux comprendre l’intérêt de cette pratique, vous pouvez lire cet [article](http://ricostacruz.com/backbone-patterns/#inline_templates) qui fait la différence entre les templates « en ligne » et les templates « compilés ».

## Backbone.Marionette

### Pourquoi Marionette ?

Backbone apporte une certaine structure dans le code JavaScript. Cependant, il possède quelques défauts. Par exemple, lorsque l’on remplace une vue par une autre, il est nécessaire de la supprimer « proprement », c'est-à-dire de faire un destroy dessus pour éliminer tous les événements associés et libérer de l’espace mémoire. Autre exemple : lorsque l’on veut afficher une collection de modèles, la meilleure pratique consiste à créer une vue associée à un modèle et une vue associée à la collection qui fera un render de chaque élément de la collection dans sa propre méthode render. Mais il faut aussi gérer des événements liés à des changements des modèles, des ajouts ou des suppressions dans la collection etc. On a donc l’impression de réinventer la roue à chaque fois pour coder toutes ces vues.

Marionette permet de réduire la taille du code tout en prenant en compte ces cas. Ainsi, des classes spécifiques ont été écrites pour afficher un modèle ou un élément (ItemView), une collection (CollectionView), pour afficher des vues contenant des élements (Layout, CompositeView) etc. Une classe application qui constitue le cœur de l’application a été créée, des callbacks sur les vues on été ajoutés (avant, pendant et après render), une classe contrôleur a été implémentée, une classe chargée de faire le rendu a été programmée etc.

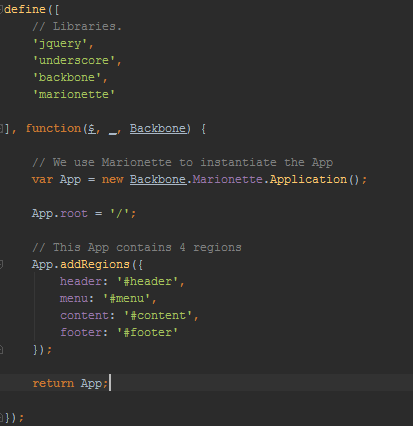
Marionette est donc adapté à des front-end de plus grande complexité.

Dans la suite de cette partie, nous allons détailler quelques éléments principaux de Marionette qui facilitent le développement. La documentation de Marionette est disponible à cette [adresse](https://github.com/marionettejs/backbone.marionette).

### Les principaux ajouts de Marionette

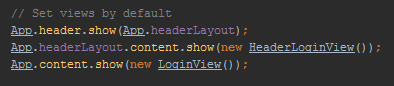
* Marionette.Application

C’est le « cœur » de notre application. C’est en fait la classe qui va permettre de démarrer notre application, d’y créer des régions dans lesquelles nous y placerons nos vues etc.



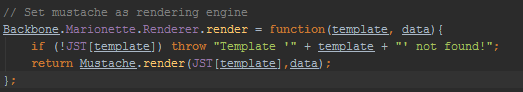
* Marionette.Region

C’est une classe qui permet de gérer des parties visuelles de l’application. On peut par exemple définir que l’application contient 4 régions qui sont le header, un menu, le contenu et un footer. Pour afficher une vue particulière dans une région, il suffit alors d’écrire maRegion.show(maVue). Pour la masquer, maRegion.close(). Et pour remplacer le contenu d’une région sans se soucier de savoir s’il y avait quelque chose avant ou pas, maRgion.show(maNouvelleVue). Marionette gère les destructions des vues, la suppression des événements associés et l’affichage de la nouvelle vue.



* Marionette.Renderer

Cette classe est appelée pour afficher les vues Marionette. Il suffit donc de la modifier pour l’adapter à son outil de templating. Avec Mustache, on obtient ainsi le code suivant :

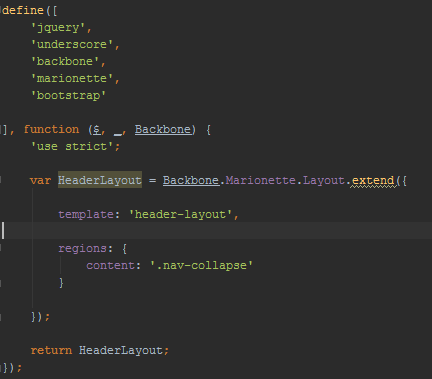


* Marionette.View

Cette classe hérite de la classe vue de Backbone et lui ajoute, entre autres, des callbacks utiles tels que onBeforeRender qui permet d’effectuer des traitements juste avant le rendu, onRender qui effectue des modifications pendant le rendu et onAfterRender qui est appelé une fois le rendu effectué. Cette classe n’est pas destinée à être utilisée directement. Il est conseillé d’utiliser ItemView, CompositeView ou CollectionView qui héritent de cette classe.

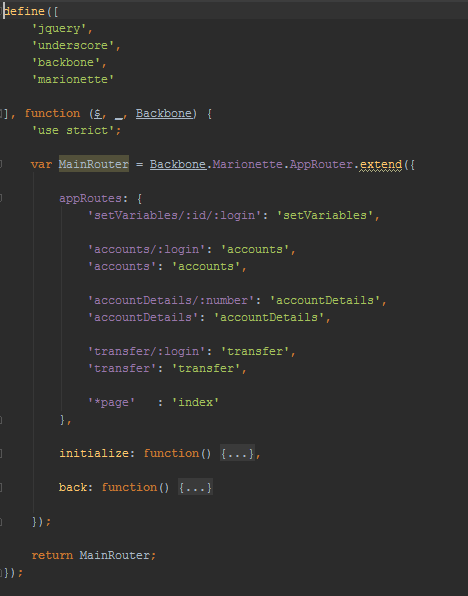
* Marionette.Layout

Un Layout permet de créer des zones, des régions dans le template qui seront utilisées pour afficher du contenu selon les vues. Une Region peut contenir un Layout qui contient lui-même des Regions.



* Marionette.AppRouter

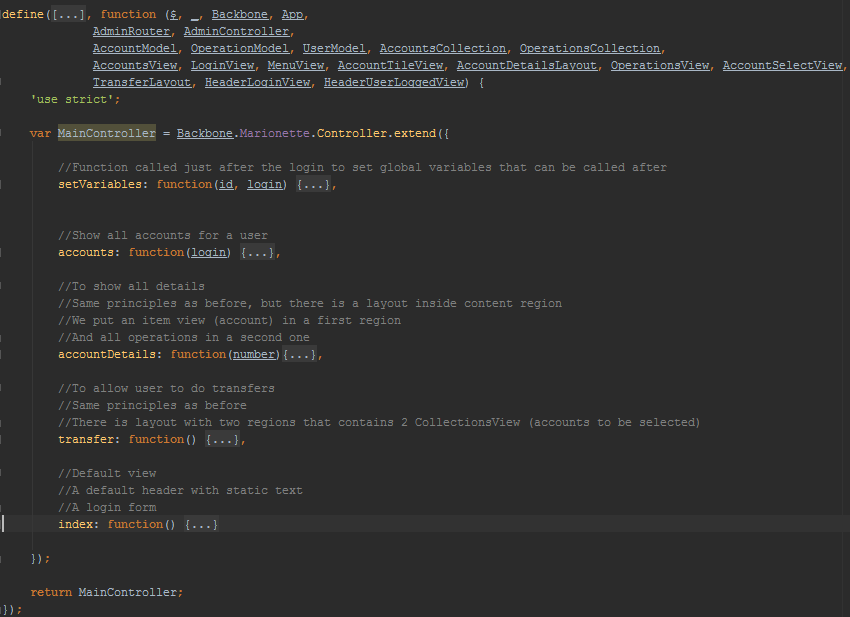
AppRouter est l’équivalent du Router de Backbone. La différence réside dans le fait que les méthodes associées à une url sont des méthodes du contrôleur associé à ce routeur. Ainsi, dans cette class, on ne se soucie que des urls.



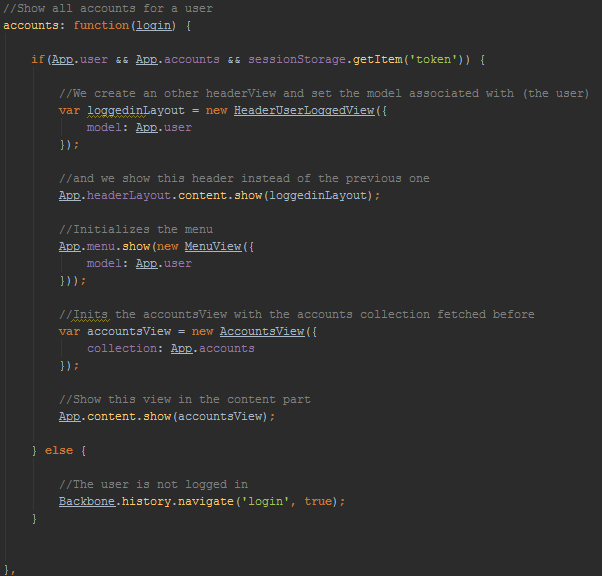
* Marionette.Controller

Le Controller correspond à ce que l’on attend dans un framework MVC, c’est lui qui va indiquer quelle vue doit être affichée à quel endroit etc.

La capture d’écran suivante correspond au contrôleur associé au routeur précédent.

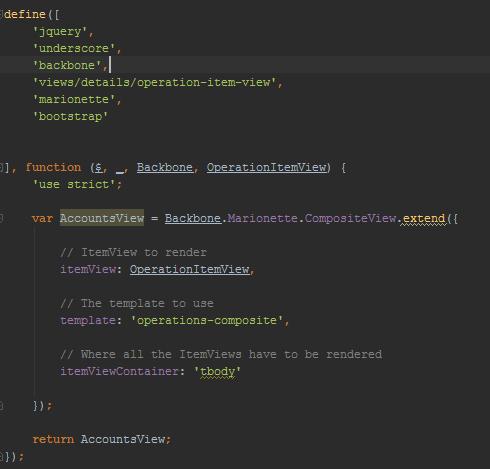


Et pour une fonction en particulier :

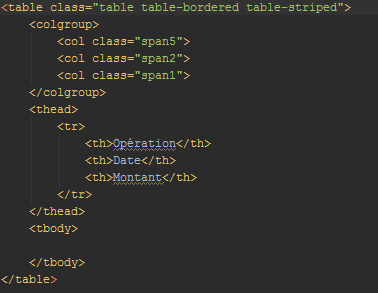


* Marionette.CompositeView

Une vue composite est une vue qui contient un template de base dans lequel on va ajouter soit un seul élément soit une collection. Si, lors de son initialisation, on lui associe une collection, Marionette va automatiquement itérer sur celle-ci pour afficher la vue associée à chaque modèle (ItemView à donner en paramètre)

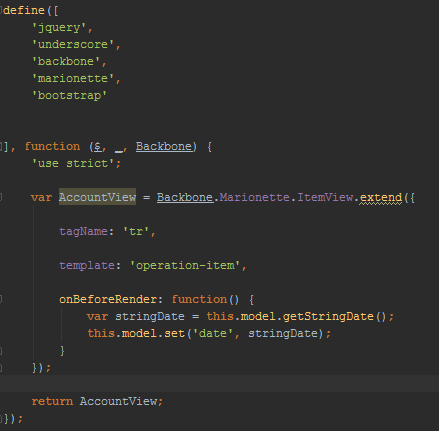


Et le template associé à cette vue est le suivant :



* Marionette.ItemView

Une ItemView est la vue associée à un modèle. Pour son rendu et sa génération, il faut qu’elle ait un template associé, une façon de donner un rendu de l’élément (pour cela elle se base du le Marionette.Renderer) et le modèle à afficher.



Et le template associé est alors très simple :

C:\Users\mduclos\AppData\Local\Temp\ScreenClip.png

Il existe également d’autres ajouts qui peuvent être utiles. Vous les trouverez dans la documentation de Marionette.

# Exemple d’un projet

## Récupération et démarrage de l’application

### Récupération des sources

Le projet se trouve ………………………….

Il faut donc récupérer les sources comme pour un projet habituel. Placez ce projet à l’endroit de votre choix. Dans cet exemple, le projet est situé dans le dossier D:\wamp\www\banqueSQLI.

Pour faire fonctionner cette application, nous utilisons un back-end développé en NodeJS qui peut être récupéré ………………………………………….. Nous avons placé ce projet dans le dossier D:\wamp\www\banquesqli-back.

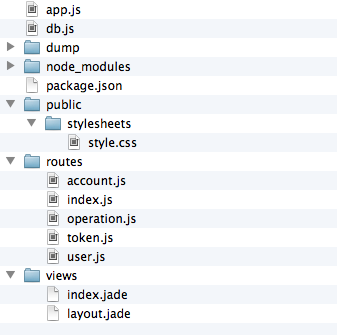
Pour la suite des manipulations, vous devez avoir NodeJS, Yeoman, Grunt et Bower d’installés et fonctionnels. Il faut également installer une base de données MongoDB et la lancer. Des données peuvent être importées depuis le dump fourni avec le code du back-end.

### Installation de l’application

Pour faire fonctionner ces deux projets, il faut d’abord récupérer les sources qui ne sont pas dans le gestionnaire de source avec NPM et Bower.

**Back-end :**

Pour le back-end, ouvrez une console et placez-vous dans le dossier D:\wamp\www\banquesqli-back (dossier contenant le code du back-end), puis lancez la commande « **npm install**». NPM va alors récupérer de nombreuses sources et les placer dans un dossier node\_modules directement sous le dossier banquesqli-back. On obtient alors l’arborescence suivante :



Le dossier node\_modules contient tous les composants Nodes nécessaires à l’application pour être lancée sur le serveur NodeJS.

Le dossier dump contient des données pour initialiser la base de données.

Les dossiers public et views ne seront pas utilisés dans notre exemple. Ils ont été générés par Express (framework pour NodeJS) et servent à faire une application servie par le serveur.

Le dossier routes contient les fichiers qui nous servent de modules et qui vont définir les réponses envoyées par le serveur lorsque l’on appelle certaines url. Les noms des fichiers correspondent aux entités stockées en base de données.

Le fichier db.js sert à indiquer un schéma de base de données et à définir les modèles.

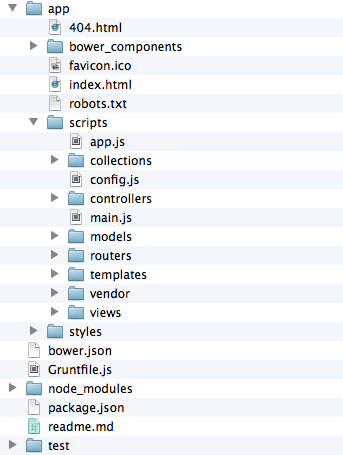
Le fichier app.js initialise le serveur Express et définit toutes les routes (URL) qui pourront être appelées, les middlewares pour l’authentification et les actions qui correspondent à chaque demande.

**Front end :**

En ce qui concerne le front-end, placez-vous dans le dossier contenant le code de l’application front-end (D:\wamp\www\banqueSQLI chez moi) dans une console. Et lancez la commande suivante : « **npm install**». De même que pour le back-end, NPM va récupérer un certain nombre de ressources et créer un dossier node\_modules situé directement sous le dossier banqueSQLI.

Une fois cette commande terminée, nous allons récupérer les librairies nécessaires pour faire fonctionner l’application. Pour cela, nous allons utiliser Bower et taper dans la console (toujours dans le dossier de banqueSQLI) « **bower install**». Bower va récupérer un certain nombre de ressources et peut vous demander de choisir lesquelles prendre lorsque plusieurs librairies ont des composants en commun. Par exemple, lors de l’installation sur mon ordinateur personnel, il m’a été demandé de choisir la version de jQuery que je souhaitais utiliser.

Une fois les sources récupérées et ces deux commandes réalisées, vous devriez obtenir l’arborescence suivante :



À la racine du dossier, on trouve le dossier node\_modules comme expliqué auparavant, un fichier Gruntfile qui définit les tâches pour Grunt, un fichier bower.json qui liste les librairies utilisées, un fichier package.json sui liste les composants Node nécessaires pour faire tourner l’application avec Grunt, un dossier test qui contient quelques tests unitaires de l’application en JavaScript et un dossier app qui contient le code faisant tourner notre webapp.

Le dossier app contient la page d’index en HTML (qui ne contient que très peu de code puisque l’affichage se fait en passant par le JavaScript), une page d’erreur 404, une favicon, un fichier pour les robots, un fichier htaccess (non visible ici), un dossier contenant les feuilles de style en SASS, un dossier bower\_components, avec toutes les librairies importées par Bower dans notre application, et un dossier scripts qui contient le code JavaScript et les templates de l’application. Les explications sur ce dossier sont données dans la suite de ce document.

### Lancement de l’application

Pour démarrer le serveur NodeJS, il faut se positionner dans le dossier banquesqli-back et lancer la commande « **node app** ». Si tout se passe bien (ce qui devrait être le cas ;) ) la console affiche « Express server listening on port 3000 »

Ensuite, dans une nouvelle fenêtre de console, placez-vous dans le dossier banqueSQLI et tapez la commande « **grunt server**». Si l’installation a été faite correctement, une fenêtre ou un nouvel onglet du navigateur web par défaut s’ouvre et affiche la page d’accueil de l’application sur l’url « localhost:9000 ». La page par défaut est un formulaire de login

## Explications sur le code Front-end

### Les fichiers qui initialisent l’application

Index.html

Lorsque Grunt server est lancé, un serveur local est démarré et affiche la page d’index sous le dossier app. Dans ce fichier, il y a surtout une ligne qui est importante, c’est celle qui fait appel à RequireJS qui lui-même va charger le fichier spécifié dans l’attribut data-main. 

Il est à noter que les commentaires qui encadrent cette balise indiquent à Grunt quoi builder lors de la phase de compilation du code.

Scripts/config.js

Ce fichier sert à configurer les chemins des librairies utilisées dans notre application. Nous retrouvons donc les chemins vers jQuery, Backbone, Underscore, Mustache ou Marionette. De plus, comme Backbone et Underscore ne sont pas entièrement compatibles avec AMD, nous définissons leurs dépendances envers les autres librairies et comment ils sont nommés dans d’autres paquets.

Nous avons également ajouté la ligne  pour indiquer que le fichier main.js est à charger une fois la configuration terminée.

Scripts/main.js

C’est ce fichier qui est appelé une fois la configuration de RequireJS terminée. C’est donc lui qui va réellement faire appel à tous les fichiers JavaScript dont nous avons besoin.

Il fait appel, entre autres, au router principal qu’il va initialiser avec le contrôleur principal, au fichier app.js qui sert à instancier un objet App de la classe Marionette.Application et aux vues affichées lors du lancement de l’application (l’en tête et le formulaire de login) .

De plus, nous avons ajouté une méthode pour que les liens avec un dièse soient interprétés par les routeurs, nous avons réimplémenté la méthode open de la classe Marionette.Renderer pour donner un effet lors de l’affichage d’une région, et la méthode render pour utiliser Mustache pour le rendu des pages.

### Les contrôleurs de l’application

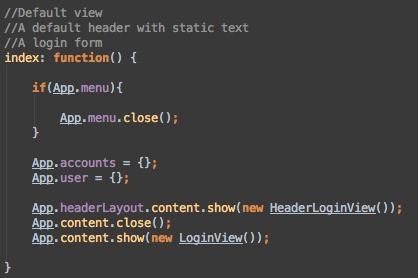
Les contrôleurs sont situés dans le dossier scripts/controllers. Nous en avons créé deux : un principal qui gère l’affichage commun (consultation des comptes et des opérations, transfert, login, header) et un autre qui gère l’affichage de l’administration.

Ces contrôleurs sont chacun associés à un routeur : MainController est associé à MainRouter et AdminController à AdminRouter.

Dans les routeurs, nous avons ajouté des méthodes qui permettent de simuler en JavaScript un retour en arrière.

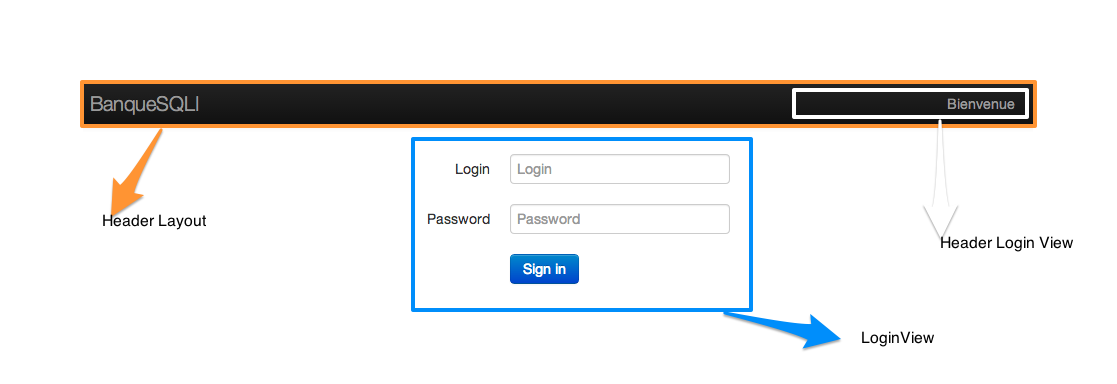
Les contrôleurs nécessitent de nombreux fichiers car ce sont eux qui initialisent les vues en fonction de l’URL appelée. Les traitements des événements utilisateurs autres que les changements d’URL sont gérés par les vues directement et non par les contrôleurs.

Dans notre cas, lorsque l’application est lancée, nous appelons l’url <http://localhost:9000/>. Cette URL ne correspond qu’au pattern ‘\*’ donné dans MainRouter qui appelle la méthode index du MainController.



Cette méthode n’affiche plus le menu s’il était visible, vide des collections qui auraient pu être initialisées auparavant, affiche une nouvelle vue (HeaderLoginView) dans la région « content » de la région « header » de l’application, ferme ce qu’il y avait dans la région « content » et y affiche une nouvelle vue pour se logger (LoginView).

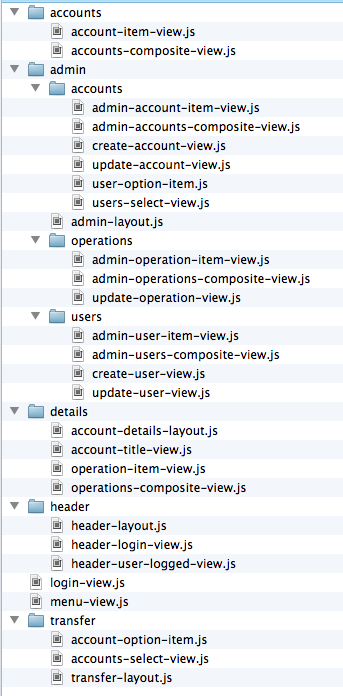
On retrouve ainsi la structure suivante :



Et il en va de même avec les autres routes. Il en existe certaines qui ne servent pas à afficher des éléments mais à initialiser les modèles et les collections. C’est le cas de la méthode setVariables du MainController qui sert uniquement à récupérer les informations concernant l’utilisateur, les comptes qui lui sont associés et à tester s’il est administrateur. Dans ce cas là, cette méthode initialise un autre routeur auquel est associé un autre contrôleur (partie administration avec AdminRouter et AdminController).

### Les vues

Toutes les vues se trouvent dans le dossier scripts/views. Nous avons décidé, sous ce dossier, d’adopter une arborescence en fonction des pages affichées. Nous aurions très bien pu adopter une architecture avec des dossiers en fonction des types de vues (Layout, CompositeView, CollectionView, ItemView etc.). Comme la page de login et le menu ne sont constitués que d’une seule vue, les fichiers qui leur correspondent sont situés à la racine du dossier views. C’est pourquoi nous obtenons l’arborescence suivante :

La vue d’admin étant un Layout, nous avons choisi d’ajouter des dossiers qui correspondent à chaque vue de cette partie dans un sous dossier.

Pour ce qui est des templates, ceux ci sont placés dans le dossier templates (sous le dossier script). Nous lui avons donné la même architecture qu’aux vues tout en essayant d’avoir des noms similaires pour s’y retrouver.

Avec la méthode utilisée pour générer les templates, à chaque fois que l’on crée un nouveau fichier mustache ou qu’en on modifie un, il est nécessaire de relancer Grunt pour que celui-ci prenne en compte le changement. Cela est du au fait que la compilation des templates se fait au lancement de l’instance de Grunt et pas lors du rechargement d’une page.

Nous allons maintenant présenter les principales vues de l’application.

#### **Le layout de l’application**

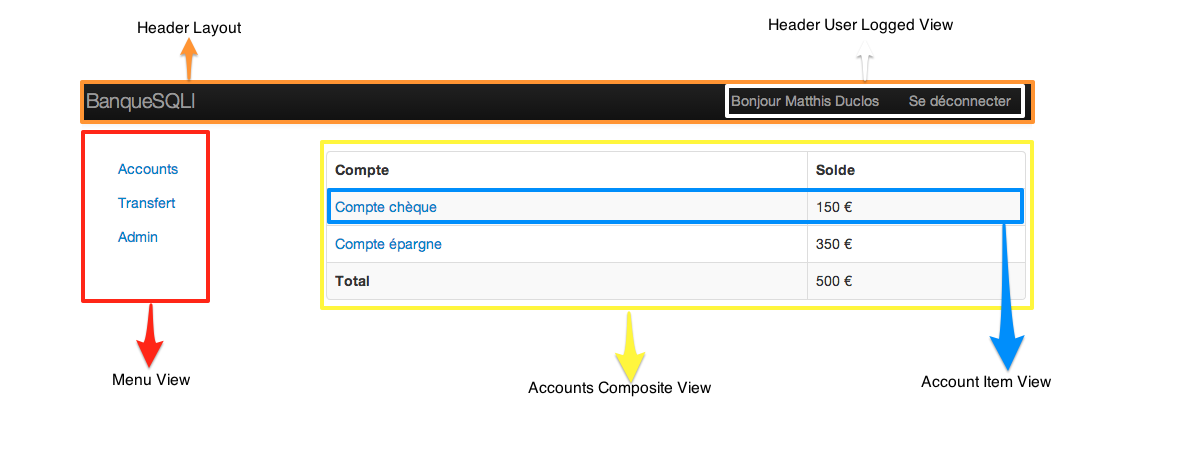
L’affichage de l’application est divisé en 3 régions : l’en-tête (header), le menu à gauche et le contenu (content) au centre.

Le menu n’est affiché qu’une fois l’utilisateur connecté et affiche le lien pour l’administration si l’utilisateur est un admin. Cette vue est un ItemView car elle affiche des liens en fonction de l'usager.

Le header est affiché en permanence mais une partie de son contenu est modifié une fois le client connecté.

La région « content » est modifiée en fonction de l’url. Elle est affichée en permanence mais son contenu est rarement le même d’une url à l’autre.

On obtient donc la structure suivante :



La partie en jaune est la région « content ». Ici elle affiche la vue AccountsCompositeView. On retrouve le fait que le header contient lui même une région.

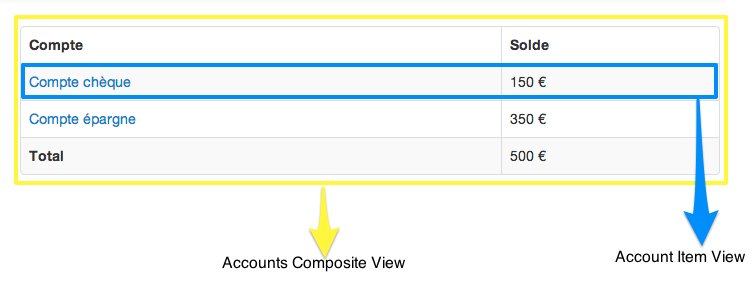
#### **La vue de login**

Cette vue est un ItemView mais auquel on ne passe aucun modèle. Marionette ne fait qu’afficher le template qui lui est associé. On a lié un événement au clic sur le bouton de login en bleu. Ce clic entraîne l’appel de la méthode formSubmitted qui fait un POST vers le serveur en jQuery. En fonction de la réponse donnée, on est redirigé vers la page d’accueil pour l’utilisateur (affichage des comptes) ou on affiche un message d’erreur.

Le template associé à cette vue est celui qui porte le nom login.mustache. Pour l’indiquer dans le code, on n’écrit que le nom du fichier mustache sans son extension, soit 

#### **La vue des comptes utilisateurs**

Cette vue est plus complexe que la précédente. Elle est affichée dans la région « content » de l’application. Elle est composée d’une AccountsCompositeView qui sert à afficher l’en-tête du tableau et son pied ainsi qu’une collection de comptes qui, quant à eux, sont chacun représentés par une AccountItemView.



L’affichage du pied du tableau est réalisé grâce au callback « onCompositeCollectionRendered » fourni par Marionette. Ce callback est appelé une fois que tous les éléments de la collection passée en paramètre de cette vue ont été affichés.

Pour définir quel type d’élément est à afficher dans cette CompositeView, on indique l’attribut itemView dans le code. Pour dire à quel endroit afficher la collection, on ajoute l’attribut itemViewContainer. Enfin, le template est défini par le nom du fichier mustache associé (sans son extension). C’est dans le contrôleur principal, dans la méthode « accounts » que l’on donne la collection de compte à afficher à cette vue.

Pour la vue d’un compte (AccountItemView), on indique que la vue sera encadrée par une balise « tr » (ligne de tableau), qu’elle aura la classe « account » et que son template est le fichier account-item.mustache. On précise également qu’en cas de changement du modèle, on doit réafficher cette vue.

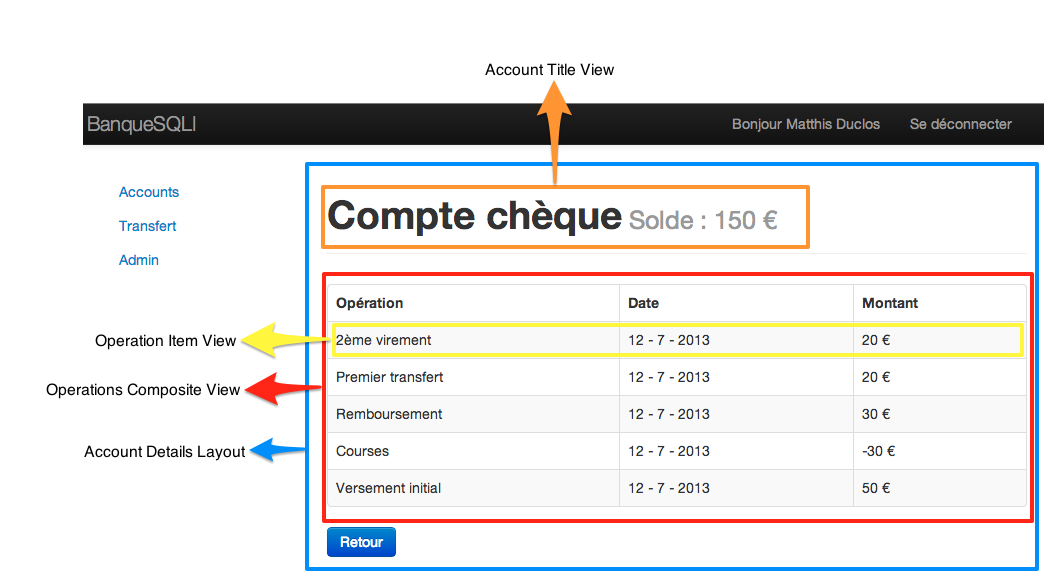
#### **La vue du détail d’un compte**

Lorsque l’utilisateur clique sur un compte, il est redirigé vers la page de détail pour ce compte. Cette page fonctionne quasiment sur le même principe que la précédente.

La seule différence réside dans le fait que nous utilisons un Layout pour afficher an haut le nom du compte et en-dessous la liste des opérations effectuées sur ce compte.

Pour le titre, il s’agit d’un ItemView (AccountTitleView) et pour la liste des opérations, c’est une vue composite (OperationsCompositeView) dans laquelle sont affichées les vues des opérations (OperationItemView).

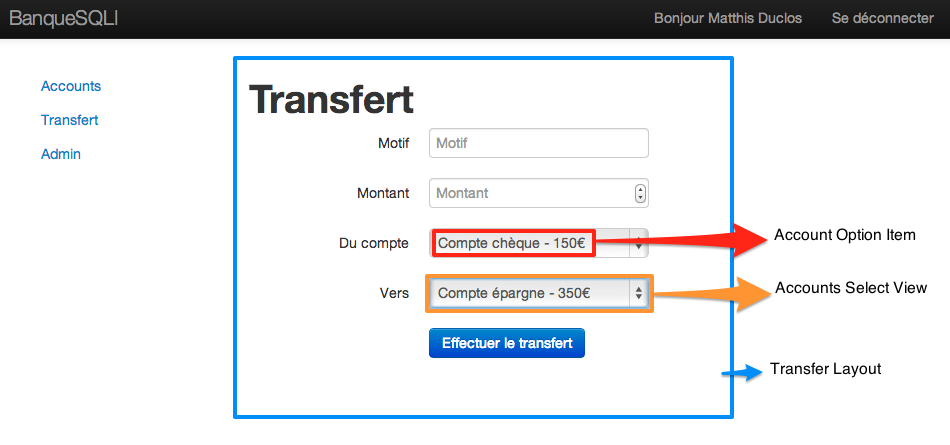
On se retrouve donc avec l’organisation suivante :



On retrouve les mêmes éléments de code quand la partie précédente. L’affichage de cette page est géré par le contrôleur principal qui passe en paramètre du titre le modèle du compte et en paramètre de la vue composite la collection d’opérations.

#### **La vue de transfert entre comptes**

Cette vue est légèrement structurée différemment puisque c’est un layout dans lequel on affiche deux collections de comptes (une pour le choix du compte à débiter et une autre pour le choix du compte à créditer). Ces collections sont deux instances de la classe AccountsSelectView.



Cette classe est donc une extension CollectionView. Elle n’a pas besoin de template car elle est composée d’un ensemble de AccountOptionView (ItemView pour réprésenter les options du select du formulaire). On lui donne seulement la collection sur laquelle elle doit itérer, le type d’élément à afficher, la balise dans lequel wrapper la collection.

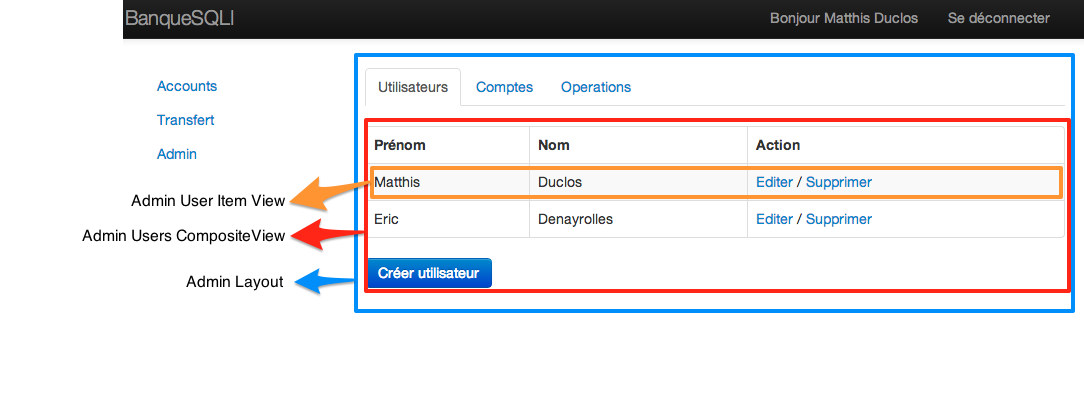
Pour les AccountOptionView, il s’agit d’ItemView. Nous leur avons donc donné une valeur selon le modèle et la balise dans laquelle le template est affiché.

Au TransferLayout, nous avons ajouté un événement lorsque le formulaire est soumis. La méthode appelée est fomSubmitted. Cette méthode à pour but de valider le formulaire à l’aide de parsley.js, de vérifier que les comptes choisis sont bien différents pour de sauvegarder la manipulation effectuée (ajout d’argent sur un compte, retrait sur un autre, création d’une opération de débit sur le premier compte et création d’une opération de crédit sur l’autre compte choisi). Pour cela, nous créons deux objets opérations qui sont envoyés au serveur à l’aide de la méthode save de Backbone. Pour que le POST soit valide côté serveur, on ajoute le token en paramètre de cette fonction. De même, en faisant des « set » sur les objets « account », puis en appellant la méthode « save », Backbone effectue une requête en PUT sur le serveur.

Grâce à la méthode « save » des modèles, il n’est pas nécessaire d’écrire entièrement les POST ou PUT en ajax avec jQuery puisque Backbone s’en charge pour nous.

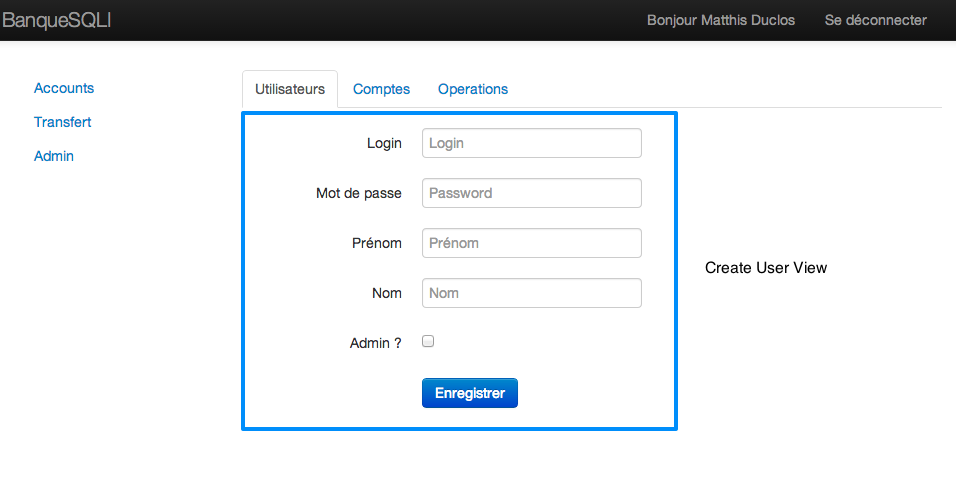
#### **La partie administration**

Cette partie est peut-être la plus complexe de l’application car c’est un layout qui est composé de 3 régions qui est lui même affiché dans une régions et car ces 3 régions ne sont jamais visibles en même temps en raison de l’utilisation d’onglets (grâce au JavaScript de Bootstrap). De plus, cette partie permet de faire du CRUD avec les utilisateurs et les comptes.



Si on prend chaque région séparément, le fonctionnement reste le même que celui vu auparavant : on affiche du contenu dans des régions, ce contenu étant soit un ItemView si on n’affiche qu’un seul élément, soit une CompositeView si on affiche une collection avec du contenu autour, la collection étant un ensemble d’ItemView.

Pour le formulaire d’ajout d’un utilisateur, on affiche des ItemView sans modèle de base, tandis que pour le formulaire d’édition, on affiche un itemView avec en paramètre l’objet utilisateur à modifier. Et comme pour le transfert, on attrape l’événement de soumission du formulaire, on le valide avec parsley et des méthodes personnelles puis on utilise la méthode « save » de Backbone.



Pour le formulaire d’ajout d’un compte, on crée un Layout avec une région qui sert à afficher une liste d’utilisateurs pouvant posséder ce compte (on reprend le même principe que celui utilisé pour les transferts avec les listes de comptes).

