

# Contents

<b>1</b>	<b>Préambule &amp; Remerciements</b>	<b>1</b>
1.1	Avertissement . . . . .	1
<b>2</b>	<b>Présentation de Backbone &amp; rappels MVC</b>	<b>2</b>
2.1	Backbone ? Webapps ? MVC ? . . . . .	2
2.1.1	Qu'est-ce qu'une "Webapp" ? . . . . .	2
2.1.2	Petit rappel : MVC ? . . . . .	3
2.2	Backbone & MVC . . . . .	4
2.3	Pourquoi j'ai choisi Backbone ? . . . . .	6
<b>3</b>	<b>Tout de suite "les mains dans le cambouis"</b>	<b>6</b>
3.1	Prérequis : les dépendances de Backbone . . . . .	6
3.2	Outils de développement . . . . .	7
3.2.1	IDE (Editeur) . . . . .	7
3.2.2	Navigateur . . . . .	7
3.3	Initialisation de notre projet de travail . . . . .	7
3.3.1	Installation . . . . .	8
3.3.2	Préparons notre page HTML . . . . .	8
3.4	Jouons avec jQuery . . . . .	9
3.4.1	"Jouons" avec notre page en mode commande . . . . .	12
3.4.2	Les événements . . . . .	18
3.4.3	Quelques bonnes pratiques . . . . .	18
3.5	Jouons avec Underscore . . . . .	20
3.5.1	Quelques exemples d'utilisations . . . . .	20
<b>4</b>	<b>1er contact ... avec Backbone</b>	<b>24</b>
4.1	1ère application Backbone . . . . .	24
4.1.1	Préparons notre page . . . . .	25
4.2	Le Modèle "Article" . . . . .	26
4.3	La Collection d'Articles . . . . .	28
4.4	Vue et Template . . . . .	29
4.4.1	Qu'avons-nous fait ? . . . . .	31
4.5	Un dernier tour de magie pour clôturer le chapitre d'initiation : "binding" . . . . .	32
4.5.1	Que venons-nous de faire ? . . . . .	33
4.5.2	Oh la vilaine erreur !!! . . . . .	33
4.6	Code final de l'exemple . . . . .	34

<b>5 Le modèle objet de Backbone</b>	<b>37</b>
5.1 Un petit tour dans le code . . . . .	38
5.2 1ère “classe” . . . . .	38
5.2.1 Un constructeur . . . . .	39
5.2.2 Des propriétés . . . . .	39
5.2.3 Des méthodes . . . . .	39
5.2.4 Des membres statiques . . . . .	40
5.3 Sans héritage point de salut ! ... ? . . . . .	41
5.4 Surcharge & super . . . . .	42
5.5 Conclusion . . . . .	43
<b>6 Il nous faut un serveur !</b>	<b>43</b>
6.1 Principes http : GET, POST, PUT, DELETE . . . . .	44
6.2 Installation(s) . . . . .	44
6.2.1 Installer Node.js . . . . .	44
6.2.2 Installer Express.js . . . . .	45
6.2.3 Installer nStore . . . . .	45
6.3 Codons notre application serveur . . . . .	45
6.3.1 Ressources statiques . . . . .	46
6.3.2 Ressources dynamiques . . . . .	47
6.4 Testons notre application serveur . . . . .	51
6.4.1 Ajoutons un enregistrement . . . . .	51
6.4.2 Obtenir tous les enregistrements . . . . .	52
6.4.3 Retrouver un enregistrement particulier (par sa clé) . . . . .	53
6.4.4 Mettre à jour un enregistrement . . . . .	54
6.4.5 Faire une requête . . . . .	55
6.4.6 Supprimer un enregistrement . . . . .	56
<b>7 Les modèles et les collections en détail</b>	<b>57</b>
7.1 Fonctionnement général . . . . .	58
7.2 Modèles . . . . .	58
7.2.1 Définition du modèle . . . . .	58
7.2.2 Getters et Setters . . . . .	59
7.2.3 Structure d'un modèle . . . . .	60
7.2.4 Méthodes “CRUD” du modèle . . . . .	61
7.2.5 Evénements . . . . .	65
7.2.6 Constructeur : initialize . . . . .	67

7.2.7	“Augmenter” le modèle : ajouter des valeurs par défaut et des méthodes au modèle	68
7.2.8	Comment détecter qu’un modèle a été changé par quelqu’un d’autre ?	69
7.3	Collections	70
7.3.1	Comment ajouter des modèles à une collection	71
7.3.2	Parcourir les modèles d’une collection	72
7.3.3	Filtrer les modèles d’une collection	73
7.3.4	Trouver le 1er modèle d’une collection correspondant à un critère	74
7.3.5	Autres méthodes de la collection	74
7.4	Les collections “parlent” au serveur	74
7.4.1	Charger les données	75
7.4.2	Requêtes	76
7.5	Événements	78
<b>8</b>	<b>Vues &amp; Templating</b>	<b>79</b>
8.1	Préparons le terrain	80
8.1.1	Création et sauvegarde des modèles	80
8.2	1ère vue	84
8.2.1	Explications & utilisation	85
8.3	Maintenant, un peu de magie ...	87
8.3.1	S’abonner aux événements	87
8.3.2	S’abonner à d’autres évènements (modèles)	89
8.3.3	Amélioration & Finalisation du code	90
8.4	Utilisation du templating ... 1ère fois	92
8.4.1	Définition de notre 1er template	92
8.5	Sous-vue(s)	95
8.5.1	Réorganisation du code html	95
8.5.2	Création & Modification des vues	96
8.5.3	Un dernier petit réglage : tri des collections	98
8.6	Utilisation d’autre(s) moteur(s) de template	100
8.6.1	Redéfinissons donc nos templates	101
8.7	Gestion des événements dans les vues	103
8.8	Authentification (côté serveur) : les utilisateurs	103
8.8.1	S’authentifier – Se déconnecter	104
8.9	Authentification (côté client)	113
8.9.1	Formulaire d’authentification	113
8.9.2	L’objet Backbone.View : Login.View	114
8.9.3	Ajoutons une gestion des évènements	115
8.9.4	Vérification	117

---

<b>9 Le Routeur</b>	<b>120</b>
9.1 Modifions notre vue . . . . .	120
9.2 Création du routeur . . . . .	121
<b>10 Organiser son code</b>	<b>124</b>
10.1 “A l’ancienne” . . . . .	124
10.1.1 Namespace . . . . .	124
10.1.2 Modules . . . . .	127
10.1.3 Au final, nous aurons ... . . . . .	128
10.2 Méthode “hype”, comme les vrais . . . . .	133
10.2.1 Préparation . . . . .	134
<b>11 Sécurisation</b>	<b>137</b>
11.1 Il nous faut un “écran d’administration” . . . . .	137
11.1.1 Création d’AdminView . . . . .	140
11.1.2 Modification de LoginView . . . . .	141
11.2 Sécurisation côté serveur . . . . .	143
11.2.1 Sécurisation des routes . . . . .	143
11.2.2 Eh bien testons, maintenant . . . . .	145
<b>12 Backbone.sync()</b>	<b>149</b>
<b>13 Backbone &lt;3 Coffeescript</b>	<b>149</b>
13.1 Coffeescript qu’est-ce que c’est ? . . . . .	149
13.2 Un code plus lisible ? . . . . .	149
13.2.1 Les fonctions . . . . .	150
13.2.2 Interopérabilité . . . . .	150
13.2.3 Interpolation et Chaînes de caractères . . . . .	151
13.2.4 Jouez un peu avec les tableaux . . . . .	151
13.3 Et enfin (et surtout ?) les classes . . . . .	152
13.3.1 Notre 1ère classe . . . . .	152
13.3.2 Propriétés & valeurs par défaut . . . . .	152
13.3.3 Comme en Java : Composition, Association, Encapsulation . . . . .	153
13.3.4 Comme en Java : les membres statiques . . . . .	153
13.3.5 Mais aussi (comme en Java) : l’héritage ! . . . . .	154
13.4 J’aime / Je n’aime pas . . . . .	154
13.5 Ré-écriture de notre blog ?! S’outiller . . . . .	155
13.5.1 Installation de Coffeescript . . . . .	155

13.5.2 “Industrialisation” de la transpilation . . . . .	156
<b>13.6 Ré-écriture / Traductions . . . . .</b>	<b>157</b>
13.6.1 Blog.coffee . . . . .	157
13.6.2 main.coffee . . . . .	157
13.6.3 post.coffee . . . . .	158
13.6.4 AdminView.coffee . . . . .	158
13.6.5 LoginView.coffee . . . . .	160
13.6.6 PostsListView.coffee . . . . .	161
13.6.7 PostView.coffee . . . . .	162
13.6.8 SidebarView.coffee . . . . .	162
13.6.9 MainView.coffee . . . . .	162
13.6.10 routes.coffee . . . . .	163
13.6.11 Transpilons . . . . .	163
13.6.12 Conclusion(s) . . . . .	163
<b>14 Autres Frameworks MVC</b>	<b>164</b>
14.1 CanJS . . . . .	164
14.1.1 Mise en oeuvre rapide . . . . .	165
14.1.2 Conclusion sur CanJS . . . . .	167
14.2 Spine . . . . .	168
14.2.1 Mise en oeuvre rapide . . . . .	169
14.2.2 Conclusion sur Spine . . . . .	172
14.3 Knockout . . . . .	172
14.3.1 Mise en oeuvre très très rapide ... Mais suffisante . . . . .	173
14.3.2 Un modèle selon Knockout . . . . .	173
14.3.3 Collections ? . . . . .	174
14.3.4 Attachons notre modèle à des vues ! . . . . .	174
14.3.5 Attachons note liste de messages à une vue . . . . .	175
14.3.6 Code définitif de notre page index.html . . . . .	176
14.3.7 Démonstration !!! . . . . .	177
14.3.8 Conclusion . . . . .	182
14.4 Encore d'autres frameworks MVC (javascript) . . . . .	183
14.5 Conclusion . . . . .	183
<b>15 Backbone et Typescript</b>	<b>184</b>
<b>16 Ressources Backbone</b>	<b>184</b>

# 1 Préambule & Remerciements

J'ai eu la prétention d'écrire un livre, et sur Backbone en plus ! En fait il n'existe peu de littérature française spécialisée sur des frameworks, qui plus est des framework javascript, alors que nos amis anglo-saxons écrivent sur Dart, Coffeescript, Backbone, etc. ... Au départ ce "bouquin" est un projet un peu fou, puisque j'ai même contacté les éditions Eyrolles (quand je vous disais que j'étais prétentieux ;). Et ils ont été d'accord ! Alors vous vous demandez pourquoi, finalement je publie ça de manière open source ?

Eh bien, écrire un livre est un travail de longue haleine, qui doit se faire dans la durée. D'un autre côté, les technologies, tout particulièrement ce qui gravite autour de javascript, progressent et changent à une allure vertigineuse. Mon constat est que, si je veux écrire tout ce que j'ai en tête, cela ne finira jamais, ou bien le contenu n'aura plus d'intérêt (obsolète) et qu'il me semble plus approprié de livrer déjà ce que j'ai "gratté" et de transformer ce livre en projet open-source.

Ainsi, ceux qui souhaite se mettre à Backbone, peuvent commencer dès maintenant (même si on ne m'a pas attendu, j'imagine qu'un peu de documentation en français devrait faire des heureux). Pour les autres s'ils souhaitent compléter, corriger, modifier, discuter, je me tiens à leur disposition. C'est pour cela que j'ai publié le contenu sur Github = ainsi vous avez l'opportunité de faire des pull-requests (proposer des modifications) sur le contenu, ou créer des issues pour donner votre avis.

Je vous attends, j'espère que cela vous sera utile. Je m'adresse à tous les publics (les plus forts n'apprendront rien, mais peuvent contribuer). Ceux qui connaissent déjà Backbone peuvent directement passer au chapitre 04.

Je tiens à remercier très fortement et tout particulièrement, pour leur écoute, leurs conseils et leur relecture :

- Muriel SHANSEIFAN (Éditions Eyrolles - Responsable éditoriales du secteur informatique)
- Laurène GIBAUD (Éditions Eyrolles - Secteur Informatique)

Remerciements aussi pour :

- [ehsavoie](#) : 1ère pull request ;)
- [lodennan](#)
- [ebonnissent](#) : format epub
- [loicdescotte](#) : relecture

## 1.1 Avertissement

Cet ouvrage est destiné à être purement éducatif. donc dès fois le code n'est pas fait dans les "règles de l'art", mais plutôt avec une "vision" pédagogique. Désolé donc pour les puristes, mais à priori vous n'êtes pas la cible ;). Cependant, je serais ravi de pouvoir inclure vos remarques et bonnes pratiques sous forme de notes dans chacun des chapitres. Donc à vos pull-requests !

## 2 Présentation de Backbone & rappels MVC

*Sommaire*

- *A quoi sert Backbone.js ?*
- *Qu'est-ce qu'une « Webapp » ?*
- *Petit rappel à propos de MVC*

*Où nous allons voir pourquoi Backbone existe et quels sont les grands principes qu'il met en œuvre.*

Ce chapitre est très court, il présente les origines de Backbone.js, le pourquoi de son utilisation, et enfin un rappel sur le patron de conception Modèle-Vue-Contrôleur, essentiel pour la bonne compréhension du framework.

### 2.1 Backbone ? Webapps ? MVC ?

Backbone est un framework javascript dédié à la création de **Webapps** en mode “**Single Page Application**”. Il implémente le pattern MVC (L'acronyme signifie : Model View Controller / Modèle Vue Contrôleur) mais, et c'est là qu'est la nouveauté, côté client, plus précisément au sein de votre navigateur. Il reproduit les mécanismes des frameworks MVC côté serveur tels Ruby on Rails, CakePHP, Play!>Framework, ASP.Net MVC (avec Razor), ... Backbone a été écrit par Jeremy Ashkenas (le papa de Coffeescript et de Underscore) à l'origine pour ses propres besoins lors du développement du site DocumentCloud (<http://www.documentcloud.org/home>). Son idée était de créer un framework qui permette de structurer ses développements en s'appuyant justement sur MVC. Mais avant toute chose, faisons quelques petits rappels (ou découvertes ?).

#### 2.1.1 Qu'est-ce qu'une “Webapp” ?

Une “Webapp” n'a pas la même vocation qu'un site web même si les technologies mises en œuvre sont les mêmes. Elle a une réelle valeur applicative (gestion de catalogue produits, utilitaires, clients mails, agenda, etc. ...) contrairement au site web qui le plus souvent est un medium de communication (journaux, blogs, etc. ...). Assez rapidement, les technologies web ont été détournées pour tenter de remplacer les applications de gestion “client-riche” classique. Imaginez, le rêve des DSI : plus de déploiement, tout se passe dans le navigateur. Cependant, c'était sans compter les utilisateurs. Je me souviens avoir vu une gestion de catalogue il y a une dizaine d'années, en ASP 3 ou à chaque création d'article, il fallait attendre que toute la liste des articles se recharge. Ce qui avant en mode texte (sous dos) prenait 1 minute, venait de prendre 3 à 5 minutes dans la vue : 3 à 5 fois plus de temps ! Bravo la productivité ! Ensuite les technologies se sont cherchées longtemps pour tenter de remédier à ce problème : apparition des applets java, des ActiveX et là on commençait à retomber dans les travers de déploiements compliqués. Puis il y eu Flash, Flex et Silverlight, pas mal du tout il faut l'avouer, mais parallèlement le moteur javascript avait évolué, les navigateurs aussi et avec l'avènement du triptyque HTML5 – CSS3 – Javascript, un nouveau concept est apparu : la “Single Page Application” (probablement boosté par les mobiles et les tablettes ... et Steve Jobs refusant que Flash/Air/Flex & Java s'installent sur l'iPhone & l'iPad). Mais qu'est donc une “Single Page Application” ? Il existe moult définitions, je vais donc vous donner la mienne, ensuite ce sera à vous de vous construire votre vision de “l'application web monopage”.

Une “Single Page Application” est une application web qui embarque tous les éléments nécessaires à son fonctionnement dans une seule page HTML. Les scripts javascript liés seront chargés en même temps. Ensuite l’application web chargera les ressources nécessaires (généralement des données, des images ...) à la demande en utilisant Ajax évitant ainsi tout rechargement de page et procurant une expérience utilisateur proche de celle que nous connaissons en mode “client-serveur”, voire meilleure dans certains cas. Ces webapps nouvelle génération peuvent aussi fonctionner offline en profitant des possibilités des derniers navigateurs (localStorage).

**Remarque :** ce qui est amusant, c'est que dès 1999 ou 2000, Microsoft avait déjà introduit cette possibilité avec Internet Explorer 4 qui intégrait une applet Java (si si !) qui permettait de faire du Remote Protocol Call d'une page html vers le serveur sans recharger la page et en s'abonnant en javascript à l'évènement de retour (à vérifier, c'est loin, tout ça). Mais ce fut éclipsé par l'apparente simplicité de mise en œuvre des ActiveX (Flash était alors utilisé principalement pour de l'animation, mais permettait aussi ce genre d'artifice).

Tout ça c'est bien beau, mais vous savez comme moi qu'un code HTML+JS (+CSS) peut rapidement devenir un plat de spaghetti impossible à maintenir, pour les autres mais pour vous aussi (retournez dans votre code 6 mois plus tard ;)). Il faut donc s'astreindre à des règles et s'équiper des bons outils afin de se faciliter la tâche, ne pas avoir à réinventer la poudre à chaque fois et pouvoir coder des applications robustes facilement modifiables (faciles à corriger, faciles à faire évoluer). Et si en plus vous pouvez vous faire plaisir ...

C'est de ce constat qu'est parti Jeremy Ashkenas, et c'est en mettant en pratique les préceptes depuis longtemps éprouvés de MVC qu'il a conçu Backbone, pour répondre à une problématique existante, ce qui le rend d'autant plus légitime. Rafraîchissons donc un peu notre mémoire à propos de MVC.

**Remarque :** pour les lecteurs qui ne connaîtraient pas ce concept, ne référez pas le livre tout de suite, vous verrez avec les exemples pratiques que le concept est simple et facilement assimilable. Donc si les quelques paragraphes théoriques qui suivent vous semblent obscurs, je vous promets que dans quelques chapitres vous aurez tout compris.

### 2.1.2 Petit rappel : MVC ?

MVC un pattern (modèle) de programmation utilisé pour développer des applications de manière structurée et organisée. Le pattern MVC, comme tous les patterns, cadre votre façon de développer. Son objectif particulier est de séparer les responsabilités de vos “bouts” de codes en les regroupant selon 3 rôles (responsabilités), donc le modèle, la vue et le contrôleur. Détaillons-les :

- **la vue** : c'est l'IHM (Interface Homme-Machine), ce qui va apparaître à l'écran de l'utilisateur. Elle va recevoir des infos du contrôleur (“affiche moi ça !”), elle va envoyer des infos au contrôleur (“on m'a cliquée dessus, il me faut la liste des clients”)
- **le contrôleur** : c'est lui donc qui reçoit des infos de la vue, qui va aller récupérer des données métiers chez le modèle (“j'ai besoin pour la vue de la liste des clients”), et va les renvoyer à la vue et éventuellement appliquer des traitements à ces données avant de les renvoyer.
- **le modèle** : c'est votre objet client, fournisseur, utilisateur, ... avec toute la mécanique qui sert à les sauvegarder, les retrouver, modifier, supprimer ...

**AVERTISSEMENT 1 :** *C'est la lisibilité qui importe. Il peut y avoir des interprétations différentes du modèle MVC quant aux responsabilités de ses composants. Par exemple, est-ce le modèle qui se sauvegarde lui-même ou est-ce un contrôleur qui se chargera de la persistance ? Peu importe (ce sont des querelles de chapelle), gardons juste à l'esprit qu'il y a trois grands modes de classement de nos objets et que l'important c'est d'avoir un code propre, structuré et maintenable (Dans 6 mois, vous devez être capables de relire votre code).*

**AVERTISSEMENT 2 :** *La difficulté n'est plus de mise. A l'attention des réfugiés de STRUTS. Mon "1er contact" avec MVC a été avec STRUTS. J'ai trouvé l'expérience peu concluante (expérience développeur désastreuse) et je suis retourné faire de l'ASP.Net "à la souris" (c'était avant 2005). Si certains d'entre vous se sont éloignés de la technique et ont des velléités de s'y remettre mais sont effrayés par MVC, je les rassure tout de suite, les développeurs nous ont enfin "concoctés" des frameworks simples et faciles à mettre en œuvre tels :*

- *ASP.Net MVC avec le moteur de template Razor*
- *Play!>Framework*
- *Express.js (avec nodeJS)*
- *Et beaucoup d'autres*

*Backbone.js respecte ce principe. Donc n'ayez pas peur, cela va être facile ;)*

L'interprétation de MVC par Backbone est un peu particulière et les développeurs Java, .Net, PHP, Ruby, Python etc. pourraient être surpris. Mais, passons donc à quelques explications.

## 2.2 Backbone & MVC

Backbone "embarque" plusieurs composants qui vont nous permettre d'organiser notre webapp selon les "préceptes" MVC et simplifier les communications avec le serveur (avec le code applicatif côté serveur).

Voyons l'interprétation que fait Backbone du modèle MVC :

- Le composant **Modèle** (selon Backbone : `Backbone.Model`) représente les données qui vont interagir avec votre code "backend" (côté serveur) via des requêtes Ajax. Ce sont eux qui auront la responsabilité de la logique métier et de la validations des données.
- Le composant **Vue** (selon Backbone : `Backbone.View`) n'est pas complètement une vue au sens où on l'entend habituellement (couche présentation). Dans le cas qui nous intéresse, la "vraie" vue est un fragment de code "natif" HTML dans la page web qui s'affiche dans le navigateur (il y a donc plusieurs vues dans une même page). Et `Backbone.View` est en fait un **Contrôleur de vues** (1) (vous verrez, ce sera plus facile à apprécier en le codant) qui va ordonner les événements et interactions au sein de la page web.

(1) : *c'est une interprétation très personnelle, c'est discutable, je reste à votre disposition*

Pour résumer, avec un parallèle avec du MVC dit "classique", nous avons :

- Modèle : `Backbone.Model`
- Vue : le code HTML
- Contrôleur (de vues) : `Backbone.View`

**Mais ce n'est pas fini !** Backbone apporte 3 composants supplémentaires :

- Le composant **Routeur** : `Backbone.Router`, qui écoute/surveille les changements d'URL (dans la barre d'url du navigateur, lors d'un clic sur un lien, ...) et qui fait le lien avec les `Backbone.Model(s)` et les `Backbone.View(s)`.
- Le composant **Collection** : `Backbone.Collection`, des collections de modèles avec des méthodes pour “travailler” avec ceux-ci (`each`, `filter`, `map`, ...)
- et enfin le petit dernier, mais non des moindres, Le composant de **Synchronisation** `Backbone.sync`, que l'on pourrait comparer à une couche middleware, qui va permettre à nos modèles de communiquer avec le serveur. C'est `Backbone.sync` qui va faire les requêtes Ajax au serveur et “remonter” les résultats aux modèles.

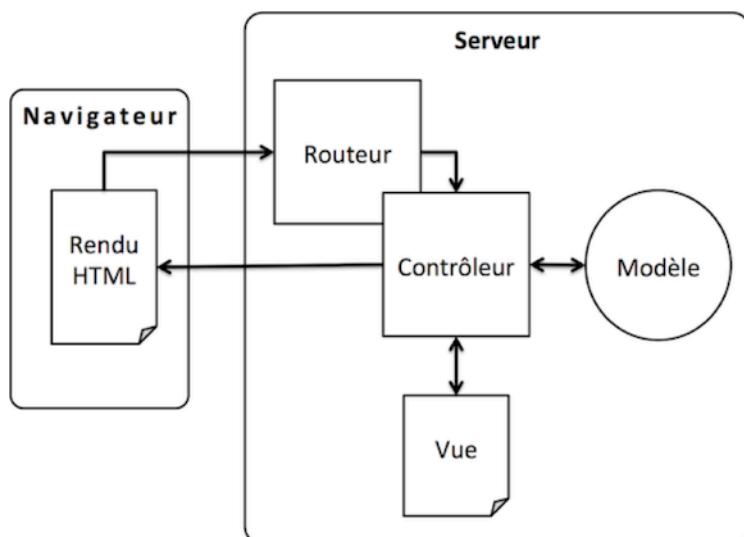


Figure 1-1. MVC Vision “Back-End”

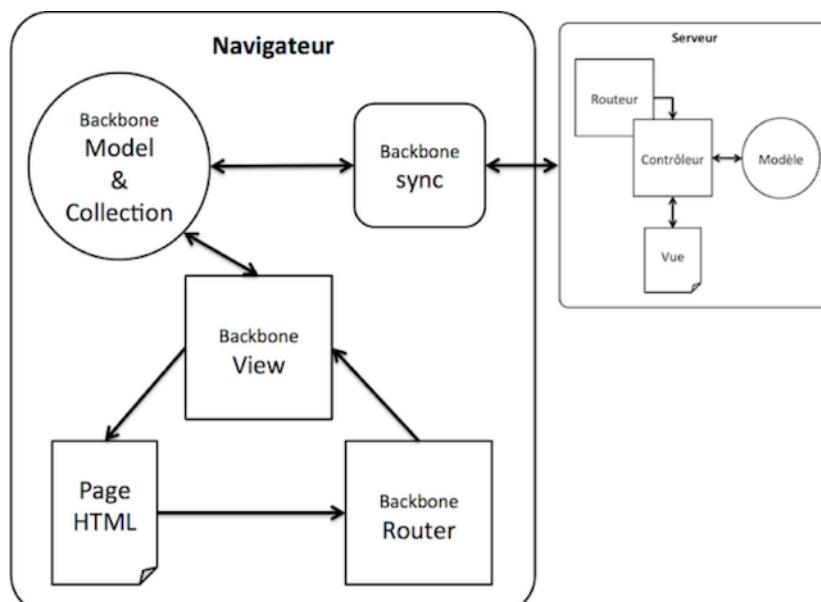


Figure 1-2. MVC Vision “Front-End”

## 2.3 Pourquoi j'ai choisi Backbone ?

//TODO:

Tout ceci vous paraît bien théorique ? Alors passons tout de suite à la pratique.

# 3 Tout de suite “les mains dans le cambouis”

*Sommaire*

- *Les prérequis & IDE pour faire fonctionner Backbone*
- *jQuery en 15 minutes*
- *Underscore.js en 10 minutes*

*Où nous allons lister les éléments nécessaires pour installer Backbone et commencer à développer avec.*

Le plus frustrant lorsque l'on débute la lecture d'un ouvrage informatique dans l'optique de s'auto former c'est que l'on est obligé de lire de nombreux chapitres avant de pouvoir commencer à s'y mettre. Je vais donc tenter de vous faire faire un 1er tour de Backbone.js en 20 minutes pour que vous en saisissez rapidement la “substantifique moelle”. Mais avant d'utiliser Backbone, quelques prérequis sont nécessaires. □

## 3.1 Prérequis : les dépendances de Backbone

Backbone a besoin au minimum de deux autres frameworks javascript pour fonctionner :

- **Underscore.js** par le créateur de Backbone. Underscore est un ensemble d'outils qui permettent d'étendre javascript et qui vont vous faciliter la vie dans la gestion des Collections, Arrays, Objects, ... mais aussi vous permettre de faire du templating (nous verrons ce que c'est plus loin). Le gros avantage d'Underscore; c'est qu'il fonctionne quel que soit votre navigateur (comme Backbone). Underscore est une dépendance de Backbone, il est donc indispensable.
- **jQuery**, qui est un framework dédié à la manipulation des éléments de votre page HTML (on parlera du DOM, Document Object Model) mais aussi aux appels de type Ajax (nécessaire pour “discuter” avec le serveur). On peut dire que jQuery est un DSL (Domain Specific Language) pour le DOM. jQuery n'est pas indispensable pour faire fonctionner Backbone, mais il va grandement nous faciliter la vie dans la création de nos Webapps et va nous garantir le fonctionnement de notre code quel que soit le navigateur.

Nous verrons dans quelques chapitres qu'il est tout à fait possible de “marier” d'autres frameworks javascript à Backbone pour :

- faire du templating (certains peuvent trouver la fonctionnalité de template d'Underscore limitée)
- gérer la persistance locale (localStorage du navigateur)

- ...

**Remarque :** *Il est possible d'utiliser Zepto.js à la place de jQuery, Zepto fonctionne à l'identique de jQuery mais il est dédié principalement aux navigateurs mobiles et est beaucoup plus léger que jQuery (avantageux sur un mobile), cependant vous n'avez plus la garantie que votre code fonctionne dans d'autres navigateurs (Zepto “marchera” très bien sous Chrome, Safari et Firefox).*

## 3.2 Outils de développement

### 3.2.1 IDE (Editeur)

Pour coder choisissez l'éditeur de code avec lequel vous vous sentez le plus à l'aise. Ils ont tous leurs spécificités, ils sont gratuits, open-source ou payants. Certains “puristes” utilisent même Vim ou Emacs. Je vous en livre ici quelques-uns que j'ai trouvé agréables à utiliser si vous n'avez pas déjà fait votre choix :

- Mon préféré mais payant : WebStorm de chez JetBrains, il possède des fonctionnalités de refactoring très utiles (existe sous Windows, Linux et OSX)
- Dans le même esprit et gratuit : Netbeans, il propose un éditeur HTML/Javascript très pertinent quant à la qualité de votre code (existe sous Windows, Linux et OSX)
- Textmate (payant) un éditeur de texte avec colorisation syntaxique, un classique sous OSX
- SublimeText (payant) un peu l'équivalent de Textmate mais toutes plateformes
- Un bon compromis est KomodoEdit dans sa version communauté (donc non payant) et qui lui aussi fonctionne sur toutes les plateformes.
- Aptana fourni aussi un bon IDE dédié Javascript sur une base Eclipse, mais je trouve qu'il propose finalement trop de fonctionnalités (comme Eclipse), et personnellement je m'y perds.

Vous voyez, il y en a pour tous les goûts. En ce qui me concerne j'utilise essentiellement WebStorm ou SublimeText.

### 3.2.2 Navigateur

Le navigateur le plus agréable, selon moi, à utiliser pour faire du développement Web est certainement Chrome (C'est un avis très personnel, donc amis utilisateurs de Firefox ne m'en veuillez pas). En effet Chrome propose une console d'administration particulièrement puissante. C'est ce que je vais utiliser, rien ne vous empêche d'utiliser votre navigateur préféré. Par contre, que cela ne vous dispense pas d'aller tester régulièrement votre code sous d'autres navigateurs.

## 3.3 Initialisation de notre projet de travail

Maintenant que nous sommes “outillés” (un éditeur de code et un navigateur) nous allons pouvoir initialiser notre environnement de développement.

### 3.3.1 Installation

- Créer un répertoire de travail `backbone001`
- Créer ensuite un sous-répertoire `libs` avec un sous-répertoire `vendors`

Nous copierons les frameworks javascript dans `vendors`.

- Téléchargez **Backbone** : <http://documentcloud.github.com/backbone/>
- Téléchargez **Underscore** : <http://documentcloud.github.com/underscore/>

**CONSEIL** : Utilisez les version non minifiées des fichiers. Il est toujours intéressant de pouvoir lire le code source des frameworks lorsqu'ils sont bien documentés, ce qui est le cas de Backbone et Underscore, n'hésitez pas à aller mettre le nez dedans, c'est instructif et ces 2 frameworks sont très lisibles, même pour des débutants.

- Téléchargez **jQuery** : <http://jquery.com/>

Nous allons aussi récupérer le framework css **TwitterBootstrap** qui nous permettra de faire de “jolies” pages sans effort. Ce n'est pas du tout obligatoire, mais c'est toujours plus satisfaisant d'avoir une belle page d'exemple. : <http://twitter.github.com/bootstrap/>. Téléchargez `bootstrap.zip`, “dé-zippez” le fichier et copiez le répertoire `bootstrap` dans votre répertoire `vendors`.

### 3.3.2 Préparons notre page HTML

A la racine de votre répertoire de travail, créez une page `index.html` avec le code suivant :

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <title>Backbone</title>

  <!-- == Styles Twitter Bootstrap -->
  <link href="libs/vendors/bootstrap/css/bootstrap.css" rel="stylesheet">
  <link href="libs/vendors/bootstrap/css/bootstrap-responsive.css" rel="stylesheet">
</head>

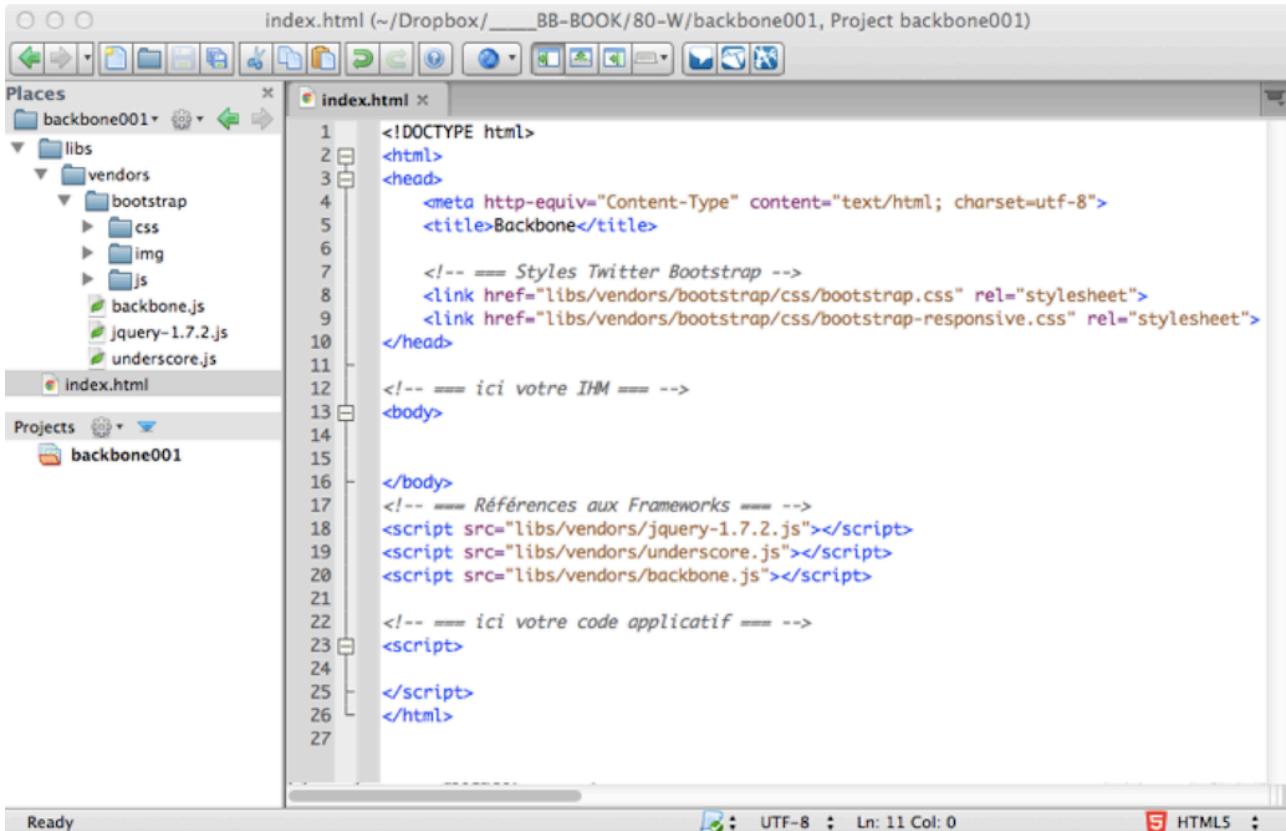
<!-- == ici votre IHM == -->
<body>

</body>
<!-- == Références aux Frameworks == -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>
```

```
<!-- == ici votre code applicatif == -->
<script>

</script>
</html>
```

A ce niveau, vous devriez avoir un squelette de projet fonctionnel avec l’arborescence suivante :



Les deux paragraphes qui suivent ne sont que pour ceux d’entre vous qui ne connaissent ni **jQuery** ni **Underscore**. Ces paragraphes n’ont pas la prétention de vous apprendre ces outils, mais vous donneront les bases nécessaires pour vous en servir, pour comprendre leur utilité et pour vous donner envie d’aller plus loin. Les autres (ceux qui connaissent déjà), passez directement au § “**1er contact ... avec Backbone**”.

### 3.4 Jouons avec jQuery

JQuery est un framework javascript initialement crée par John Resig qui vous permet de prendre le contrôle de votre page HTML. Voyons tout de suite comment nous en servir. Dans notre toute nouvelle page `index.html`, préparons un peu notre bac à sable et saisissons le code suivant :

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Backbone</title>
```

```

<!-- === Styles Twitter Bootstrap -->
<link href="libs/vendors/bootstrap/css/bootstrap.css" rel="stylesheet">

<!-- à insérer entre les 2 <link> ==-->
<style>
    body {
        padding-top: 60px;
        /* 60px pour mettre un peu d'espace entre la barre de titre et le contenu */
        padding-bottom: 40px;
    }
</style>

<link href="libs/vendors/bootstrap/css/bootstrap-responsive.css" rel="stylesheet">

</head>

<!-- ici votre IHM ==-->
<body>
    <!--
        les classes CSS "navbar navbar-fixed-top", "navbar-inner", "container",
        "brand", "hero-unit"
        viennent de la feuille de style "twitter bootstrap "
    -->
    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand">Mon Blog</a>
            </div>
        </div>
    </div>

    <div class="container">

        <div class="hero-unit">
            <h1>Backbone rocks !!!</h1>
            <p>
                "Ma vie mon oeuvre"
            </p>
        </div>

        <div id="articles_box">

            <h1 id="current_articles_title">les articles du blogs</h1>

            <ul id="current_articles_list">
                <li>Backbone et les modèles</li>
                <li>Backbone et les vues</li>
                <li>Backbone : mais y a-t-il vraiment un contrôleur dans l'avion ?</li>
            </ul>
        </div>
    </div>
</body>

```

```
<h1 id="next_articles_title">les articles à venir</h1>

<ul id="next_articles_list">
    <li>Backbone et le localStorage</li>
    <li>Backbone.sync : comment ça marche</li>
</ul>

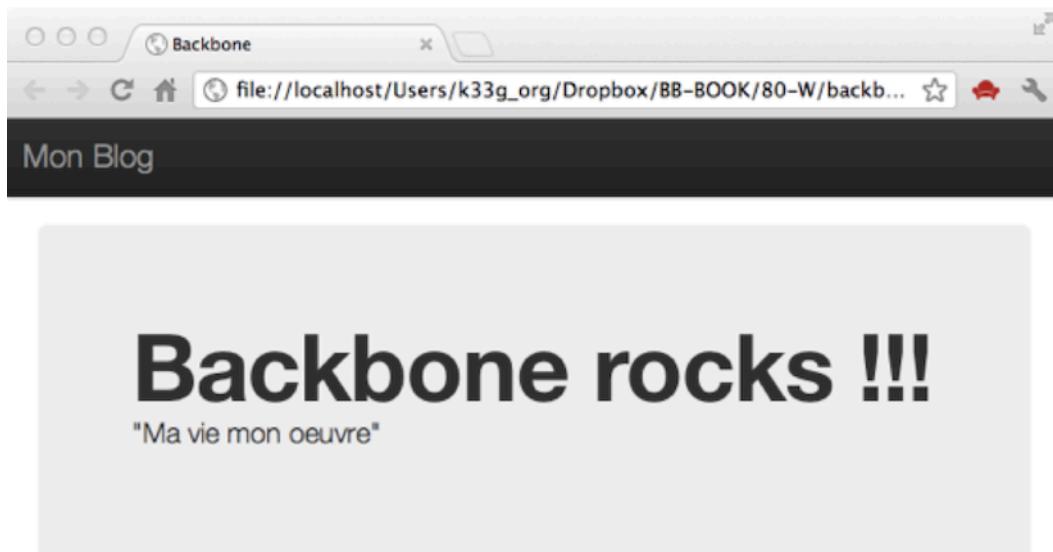
</div>
</div>

</body>
<!-- === Références aux Frameworks === -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>

<!-- === ici votre code applicatif === -->
<script>

</script>
</html>
```

Une fois votre page terminée, sauvegardez là et ouvrez là dans votre navigateur préféré (qui je le rappelle, pour des raisons purement pédagogique est Chrome) :



## les articles du blogs

- Backbone et les modèles
- Backbone et les vues
- Backbone : mais y a-t-il vraiment un contrôleur dans l'avion ?

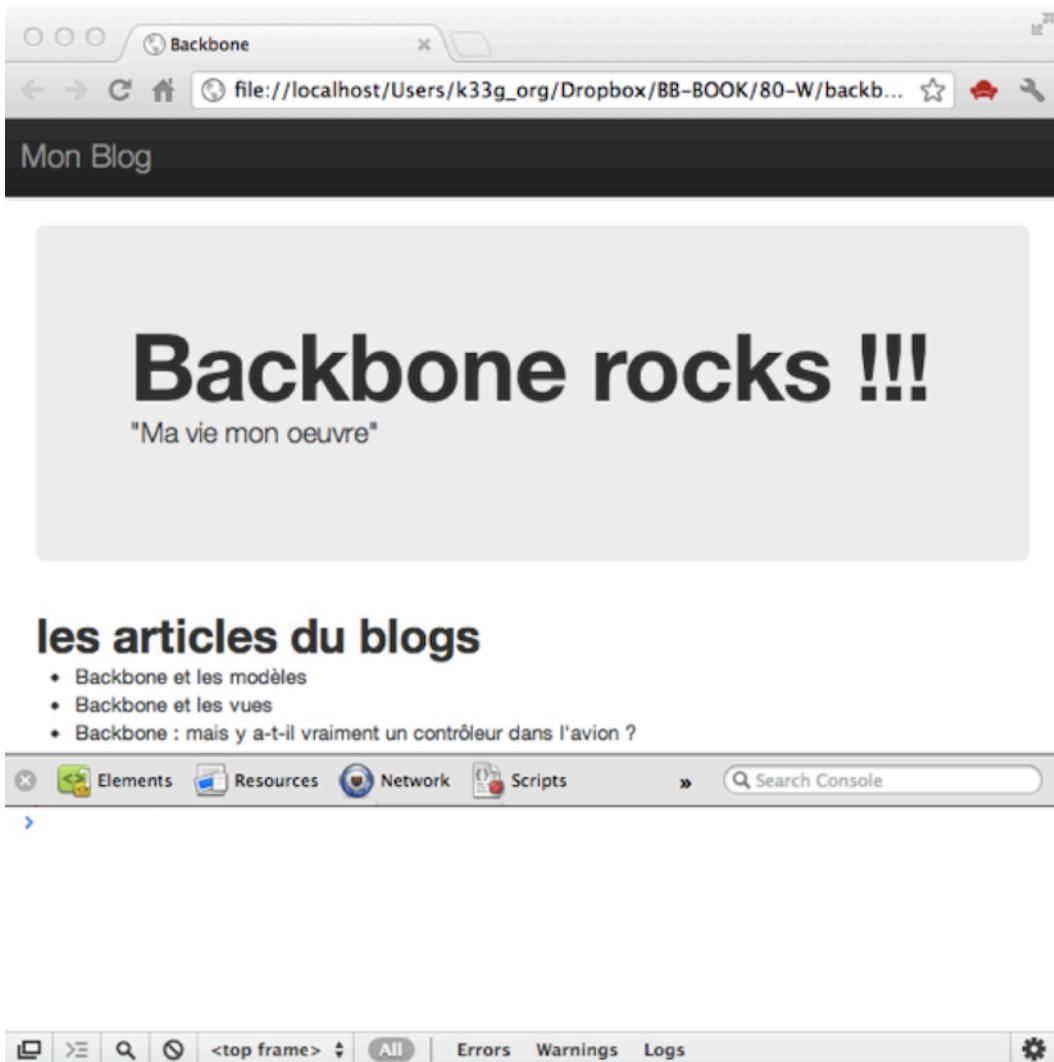
## les articles à venir

- Backbone et le localstorage
- Backbone.sync : comment ça marche

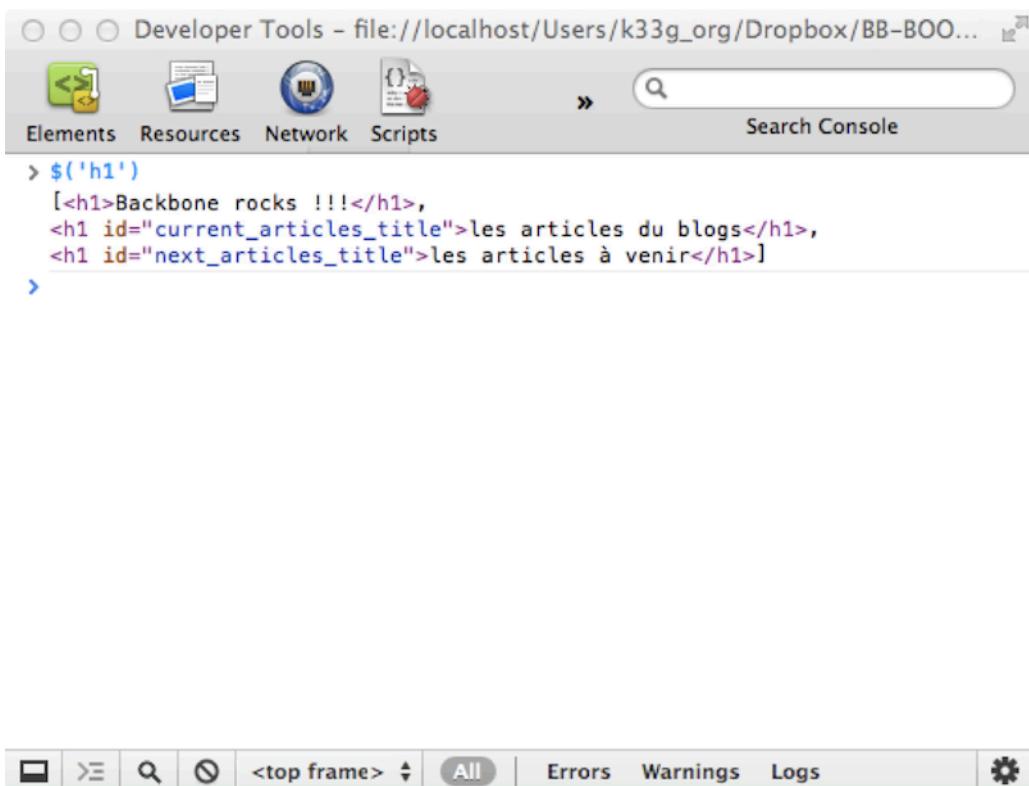
Notez au passage la qualité graphique de votre page ;), tout ça sans trop d'efforts, grâce à TwitterBootstrap.

### 3.4.1 “Jouons” avec notre page en mode commande

Dans un premier temps, ouvrez la console de Chrome (ou Safari) : faites un clic droit sur la page et sélectionnez “Inspect Element” (ou “Inspecter l’élément”). Pour les aficionados de Firefox : utilisez les menus : Tools/Web Developer/Web Console. Vous devriez obtenir ceci (cliquez sur le bouton “Console” si nécessaire) :



**Saisissons nos 1ères commandes :** Je voudrais la liste de mes titres <H1> : dans la console, saisir : \$('h1'), validez, et vous obtenez un tableau (Array au sens javascript) des nodes html de type <H1> présentes dans votre page html :

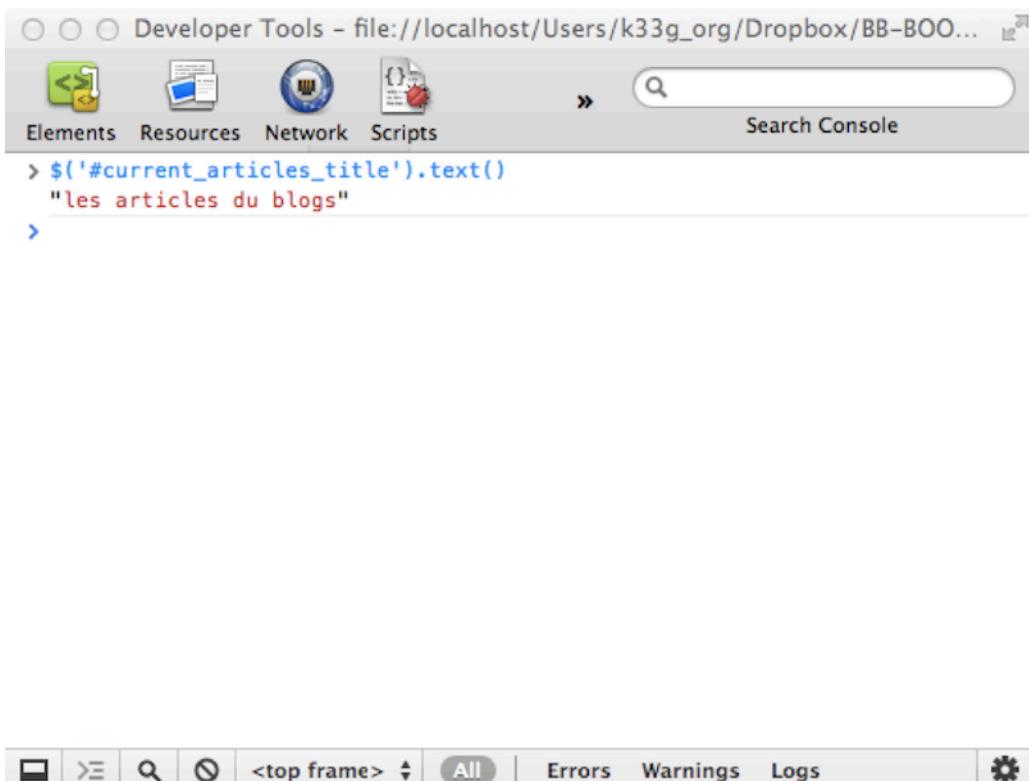


The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The title bar indicates 'Developer Tools - file:///localhost/Users/k33g\_org/Dropbox/BB-BOO...'. Below the title bar are four tabs: 'Elements' (highlighted), 'Resources', 'Network', and 'Scripts'. A search bar labeled 'Search Console' is positioned to the right of the tabs. The main area contains the following code:

```
> $('h1')
[<h1>Backbone rocks !!!</h1>,
<h1 id="current_articles_title">les articles du blogs</h1>,
<h1 id="next_articles_title">les articles à venir</h1>]
```

Below the code, there is a toolbar with icons for copy, paste, clear, and search, followed by '<top frame>' and a dropdown menu set to 'All'. To the right of the dropdown are buttons for 'Errors', 'Warnings', and 'Logs', and a gear icon for settings.

Je voudrais le texte du titre <H1> dont l'id est "current\_articles\_title" : dans la console, saisir :  
`$('#current_articles_title').text()`. L'identifiant étant unique, en fait le type de la node est peu important :

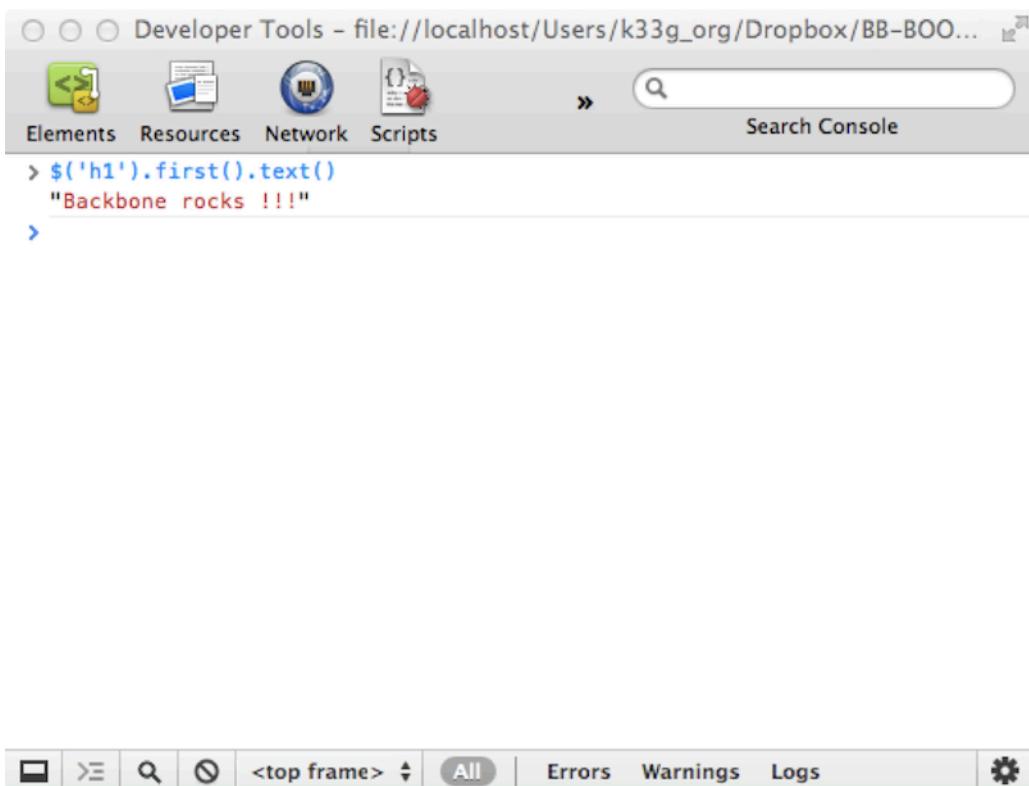


The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The title bar indicates 'Developer Tools - file:///localhost/Users/k33g\_org/Dropbox/BB-BOO...'. Below the title bar are four tabs: 'Elements' (highlighted), 'Resources', 'Network', and 'Scripts'. A search bar labeled 'Search Console' is positioned to the right of the tabs. The main area contains the following code:

```
> $('#current_articles_title').text()
"les articles du blogs"
```

Below the code, there is a toolbar with icons for copy, paste, clear, and search, followed by '<top frame>' and a dropdown menu set to 'All'. To the right of the dropdown are buttons for 'Errors', 'Warnings', and 'Logs', and a gear icon for settings.

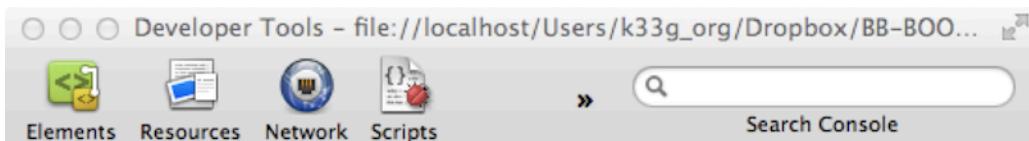
Mais comment dois-je faire pour avoir le texte du premier <H1> de ma page, il n'a pas d'id ?!?. Tout simplement, en utilisant la commande suivante : `$('.h1').first().text()` :



**Modifions l'aspect de notre page dynamiquement :** Les commandes sont toujours à saisir dans la console du navigateur. Je voudrais :

- changer le titre de mon blog : `$('h1').first().text("Backbone c'est top !")`, attention pensez bien au `first()` sinon vous allez changer tous les textes de tous les H1 de la page.
- récupérer le code HTML de la “boîte de titre” (le div avec la classe css : `class="hero-unit"`) : `$('[class="hero-unit"]').html()`, notez bien que `$('[class="hero-unit"]')`.`text()` ne retourne pas le même résultat. On peut aussi écrire ceci plus simplement : `$('.hero-unit').html()` : le `".."` correspond à une classe css, comme le `#` permet de rechercher un élément par son id.
- changer les couleurs de police et de fond de tous les tags H1 :  
`$('h1').css("color","white").css("background-color","black")`, vous voyez que vous pouvez faire des appels chaînés, mais une autre possibilité serait la suivante :

```
$('h1').css({color:"yellow", backgroundColor:"green"})
```



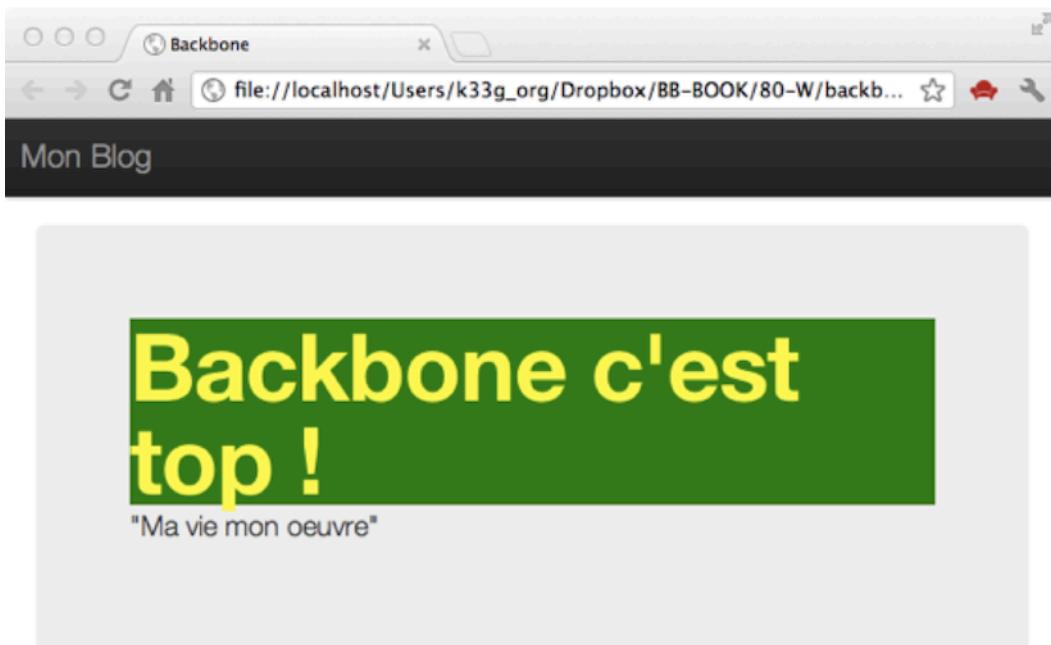
```

> $('h1').first().text("Backbone c'est top !")
[<h1>Backbone c'est top !</h1>]
> $('[class="hero-unit"]').html()
"
<h1>Backbone c'est top !</h1>
<p>
    "Ma vie mon oeuvre"
</p>
"
> $('[class="hero-unit"]').text()
"
    Backbone c'est top !
    "Ma vie mon oeuvre"
"

> $('h1').css("color","white").css("background-color","black")
[
<h1 style="color: white; background-color: black; ">Backbone c'est top
!</h1>
,
<h1 id="current_articles_title" style="color: white; background-color:
black; ">les articles du blogs</h1>
,
<h1 id="next_articles_title" style="color: white; background-color:
black; ">les articles à venir</h1>
]
> $('h1').css({color:"yellow", backgroundColor:"green"})
[
<h1 style="color: yellow; background-color: green; ">Backbone c'est top
!</h1>
,
<h1 id="current_articles_title" style="color: yellow; background-color:
green; ">les articles du blogs</h1>
,
<h1 id="next_articles_title" style="color: yellow; background-color:
green; ">les articles à venir</h1>
]
>

```

Toolbar icons: Stop, Refresh, Back, Forward, Stop, All, Errors, Warnings, Logs, Settings.



## les articles du blogs

- Backbone et les modèles
- Backbone et les vues
- Backbone : mais y a-t-il vraiment un contrôleur dans l'avion ?

## les articles à venir

- Backbone et le localstorage
- Backbone.sync : comment ça marche

Allons plus loin ... Je voudrais :

- la valeur de l'id de la deuxième liste (UL) : `$('#ul').eq(1).attr("id")`, je cherche la liste d'index 1 (le 1er élément possède l'index 0).
- parcourir les lignes (LI) de la liste dont l'id est "next\_articles\_list" et obtenir leur texte : `$('#next_articles_list').find('li').each(function (index) { console.log($(this).text()); })`
- ajouter une nouvelle ligne à la 2ème liste :

```
$('#<li>Templating et Backbone</li>').appendTo('#next_articles_list')
```

- cacher la 1ère liste : `$('#current_articles_list').hide()`
- l'afficher à nouveau : `$('#current_articles_list').show()`
- la cacher à nouveau, mais “doucement” : `$('#current_articles_list').hide('slow')`
- l'afficher à nouveau, mais “rapidement” : `$('#current_articles_list').show('fast')`



The screenshot shows the Chrome Developer Tools with the 'Console' tab selected. The console window displays a series of jQuery commands being executed. The commands involve selecting an 'ul' element, finding 'li' elements within it, logging their text to the console, and then manipulating the visibility of 'current\_articles\_list' and 'next\_articles\_list' lists. The output shows the resulting HTML structure and CSS styles for these lists.

```

> $('ul').eq(1).attr("id")
"next_articles_list"
> $('#next_articles_list').find('li').each(function (index) { console.log(
$(this).text() ); })
Backbone et le localstorage
Backbone.sync : comment ça marche
< [ <li>Backbone et le localstorage</li>,
<li>Backbone.sync : comment ça marche</li>]
> $('- Templating et Backbone</li>').appendTo('#next_articles_list')
[<li>Templating et Backbone</li>]
> $('#current_articles_list').hide()
[▶<ul id="current_articles_list" style="display: none; ">...</ul>]
> $('#current_articles_list').show()
[▶<ul id="current_articles_list" style="display: block; ">...</ul>]
> $('#current_articles_list').hide('slow')
[
▶<ul id="current_articles_list" style="display: block; overflow-x: hidden; overflow-y: hidden; height: 53.64510434298379px; margin-top: 0px; margin-bottom: 8.940850723830632px; padding-top: 0px; padding-bottom: 0px; width: 572.214463251605px; margin-left: 24.835696455085092px; margin-right: 0px; padding-left: 0px; padding-right: 0px; opacity: 0.9934278582034036; ">...</ul>
]
> $('#current_articles_list').show('fast')
[
▶<ul id="current_articles_list" style="overflow-x: hidden; overflow-y: hidden; display: block; height: 10.289626719250135px; margin-top: 0px; margin-bottom: 1.577483782514174px; padding-top: 0px; padding-bottom: 0px; width: 101.78368610507223px; margin-left: 4.381899395872706px; margin-right: 0px; padding-left: 0px; padding-right: 0px; opacity: 0.17527597583490823; ">...</ul>
]

```

### 3.4.2 Les évènements

//À traiter ...

### 3.4.3 Quelques bonnes pratiques

**Pensez performances :** Si vous devez utiliser plusieurs fois le même élément de votre page : par exemple `$('#current_articles_list')`, sachez qu'à chaque fois jQuery “interroge” le DOM. Pour des raisons de performances, il est conseillé d'affecter le résultat de la sélection à une variable que vous réutiliserez ensuite. De cette manière, le DOM n'est interrogé qu'une seule fois. Vous pouvez tester ceci dans la console :

```

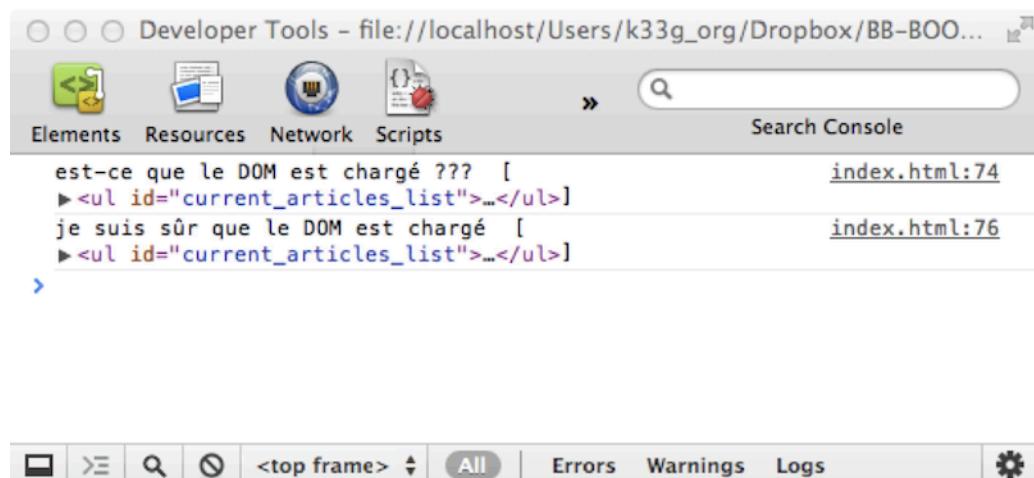
var currArtList = $('#current_articles_list');
currArtList.hide('slow');
currArtList.show('fast');

```

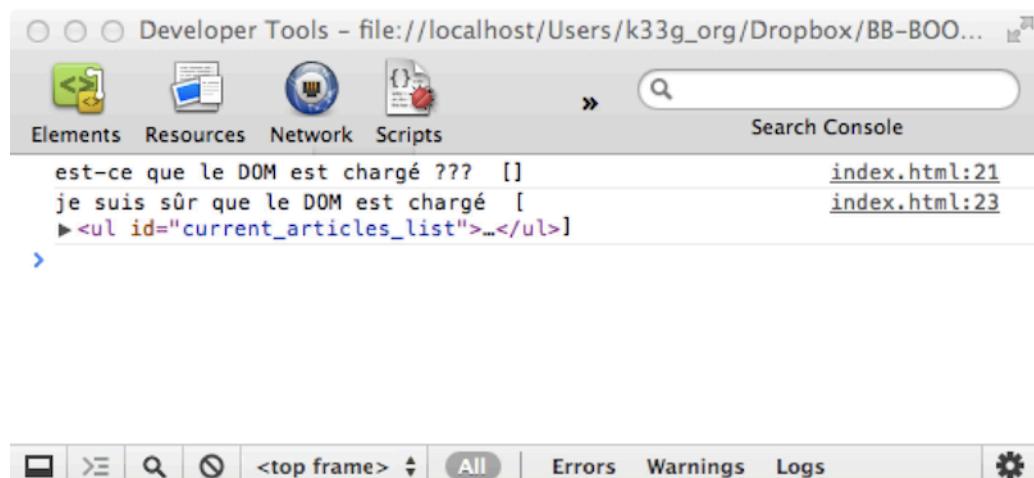
**Soyez sûr que les éléments de votre page sont tous chargés :** Il est intéressant (indispensable) d'avoir la garantie que son code javascript n'est exécuté qu'une seule fois la page HTML chargée dans son entièreté, surtout si ce code accède à des éléments du DOM. jQuery a une fonction pour ça : `$(document).ready(handler)` ou encore plus court : `$(handler)` où `handler` est une fonction. Mettez ce code dans la balise `<script>` de votre page `index.html` :

```
console.log("est-ce que le DOM est chargé ??? ", $('#current_articles_list'));
$(function () {
    console.log("je suis sûr que le DOM est chargé ", $('#current_articles_list'));
});
```

Puis ouvrez la page dans votre navigateur et activez la console :



Il semble que tous les éléments soient chargés correctement avec ou sans l'utilisation de la méthode `ready()` de jQuery. Vous avez du remarquer que j'avais déplacé mon code javascript et les références aux autres code javascript “en bas de ma page”. Maintenant, déplacez `<script src="libs/vendors/jquery-1.7.2.js"></script>` et le code source que nous avons écrit au niveau du header (balise `<head>`) de la page, ce qui est plus “classique” et rechargez la page :



Et là on voit bien qu'au 1er appel `$('#current_articles_list')` jQuery ne trouve rien, puis une fois le DOM chargé, jQuery trouve la liste. J'ai mis mes codes en bas de page, pour des raisons de performances et c'est pour ça que cela “semblait” fonctionner même à l'extérieur de `$(document).ready(handler)`, les éléments se chargeant plus rapidement, mais ça ne garantit rien, tout particulièrement lorsque

vos page n'est plus en local. Donc n'oubliez jamais d'exécuter votre code au bon moment grâce à `$(document).ready(handler)`, ... Et remettez quand même votre code en bas de page ;).

Vous venez de voir une infime partie des possibilités de jQuery, mais cela vous donne déjà un aperçu et vous permet de commencer à jouer avec et aller plus loin. jQuery permet aussi de faire des requêtes AJAX (<http://>) vers des serveurs web, mais nous verrons cela un peu plus tard.

```
//TODO: traiter la notion d'id versus la notion de name
```

### 3.5 Jouons avec Underscore

Underscore est un framework javascript (par le créateur de Backbone) qui apporte de nombreuses fonctionnalités pour faire des traitements sur des tableaux de valeurs (Array), des collections (tableaux d'objet). Certaines de ces fonctionnalités existent en javascript, mais uniquement dans sa dernière version, alors qu'avec Underscore vous aurez la garantie qu'elles s'exécutent sur tous les navigateurs. Mais Underscore, ce sont aussi des fonctionnalités autour des fonctions et des objets (là aussi, le framework vous procure les possibilités de la dernière version de javascript quel que soit votre navigateur ... ou presque, je n'ai pas testé sous IE6) et autres utilitaires, tels le templating. Je vous engage à aller sur le site, la documentation est particulièrement bien faite.

#### 3.5.1 Quelques exemples d'utilisations

Backbone utilise et encapsule de nombreuses fonctionnalités d'Underscore (Collection, modèle objet, ...) donc vous n'aurez pas forcément l'obligation d'utiliser Underscore directement. Je vous livre cependant quelques exemples, car cette puissante librairie peut vous aider sur d'autres projets pas forcément dédiés Backbone. Pour les tester, nous continuons avec la console de notre navigateur (toujours avec notre page index.html).

**Tableaux et Collections :** Commencez par saisir ceci :

```
var buddies = [],
    bob = { name : "Bob Morane", age : 32 },
    sam = { name : "Sammy", age : 43 },
    tom = { name : "Tommy", age : 25 };
buddies.push(bob, sam, tom);
```

Nous avons donc un tableau de 3 objets :

The screenshot shows the Chrome Developer Tools interface. At the top, there are four tabs: Elements, Resources, Network, and Scripts. The Scripts tab is active, displaying the following JavaScript code:

```

> var buddies = [],
    bob = { name : "Bob Morane", age : 32 },
    sam = { name : "Sammy", age : 43 },
    tom = { name : "Tommy", age : 25 };
buddies.push(bob, sam, tom);
3
> buddies
[▼ Object , ▼ Object , ▼ Object ]
  age: 32          age: 43          age: 25
  name: "Bob Morane"  name: "Sammy"  name: "Tommy"
  ► __proto__: Object  ► __proto__: Object  ► __proto__: Object
>

```

Below the code editor is a search bar labeled "Search Console". At the bottom of the developer tools window, there is a toolbar with icons for refresh, zoom, and search, followed by buttons for "All", "Errors", "Warnings", "Logs", and a gear icon.

Je souhaite maintenant parcourir le tableau d'objets et afficher les informations de chacun d'eux. Pour cela utilisez la commande `each()` de la manière suivante :

```

_.each(buddies, function (buddy) {
  console.log (buddy.name, buddy.age);
});

```

Et vous obtiendrez ceci :

The screenshot shows the Chrome Developer Tools interface with the DevTools.js tab selected. At the top, there are tabs for Elements, Resources, Network, and Scripts. A search bar labeled "Search Console" is present. Below the tabs, the code is displayed:

```

> var buddies = [],
    bob = { name : "Bob Morane", age : 32 },
    sam = { name : "Sammy", age : 43 },
    tom = { name : "Tommy", age : 25 };
buddies.push(bob, sam, tom);
3
> buddies
[▼ Object , ▼ Object , ▼ Object ]
  age: 32          age: 43          age: 25
  name: "Bob Morane"  name: "Sammy"  name: "Tommy"
  ► __proto__: Object  ► __proto__: Object  ► __proto__: Object
> _.each(buddies, function (buddy) {
  console.log (buddy.name, buddy.age);
});
Bob Morane 32
Sammy 43
Tommy 25
< undefined
>

```

At the bottom, there is a toolbar with icons for back, forward, search, and refresh, followed by buttons for <top frame>, All (which is selected), Errors, Warnings, Logs, and a gear icon.

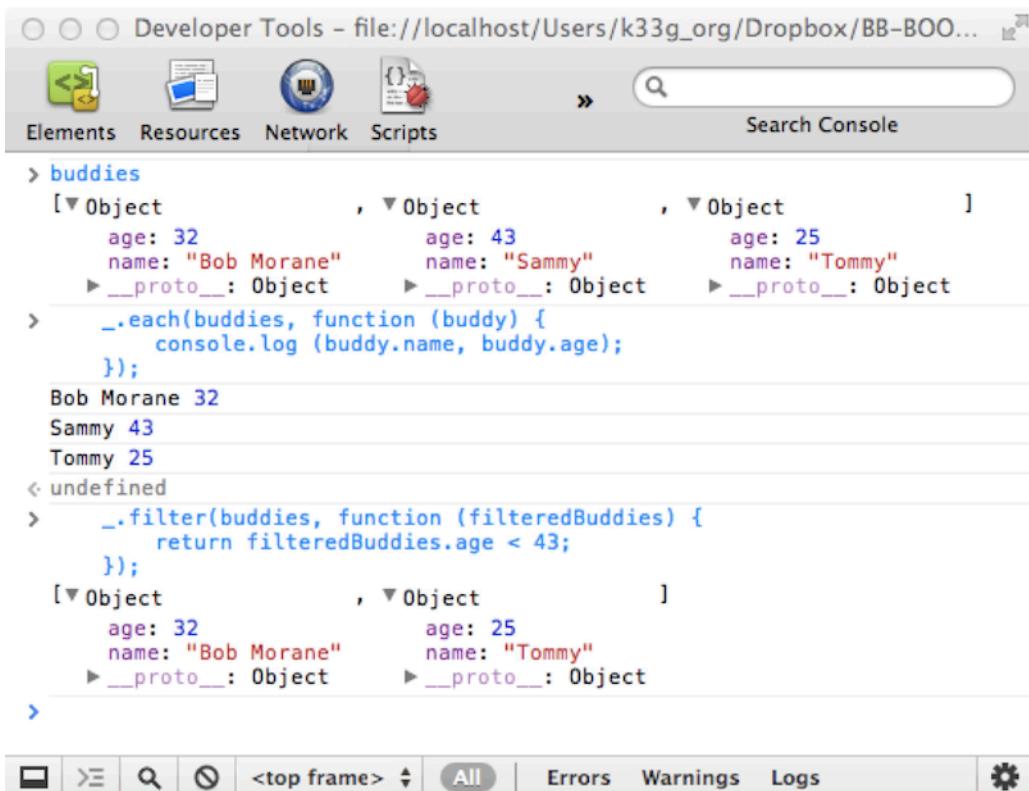
Je voudrais maintenant les “buddies” dont l’âge est inférieur à 43 ans. Nous allons utiliser la commande `filter()` :

```

_.filter(buddies, function (filteredBuddies) {
  return filteredBuddies.age < 43;
});

```

Et nous obtenons bien :



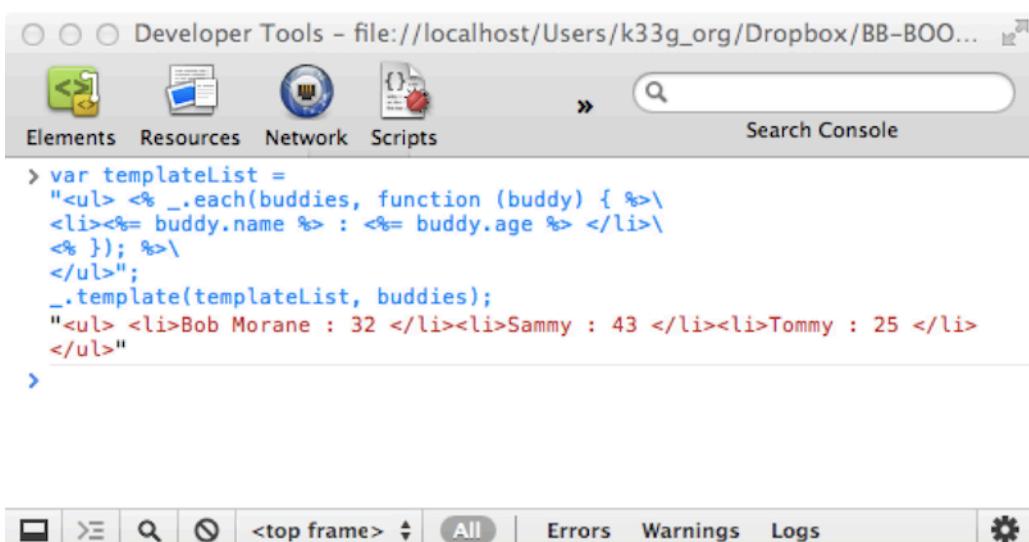
**Templating :** Je vous en parle maintenant, car ce “bijou” va nous servir très rapidement. Je voudrais générer une liste au sens HTML (`<ul><li></li></ul>`) à partir de mon tableau d’objets buddies. Nous allons donc créer une variable “template” (un peu comme une page JSP ou ASP) :

```
var templateList =
"<ul> <% _.each(buddies, function (buddy) { %>\n<li><%= buddy.name %> : <%= buddy.age %> </li>\n<% }); %>\n</ul>";
```

Que nous utiliserons de cette façon (nous passons à la méthode le template et les données):

```
_.template(templateList, buddies);
```

Pour le résultat suivant :



The screenshot shows the Chrome Developer Tools Network tab. At the top, there are four icons: Elements, Resources, Network, and Scripts. Below them is a search bar labeled "Search Console". The main area contains a block of JavaScript code:

```
> var templateList =
  "<ul> <% _.each(buddies, function (buddy) { %>
  <li><%= buddy.name %> : <%= buddy.age %> </li>
  <% }); %>
</ul>";
_.template(templateList, buddies);
"<ul> <li>Bob Morane : 32 </li><li>Sammy : 43 </li><li>Tommy : 25 </li>
</ul>"
```

Below the code, there is a toolbar with various icons and a status bar showing "All" selected, followed by "Errors", "Warnings", and "Logs".

Voilà, nous avons fait un rapide tour d'horizon des éléments qui nous seront nécessaires par la suite. Nous pouvons enfin commencer.

## 4 1er contact ... avec Backbone

*Sommaire*

- *Premier modèle*
- *Première collection*
- *Première vue & premier template*

*Nous allons faire un premier exemple Backbone pas à pas, même sans connaître le framework. Cela va permettre de « désacraliser » la bête et de mettre un peu de liant avec tout ce que nous avons vu précédemment. Puis nous passerons dans le détail tous les composants de Backbone dans les chapitres qui suivront.*

Voilà, il est temps de s'y mettre. L'application que nous allons réaliser avec Backbone tout au long de cet ouvrage va être un Blog, auquel nous ajouterons au fur et à mesure des fonctionnalités pour finalement le transformer en CMS (Content Management System). Je vous l'accorde ce n'est pas très original, mais cela répond à des problématiques classiques (récurrentes ?) dans notre vie “d'informaticien” et cela a le mérite d'avoir un aspect pratique et utile. Notre point de départ va être un blog que nous agrémenterons de fonctionnalités au fil des chapitres.

### 4.1 1ère application Backbone

Nous allons faire ici un exemple très rapide, sans forcément entrer dans le détail ni mettre en œuvre les bonnes pratiques d'organisation de code. Cet exercice est là pour démontrer la simplicité d'utilisation, et le code devrait être suffisamment simple pour se passer d'explications. Donc, “pas de panique !”, laissez-vous guider, dans **15 minutes** vous aurez une 1ère ébauche.

#### 4.1.1 Préparons notre page

Nous allons utiliser notre même page `index.html`, mais faisons un peu de ménage à l'intérieur avant de commencer :

```

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Backbone</title>
    <link href="libs/vendors/bootstrap/css/bootstrap.css" rel="stylesheet">

    <style>
        body {
            padding-top: 60px;
            padding-bottom: 40px;
        }
    </style>

    <link href="libs/vendors/bootstrap/css/bootstrap-responsive.css" rel="stylesheet">

</head>

<body>
    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand">Mon Blog</a>
            </div>
        </div>
    </div>

    <div class="container">

        <div class="hero-unit">
            <h1>Backbone rocks !!!</h1>
        </div>

    </div>

</body>
<!-- == Références aux Frameworks == -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>

<script>
$(function (){
```

```
});  
</script>  
</html>
```

L'essentiel de notre travail va se passer dans la balise `<script></script>` en bas de page. De quoi avons-nous besoin dans un blog ?

- Des articles : un ensemble d'articles (ou “posts”), généralement écrits par une seule personne (le blog est personnel, c'est en lui donnant des fonctionnalités multi-utilisateurs que nous nous dirigerons doucement vers un CMS).
- Des commentaires : Il est de bon ton de permettre aux lecteurs du blog de pouvoir commenter les articles.

Pour le moment nous allons nous concentrer uniquement sur les articles, notre objectif sera le suivant : “Afficher une liste d’articles sur la page principale”.

## 4.2 Le Modèle “Article”

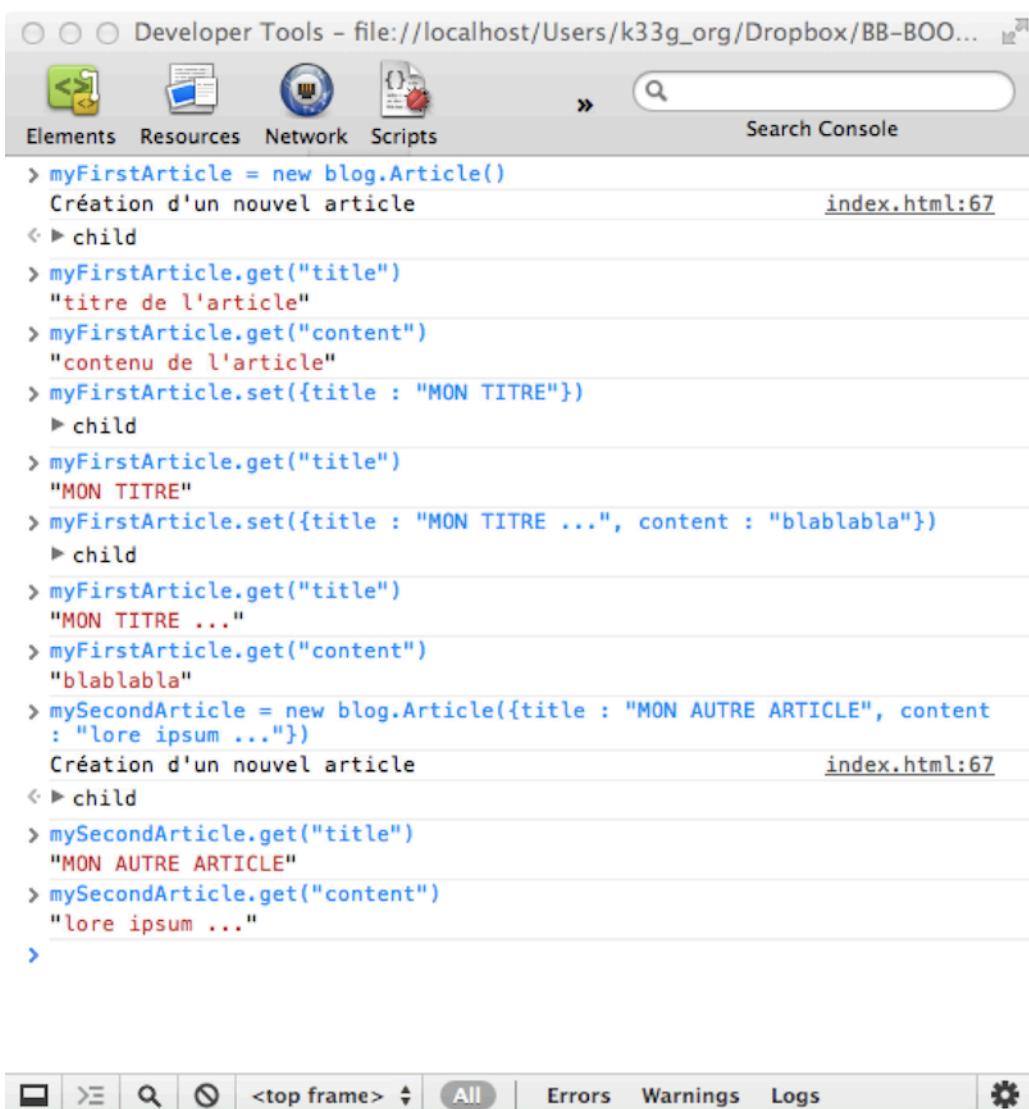
Dans la balise `<script></script>` saisissez le code suivant :

*Définition d'un modèle Article :*

```
<script>  
  
$(function (){  
    //permettra d'accéder à nos variables en mode console  
    window.blog = {};  
  
    /*--- Modèle article ---*/  
  
    // une "sorte" de classe Article  
    blog.Article = Backbone.Model.extend({  
        //les valeurs par défaut d'un article  
        defaults : {  
            title : "titre de l'article",  
            content : "contenu de l'article",  
            publicationDate : new Date()  
        },  
        // s'exécute à la création d'un article  
        initialize : function () {  
            console.log ("Création d'un nouvel article")  
        }  
    });  
  
});  
  
</script>
```

Sauvegardez, relancez dans le navigateur, et allez dans la console :

- Pour créer un nouvel article : tapez la commande `myFirstArticle = new blog.Article()`
- Pour “voir” le titre de l’article : tapez la commande `myFirstArticle.get("title")`
- Pour “voir” le contenu de l’article : tapez la commande `myFirstArticle.get("content")`
- Pour changer le titre de l’article : tapez la commande `myFirstArticle.set("title", "MON TITRE")` ou `myFirstArticle.set({title : "MON TITRE"})`
- Pour changer simultanément le titre et le contenu : tapez la commande `myFirstArticle.set({title : "MON TITRE ...", content : "blablabla"})`
- Pour créer un article directement avec un titre et du contenu : tapez la commande `mySecondArticle = new blog.Article({title : "MON AUTRE ARTICLE", content : "lore ipsum ..."})`



The screenshot shows the Network tab of the Google Chrome Developer Tools. It lists several requests made by Backbone.js:

- `myFirstArticle = new blog.Article()` - Creation d'un nouvel article (index.html:67)
- `myFirstArticle.get("title")` - titre de l'article
- `myFirstArticle.get("content")` - contenu de l'article
- `myFirstArticle.set({title : "MON TITRE"})` - child
- `myFirstArticle.get("title")` - "MON TITRE"
- `myFirstArticle.set({title : "MON TITRE ...", content : "blablabla"})` - child
- `myFirstArticle.get("title")` - "MON TITRE ..."
- `myFirstArticle.get("content")` - "blablabla"
- `mySecondArticle = new blog.Article({title : "MON AUTRE ARTICLE", content : "lore ipsum ..."})` - Creation d'un nouvel article (index.html:67)
- `mySecondArticle.get("title")` - "MON AUTRE ARTICLE"
- `mySecondArticle.get("content")` - "lore ipsum ..."

Vous venez donc de voir que nous avons défini le modèle article “un peu” comme une classe qui hériterait (`extend`) de la classe `Backbone.Model`, que nous lui avons défini des valeurs par défauts (`defaults`), et affecté une méthode d’initialisation (`initialize`). Et qu’il existe un système de getter et de setter un peu particulier (`model.get(property_name)`, `model.set(property_name, value)`), mais nous verrons ultérieurement dans le détail comment fonctionnent les modèles.

**Remarque :** le modèle de programmation de Javascript est bien orienté objet, mais n'est pas orienté "classe" comme peut l'être par exemple Java. Cela peut déstabiliser au départ, mais je vous engage à lire [REF VERS ARTICLE] à ce propos.

### 4.3 La Collection d'Articles

Nous allons maintenant définir une collection qui nous aidera à gérer nos articles. Donc, à la suite du modèle Article saisissez le code suivant :

Définition d'une collection d'articles :

```
/*--- Collection d'articles ---*/
blog.ArticlesCollection = Backbone.Collection.extend({
  model : blog.Article,
  initialize : function () {
    console.log ("Création d'une collection d'articles")
  }
});
```

**Notez** qu'il faut bien préciser le type de modèle adressé par la collection (on pourrait dire que la collection est typée).

Sauvegarder, relancer dans le navigateur, et retournez à nouveau dans la console et saisissez les commandes suivantes :

- Création de la collection :

```
listeArticles = new blog.ArticlesCollection()
```

- Ajout d'articles à la collection :

```
listeArticles.add(new blog.Article({ title : "titre1", content : "contenu1" }))
listeArticles.add(new blog.Article({ title : "titre2", content : "contenu2" }))
listeArticles.add(new blog.Article({ title : "titre3", content : "contenu3" }))
```

Nous venons donc d'ajouter 3 articles à notre collection,

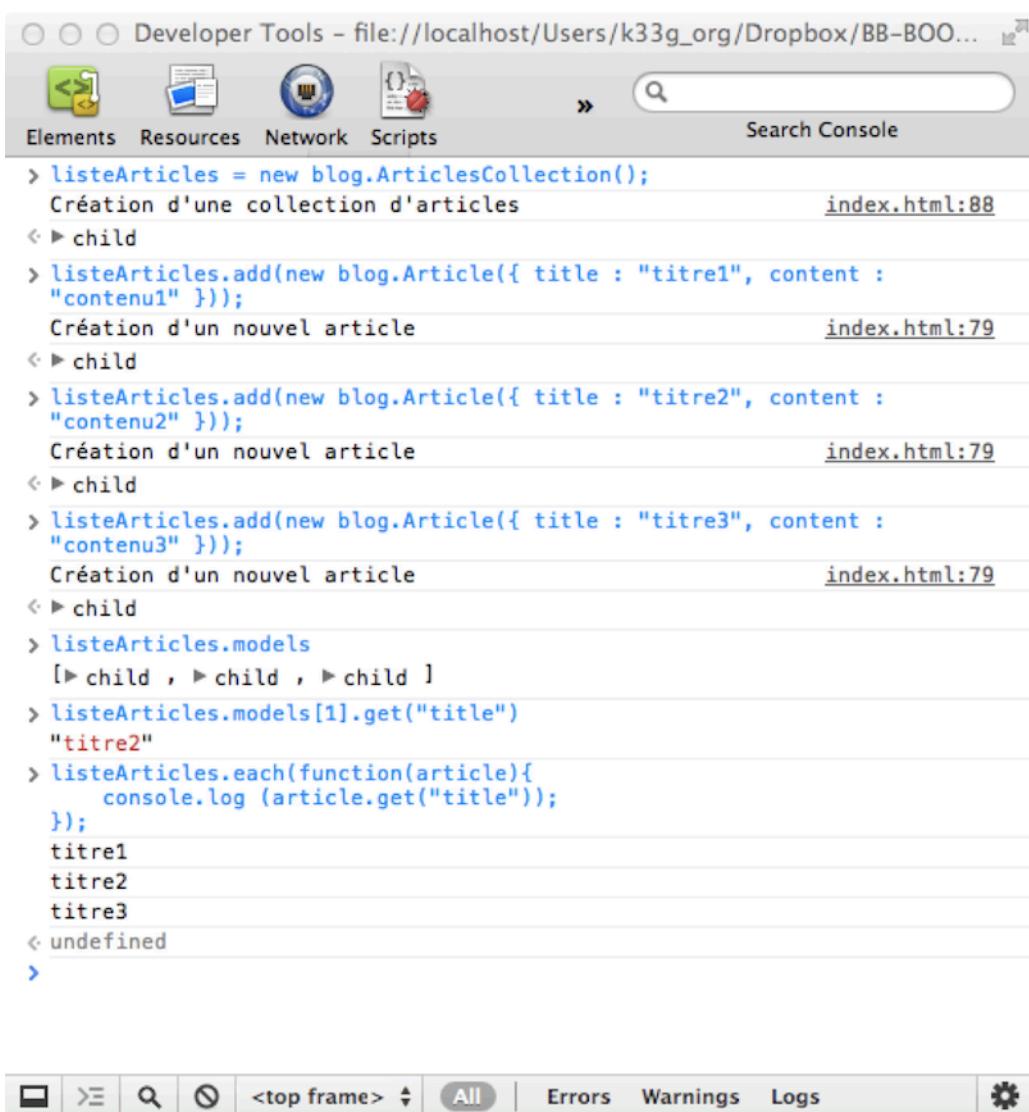
- Si vous tapez la commande `listeArticles.models` vous obtiendrez un tableau de modèles
- Si vous souhaitez obtenir le titre du 2ème article de la collection, tapez :

```
listeArticles.models[1].get("title")
```

- vous souhaitez parcourir les articles de la collection et afficher leur titre :

```
listeArticles.each(function(article){ console.log (article.get("title")); })
```

Cela vous rappelle quelque chose ? Le `each` de Backbone est implémenté grâce à Underscore.



The screenshot shows the Network tab of the Chrome Developer Tools. It lists several requests made by Backbone.js:

- listeArticles = new blog.ArticlesCollection(); (index.html:88)
- Création d'une collection d'articles
- child
- listeArticles.add(new blog.Article({ title : "titre1", content : "contenu1" })); (index.html:79)
- Création d'un nouvel article
- child
- listeArticles.add(new blog.Article({ title : "titre2", content : "contenu2" })); (index.html:79)
- Création d'un nouvel article
- child
- listeArticles.add(new blog.Article({ title : "titre3", content : "contenu3" })); (index.html:79)
- Création d'un nouvel article
- child
- listeArticles.models (index.html:79)
- [► child , ► child , ► child ]
- listeArticles.models[1].get("title")
- "titre2"
- listeArticles.each(function(article){ console.log (article.get("title"));});
- titre1
- titre2
- titre3
- undefined
- >



Maintenant que nous avons de quoi gérer nos données, il est temps de les afficher dans notre page HTML.

## 4.4 Vue et Template

Avant toute chose, allons ajouter dans notre code javascript (en bas de la page HTML) le bout de code qui va créer les articles et la collection d'articles pour nous éviter de tout re-saisir à chaque fois. Donc après le code de la collection, ajoutez ceci :

```
/*--- bootstrap ---*/
blog.listeArticles = new blog.ArticlesCollection();

blog.listeArticles.add(new blog.Article({ title : "titre1", content : "contenu1" }));
blog.listeArticles.add(new blog.Article({ title : "titre2", content : "contenu2" }));
blog.listeArticles.add(new blog.Article({ title : "titre3", content : "contenu3" }));
blog.listeArticles.add(new blog.Article({ title : "titre4", content : "contenu4" }));
blog.listeArticles.add(new blog.Article({ title : "titre5", content : "contenu5" }));
```

Ensuite dans le code html, ajoutons le template de notre vue et le div dans lequel les données seront affichées :

```
<% _.each(articles, function(article) { %>
<h1><%= article.title %></h1>
<h6><%= article.publicationDate %></h6>
<p><%= article.content %></p>
<% }); %>
```

donc :

```
<body>

<div class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <a class="brand">Mon Blog</a>
    </div>
  </div>
</div>

<div class="container">

  <div class="hero-unit">
    <h1>Backbone rocks !!!</h1>
  </div>

  <!-- ici notre template -->
  <script type="text/template" id="articles-collection-template">

    <% _.each(articles, function(article) { %>
      <h1><%= article.title %></h1>
      <h6><%= article.publicationDate %></h6>
      <p><%= article.content %></p>
    <% }); %>

  </script>
  <!-- Les données seront affichées ici -->
  <div id="articles-collection-container"></div>

</div>

</body>
```

Puis dans le code javascript, à la suite du code de la collection et avant le code de chargement des données (bootstrap), ajoutez le code de la vue Backbone :

```
/*--- Vues ---*/
blog.ArticlesView = Backbone.View.extend({
  el : $("#articles-collection-container"),

  initialize : function () {
```

```

    this.template = _.template($("#articles-collection-template").html());
  },

  render : function () {
    var renderedContent = this.template({ articles : this.collection.toJSON() });
    $(this.el).html(renderedContent);
    return this;
  }
);

```

#### 4.4.1 Qu'avons-nous fait ?

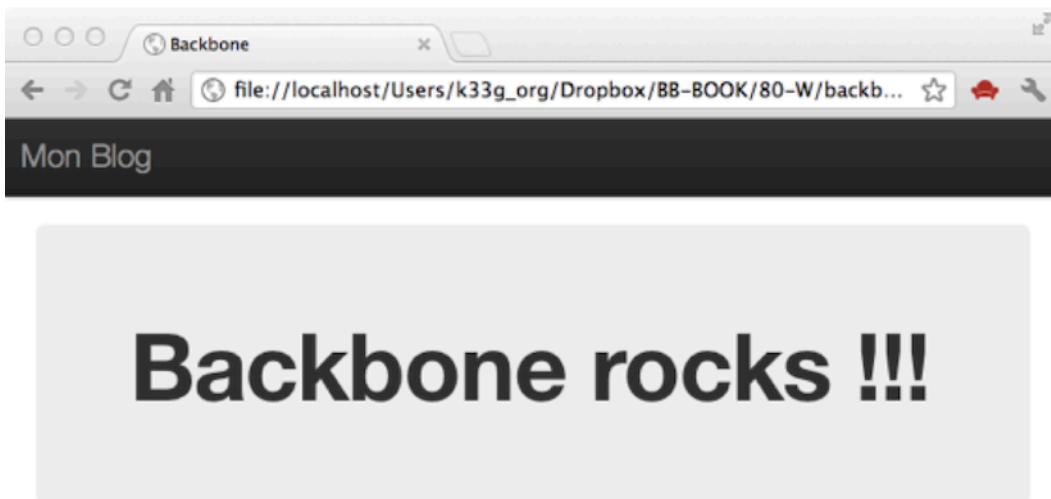
Eh bien, nous avons défini une vue avec :

- Une propriété `el` (pour élément) à laquelle on “attache” le `<div>` dont l’id est : “`articles-collection-container`”. C’est dans ce `<div>` que seront affichés les articles
- Une méthode `initialize`, qui affecte une méthode `template()` à l’instance de la vue en lui précisant que nous utiliserons le modèle de code html défini dans le `<div>` dont l’id est “`articles-collection-template`”
- Une méthode `render`, qui va passer les données en paramètre à la méthode `template()` puis les afficher dans la page

Sauvegarder, relancer dans le navigateur, et retournez encore dans la console pour saisir les commandes suivantes :

- Pour instancier une vue : `articlesView = new blog.ArticlesView({ collection : blog.listeArticles })` à laquelle nous passons la collection d’articles en paramètre
- Pour afficher les données : `articlesView.render()`

**Et là la “magie” de Backbone s’opère, vos articles s’affichent instantanément dans votre page : :)**



## titre1

SAT MAY 12 2012 07:01:23 GMT+0200 (CEST)

contenu1

## titre2

SAT MAY 12 2012 07:01:23 GMT+0200 (CEST)

contenu2

## titre3

SAT MAY 12 2012 07:01:23 GMT+0200 (CEST)

contenu3

## titre4

SAT MAY 12 2012 07:01:23 GMT+0200 (CEST)

contenu4

## titre5

**Remarque :** Notez bien que la collection doit être transformée en chaîne JSON pour être interprétée dans le template (`this.template({ articles : this.collection.toJSON() })`) et que nous avons nommé le paramètre `articles` pour faire le lien avec le template (`_.each(articles, function(article) {})`).

### 4.5 Un dernier tour de magie pour clôturer le chapitre d'initiation : “binding”

A la fin de la méthode `initialize` de la vue, ajoutez le code suivant :

```

/*--- binding ---*/
_.bindAll(this, 'render');

this.collection.bind('change', this.render);
this.collection.bind('add', this.render);
this.collection.bind('remove', this.render);
/*-----*/

```

#### 4.5.1 Que venons-nous de faire ?

Nous venons “d’expliquer” à Backbone, qu’à chaque changement dans la collection, la vue doit rafraîchir son contenu. `_.bindAll` est une méthode d’Underscore (<http://documentcloud.github.com/underscore/#bind>) qui permet de conserver le contexte initial, c’est à dire : quel que soit “l’endroit” d’où l’on appelle la méthode `render`, ce sera bien l’instance de la vue (attachée à `this`) qui sera utilisée.

```
//TODO: à expliquer plus simplement
```

Une dernière fois, sauvegarder, relancer le navigateur, et retournez encore dans la console pour saisir les commandes suivantes :

- Création de la vue: `articlesView = new blog.ArticlesView({ collection : blog.listeArticles })`
- Afficher les données : `articlesView.render()`
- Ajouter un nouvel article à la collection : `blog.listeArticles.add(new blog.Article({title:"Hello", content:"Hello World"}))`

**Et là, magique ! : L'affichage s'est actualisé tout seul :**

#### 4.5.2 Oh la vilaine erreur !!!

Si vous avez bien suivi, j’ai fait une grossière erreur (je l’ai laissé volontairement, car c’est une erreur que j’ai déjà faite, et il n’est donc pas impossible que d’autres la fassent), la date de publication ne change pas ! En effet, je l’affecte dans les valeurs par défaut qui ne sont “settées” qu’une seule et unique fois lors de la définition de la “pseudo” classe `Backbone.Model`. Il faut donc initialiser la date de publication lors de l’instanciation du modèle, et ce dans la méthode `initialize()`. Modifiez donc le code du modèle de la manière suivante :

```
/*--- Modèle article ---*/  
  
blog.Article = Backbone.Model.extend({ // une "sorte" de classe Article  
  defaults : { //Les valeurs par défaut d'un article  
    title : "titre de l'article",  
    content : "contenu de l'article",  
    //publicationDate : null  
  },  
  initialize : function () { // s'exécute à la création d'un article  
    console.log ("Création d'un nouvel article");  
    this.set("publicationDate",new Date());  
  }  
});
```

Refaites les manipulations précédentes, et là (si vous avez laissez suffisamment de temps entre la création des articles), vous pourrez noter que la date est bien mise à jour :

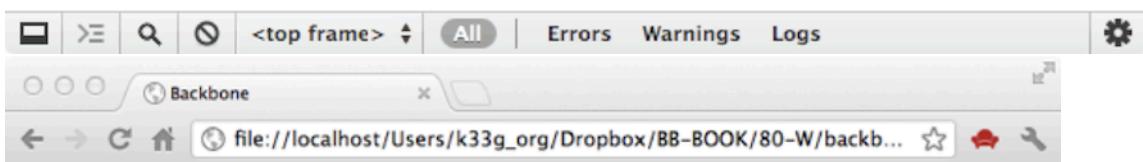
Developer Tools – file:///localhost/Users/k33g\_org/Dropbox/BB-BOOK/80-W/...

Elements Resources Network Scripts Timeline Profiles » Search Console

```

Création d'une collection d'articles index.html:85
Création d'un nouvel article index.html:75
> articlesView = new blog.ArticlesView({ collection : blog.listeArticles })
  ► child
  > articlesView.render()
    ► child
  > blog.listeArticles.add(new blog.Article({title:"Hello", content:"Hello World"}))
    Création d'un nouvel article index.html:75
    < ► child
  > blog.listeArticles.add(new blog.Article({title:"Salut", content:"Salut à tous"}))
    Création d'un nouvel article index.html:75
    < ► child
  >

```



## titre5

SAT MAY 12 2012 07:59:56 GMT+0200 (CEST)  
contenu5

## Hello

SAT MAY 12 2012 08:00:13 GMT+0200 (CEST)  
Hello World

## Salut

SAT MAY 12 2012 08:00:23 GMT+0200 (CEST)  
Salut à tous

**Remarque :** la propriété date n'existe plus dans les valeurs par défaut, elle est créée à linstanciation du modèle lors de l'appel de `this.set("publicationDate", new Date())` dans la méthode `initialize`. De la même manière, vous pouvez créer à la volée des propriétés “à posteriori” pour les instances des modèles.

**Et voilà, l'initiation est terminée. Nous allons pouvoir passer “aux choses sérieuses” et découvrir jusqu'où nous pouvons “pousser” Backbone.**

## 4.6 Code final de l'exemple

Le code final de votre page devrait ressembler à ceci :

```

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Backbone</title>
    <link href="libs/vendors/bootstrap/css/bootstrap.css" rel="stylesheet">
    <style>
        body {
            padding-top: 60px;
            padding-bottom: 40px;
        }
    </style>
    <link href="libs/vendors/bootstrap/css/bootstrap-responsive.css" rel="stylesheet">
</head>

<body>

    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand">Mon Blog</a>
            </div>
        </div>
    </div>

    <div class="container">

        <div class="hero-unit">
            <h1>Backbone rocks !!!</h1>
        </div>

        <!-- Template d'affichage des articles -->
        <script type="text/template" id="articles-collection-template">

            <% _.each(articles, function(article) { %>
                <h1><%= article.title %></h1>
                <h6><%= article.publicationDate %></h6>
                <p><%= article.content %></p>
            <% }); %>

        </script>
        <!-- div où seront affichés les articles -->
        <div id="articles-collection-container"></div>

    </div>

</body>
<!-- === Frameworks === -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
<!--<script src="libs/vendors/bootstrap/js/bootstrap.js"></script>-->

```

```

<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>

<!-- == code applicatif == -->
<script>

$(function (){
    window.blog = {};

    /*--- Modèle article ---*/

    blog.Article = Backbone.Model.extend({
        defaults : {
            title : "titre de l'article",
            content : "contenu de l'article",
        },
        initialize : function () {
            console.log ("Création d'un nouvel article");
            this.set("publicationDate", new Date());
        }
    });

    /*--- Collection d'articles ---*/

    blog.ArticlesCollection = Backbone.Collection.extend({
        model : blog.Article,
        initialize : function () {
            console.log ("Création d'une collection d'articles")
        }
    });

    /*--- Vues ---*/
    blog.ArticlesView = Backbone.View.extend({

        el : $("#articles-collection-container"),

        initialize : function () {
            this.template = _.template($("#articles-collection-template").html());

            /*--- binding ---*/
            _.bindAll(this, 'render');

            this.collection.bind('change', this.render);
            this.collection.bind('add', this.render);
            this.collection.bind('remove', this.render);
            /*-----*/
        },

        render : function () {
            var renderedContent = this.template({

```

```

        articles : this.collection.toJSON()
    });
    $(this.el).html(renderedContent);
    return this;
}
});

/*--- bootstrap ---*/
blog.listeArticles = new blog.ArticlesCollection();

blog.listeArticles.add(new blog.Article({
    title : "titre1", content : "contenu1"
}));
blog.listeArticles.add(new blog.Article({
    title : "titre2", content : "contenu2"
}));
blog.listeArticles.add(new blog.Article({
    title : "titre3", content : "contenu3"
}));
blog.listeArticles.add(new blog.Article({
    title : "titre4", content : "contenu4"
}));
blog.listeArticles.add(new blog.Article({
    title : "titre5", content : "contenu5"
}));

});
</script>
</html>
```

## 5 Le modèle objet de Backbone

*Sommaire*

- *Un petit tour dans le code*
- *Une classe de base*
- *Héritage*

*Ce qui est souvent déstabilisant pour le développeur Java (PHP, .Net, etc. ....) c'est le modèle objet de javascript qui diffère du classique modèle orienté « classes » que nous connaissons tous (normalement). De nombreux ouvrages, articles, ... se sont attaqués au sujet, mais ce n'est pas l'objet de ce chapitre.*

Je vais vous présenter de quelle façon Backbone gère son « Orientation objet » et comment réutiliser cette fonctionnalité. L'objectifs et double : Mieux comprendre le fonctionnement de Backbone et vous donner un moyen de faire de l'objet en javascript sans être dépaysé (quelque chose qui ressemble dans sa logique, à ce que vous connaissez déjà).

## 5.1 Un petit tour dans le code

Si vous avez la curiosité d'aller lire le code de Backbone (je vous engage à le faire, le code est clair et simple et avec le temps très instructif), vous « tomberez » sur une ligne particulièrement intéressante (vers la fin du code source dans `backbone.js` pour ceux qui iront réellement lire le code) :

```
// Set up inheritance for the model, collection, and view.
Model.extend = Collection.extend = Router.extend = View.extend = extend;
```

Il existe une méthode (privée) `extend` dans Backbone qui permet à un objet d'hériter des membres d'un autre objet, par exemple, si j'écris:

```
/*--- Modèle article ---*/
// une "sorte" de classe Article
var Article = Backbone.Model.extend({
});
```

Je signifie que je crée une “sorte” de classe `Article` qui hérite des fonctionnalités de `Model`. De la même façon je pourrais ensuite définir une autre classe `ArticleSpecial` qui héritera de `Article` (et qui conservera les spécificités (membres de classe) de `Model`):

```
var ArticleSpecial = Article.extend({
});
```

Je vous expliquais que la méthode `extend` était privée, Backbone ne l'expose pas directement, mais il est tout à fait possible d'y accéder par un des composants de Backbone, de la façon suivante :

```
var Kind = function() {};
Kind.extend = Backbone.Model.extend;
```

**Remarque 1 :** J'ai utilisé « `Kind` » pour ne pas utiliser « `Class` » ou « `class` » qui est un terme réservé pour les futures versions de javascript.

**Remarque 2 :** Je vais utiliser du français dans mon code. Je sais c'est moche, promis j'essaye de ne plus le faire (à part dans les commentaires)

Nous pouvons donc maintenant écrire :

```
var Personne = Kind.extend({ });
```

## 5.2 1ère “classe”

Voyons donc ce que nous apporte le modèle objet de Backbone.

### 5.2.1 Un constructeur

La déclaration d'un constructeur se fait avec le mot clé `constructor` :

Utilisation de `Kind.extend()` et définition de `constructor()`

```
var Personne = Kind.extend({
  constructor : function () {
    console.log("Bonjour, je suis le constructeur de Personne");
  }
});

var bob = new Personne();
```

Nous obtiendrons à l'exécution :

Bonjour, je suis le constructeur de Personne

### 5.2.2 Des propriétés

Les propriétés se déclarent dans le constructeur (elles sont générées à l'exécution), et vous pouvez déclarer les valeurs par défaut à l'extérieur du constructeur :

Ajout de propriétés

```
var Personne = Kind.extend({
  prenom : "John",
  nom : "Doe",
  constructor : function (prenom, nom) {
    if(prenom) this.prenom = prenom;
    if(nom) this.nom = nom;

    console.log("Bonjour, je suis ", this.prenom, this.nom);
  }
});

var john = new Personne();
var bob = new Personne("Bob", "Morane");
```

Nous obtiendrons à l'exécution :

Bonjour, je suis John Doe  
Bonjour, je suis Bob Morane

### 5.2.3 Des méthodes

Les méthodes se déclarent de la même façon que le constructeur, ajoutons une méthode `bonjour()` :

Ajout d'une méthode

```

var Personne = Kind.extend({
  prenom : "John",
  nom : "Doe",
  constructor : function (prenom, nom){
    if(prenom) this.prenom = prenom;
    if(nom) this.nom = nom;
  },
  bonjour : function () {
    console.log("Bonjour, je suis ", this.prenom, this.nom);
  }
});

var john = new Personne();
var bob = new Personne("Bob", "Morane");

john.bonjour();
bob.bonjour();

```

Nous obtiendrons à l'exécution :

```

Bonjour, je suis John Doe
Bonjour, je suis Bob Morane

```

#### 5.2.4 Des membres statiques

La méthode `extend` accepte un deuxième paramètre qui permet de déclarer des membres statiques :  
Ajout & utilisation de membres statiques

```

var Personne = Kind.extend({
  prenom : "John",
  nom : "Doe",
  constructor : function (prenom, nom){
    if(prenom) this.prenom = prenom;
    if(nom) this.nom = nom;

    //Utilisation de la propriété statique
    Personne.compteur += 1;
  },
  bonjour : function () {
    console.log("Bonjour, je suis ", this.prenom, this.nom);
  }
}, { //ici Les membres statiques
  compteur : 0,
  combien : function () {
    return Personne.compteur;
  }
});

```

```
var john = new Personne();
var bob = new Personne("Bob", "Morane");

console.log("Il y a ", Personne.combien(), " personnes");
```

Nous avons donc une propriété statique `compteur` et une méthode statique `combien()`. Nous obtiendrons ceci à l'exécution :

```
Il y a 2 personnes
```

### 5.3 Sans héritage point de salut ! ... ?

Même s'il ne faut pas abuser de l'héritage en programmation objet (mais c'est un autre débat), il faut avouer que cela peut être pratique pour la structuration de notre code. Dès le départ, dans ce chapitre nous avons fait de l'héritage :

```
var Personne = Kind.extend({});
```

Donc `Personne` hérite de `Kind`. Mais essayons un exemple plus complet pour bien appréhender les possibilités de Backbone. `Personne` héritant de `Kind` possède donc aussi une méthode `extend`, nous allons donc pouvoir créer une `Femme` et un `Homme` :

```
var Homme = Personne.extend({
  sexe : "male"
});

var Femme = Personne.extend({
  prenom : "Jane",
  sexe : "femelle"
});

var jane = new Femme();
var john = new Homme();

var angelina = new Femme("Angelina", "Jolie");
var bob = new Homme("Bob", "Morane");

jane.bonjour();
john.bonjour();

angelina.bonjour();
bob.bonjour();

console.log("Il y a ", Personne.combien(), " personnes");
```

A l'exécution nous obtiendrons ceci :

```
Bonjour, je suis Jane Doe
Bonjour, je suis John Doe
Bonjour, je suis Angelina Jolie
Bonjour, je suis Bob Morane
Il y a 4 personnes
```

Nous pouvons donc vérifier que l'on a bien hérité de la méthode `bonjour()`, du constructeur `constructor()` et de `nom` et `prenom` (ainsi que de leurs valeurs par défaut). Nous remarquons aussi que l'incrémentation des “personnes” continue puisque `Homme` et `Femme` héritent de `Personne`. Voyons maintenant, comment surcharger les méthodes du parent et continuer à appeler les méthodes du parent.

## 5.4 Surcharge & super

Modifions le code des “pseudo classes” de la façon suivante :

Surcharge et utilisation de `super()`

```
var Homme = Personne.extend({
  sexe : "male",
  //surcharge du constructeur
  constructor : function (prenom, nom) {
    //appeler le constructeur de Personne
    Homme.__super__.constructor.call(this, prenom, nom);
    console.log("Hello, je suis un ", this.sexe);
  },
  bonjour : function () {
    //appeler la methode bonjour() du parent
    Homme.__super__.bonjour.call(this);
    console.log("Bonjour, je suis un garçon");
  }
});

var Femme = Personne.extend({
  prenom : "Jane",
  sexe : "femelle",
  //surcharge du constructeur
  constructor : function (prenom, nom) {
    //appeler le constructeur de Personne
    Femme.__super__.constructor.call(this, prenom, nom);
    console.log("Hello, je suis une ", this.sexe);
  },
  bonjour : function () {
    //appeler la methode bonjour() du parent
    Femme.__super__.bonjour.call(this);
    console.log("Bonjour, je suis une fille");
  }
});

var angelina = new Femme("Angelina", "Jolie");
```

```
var bob = new Homme("Bob", "Morane");

angelina.bonjour();
bob.bonjour();
```

Nous avons surchargé les constructeurs pour pouvoir afficher un message au moment de l'instanciation et nous avons appelé le constructeur du parent pour continuer à permettre l'affectation de `nom` et `prenom`. Nous avons appliqué le même principe pour la méthode `bonjour()`. Donc l'appel d'une méthode du parent se fait de la manière suivante : `Nom_de_la_classe_courante.__super__.methode.call(this, paramètres)`.

A l'exécution nous obtiendrons donc :

```
Hello, je suis une femelle
Hello, je suis un male
Bonjour, je suis Angelina Jolie
Bonjour, je suis une fille
Bonjour, je suis Bob Morane
Bonjour, je suis un garçon
```

## 5.5 Conclusion

Nous venons de voir comment continuer à programmer objet sans trop bouleverser vos habitudes (cela ne doit pas vous empêcher d'étudier le modèle objet de javascript plus en profondeur). Cela va vous permettre de mieux structurer votre code (et en javascript, c'est important) mais aussi vos idées, de comprendre le fonctionnement de Backbone, mais de pouvoir aussi écrire des extensions à Backbone plus facilement.

# 6 Il nous faut un serveur !

## Sommaire

- *Petit rappel sur les requêtes http*
- *Installations des composants nécessaires*
- *Construction et test de notre serveur d'application*

*Faisons un dernier détour avant de revenir à Backbone. Pour bien en comprendre le fonctionnement, nous allons nous mettre en situation réelle. Ne laissons pas notre future webapp toute seule. Généralement une application web comporte une partie serveur qui sert à distribuer des données vers l'IHM client (dans le navigateur). Pour que le tour de Backbone.js soit complet, il serait impensable de ne pas étudier les interactions avec un serveur (interrogation de données, ajax,...)*

Pour ce chapitre, je me suis longuement posé la question : « quelle technologie serveur utiliser ? ». PHP, Ruby, Java, .Net,... ? C'était aussi prendre le risque de vous désintéresser complètement. D'un autre côté, je souhaitais que vous puissiez rapidement entrer dans le vif du sujet. J'ai donc finalement choisi de vous faire utiliser Node.js puisque c'est aussi du Javascript et que sa mise en œuvre est rapide sans

pour autant être obligé de connaître Node.js dans son ensemble. **Objectif : disposer d'un serveur d'application fournissant des services JSON en moins d'une demi-heure !**

Nous aurons besoin de :

- **Node.js** pour le serveur d'application
- **Express.js** qui vas nous permettre de construire notre application (gestion des routes, sessions, etc.)
- **nStore** : une petite base de données noSQL simulée avec un fichier texte (nous n'allons pas nous embêter à faire des requêtes SQL, et le noSQL est à la mode ;))

## 6.1 Principes http : GET, POST, PUT, DELETE

Mon but est de faire une application web avec Node & Express sur les principes REST (Representational State Transfer) qui permettra de faire des opérations de type **CRUD** avec les modèles Backbone en utilisant des services basés sur le protocole http avec les verbes suivants :

- **Create** : POST
- **Read** : GET
- **Update** : PUT
- **Delete** : DELETE

Si cela vous paraît obscur, pas d'inquiétude, la partie pratique qui suit devrait vous éclairer. Mais je vous engage fortement à lire <http://naholyr.fr/2011/08/ecrire-service-rest-nodejs-express-partie-1/> de [@naholyr](#).

//TODO: vois si ça nécessite d'être développé

## 6.2 Installation(s)

### 6.2.1 Installer Node.js

Tout d'abord, allez sur le site <http://nodejs.org/>, si vous êtes sous OSX ou Windows, vous avez de la chance, il existe des installateurs tout prêts, si vous êtes sous Linux, les manipulations ne sont pas compliquées, je vous engage à lire ceci : <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>. Une fois les installateurs utilisés (ou les manipulations Linux) vous disposerez de Node.js ainsi que du gestionnaire de packets NPM qui va nous permettre d'installer de nombreux modules pour Node.js.

### 6.2.2 Installer Express.js

Express.js est un framework qui se greffe sur Node.js et vous permet de réaliser rapidement des applications web, en vous apportant quelques facilités comme la gestion des routes, des sessions, etc. ...

Commençons à créer notre application serveur. Créez un répertoire `blog` sur votre disque dur. Quel que soit votre système d'exploitation, ouvrez une console ou un terminal et tapez les commandes suivante (et validez):

```
cd blog
npm install express
```

**Remarque 1 :** sous OSX ou linux vous devrez probablement passer en mode super utilisateur, faites donc précéder la commande par `sudo` : `sudo npm install express`

Nous venons donc d'installer le module `express` dans notre répertoire `blog`.

**Remarque 2 :** il y a d'autres méthodes pour créer une application avec express, mais dans notre cas j'ai besoin du strict minimum.

### 6.2.3 Installer nStore

Nous aurons besoin d'un moyen de sauvegarde de nos données. Pour cela nous allons utiliser **nStore** qui est une sorte de base de données NoSQL clé/valeur pour node.js (il existe de nombreuses autre solutions telle MongoDB, CouchDB, des bases de donnée relationnelles,... mais ce n'est pas l'objet de cet ouvrage). Pour installer nStore (toujour dans le répertoire `blog`) tapez la commande suivante :

```
npm install nstore
```

**Remarque :** vous pouvez noter maintenant la présence d'un répertoire `node_modules` dans `blog`, contenant lui-même deux sous répertoires `express` et `nstore`.

Voilà, nous avons tout ce qu'il faut pour commencer à créer notre application.

## 6.3 Codons notre application serveur

Notre application se découpe en 2 parties :

- une partie statique, c'est là que viendra tout ce "qui touche" à Backbone
- une partie dynamique, notre serveur, ce sera notre "fournisseur" de données

### 6.3.1 Ressources statiques

Tout d'abord vous devez créer un sous-répertoire `public`, où nous allons copier les ressources statiques de notre application. Pour cela, utilisez les fichiers dont nous nous sommes servis pour notre exemple de découverte, et copiez les fichiers ci-dessous dans le répertoire `public` :

- `libs/vendors/backbone.js`
- `libs/vendors/underscore.js`
- `libs/vendors/jquery-1.7.2.js` (*ou une version plus récente*)

Et copiez aussi le répertoire `bootstrap` de notre exemple. Ensuite, préparez une page `index.html` dans le répertoire `public`, avec le code suivant :

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Backbone</title>
    <link href="libs/vendors/bootstrap/css/bootstrap.css" rel="stylesheet">
    <style>
        body {
            padding-top: 60px;
            padding-bottom: 40px;
        }
    </style>
    <link href="libs/vendors/bootstrap/css/bootstrap-responsive.css" rel="stylesheet">
</head>

<body>

    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand">Mon Blog</a>
            </div>
        </div>
    </div>

    <div class="container">
        <div class="hero-unit">
            <h1>Backbone rocks !!!</h1>
        </div>
    </div>

</body>
<!-- == Frameworks == -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
```

```

<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>

<!-- == code applicatif == -->
<script>
</script>
</html>

```

### 6.3.2 Ressources dynamiques

Toujours dans le répertoire `public`, créer un fichier `app.js` qui sera le programme principal de notre application et qui contiendra le code suivant :

**Remarque** : ce n'est pas grave si vous ne comprenez pas le code à ce stade, l'important c'est que cela fonctionne. Mais vous allez voir, à l'utilisation « tout s'éclaire ».

Rapidement, le code serveur comporte :

- l'intialisation de la base de données des posts et celle des users
- 6 routes :
  - `'/blogposts'` (GET) : pour récupérer tous les posts du blog
  - `'/blogposts/query/:query'` (GET) : pour récupérer certains posts du blog
  - `'/blogposts/:id'` (GET) : pour récupérer un post
  - `'/blogposts'` (POST) : pour créer un nouveau post
  - `'/blogposts/:id'` (PUT) : pour modifier un post existant
  - `'/blogposts/:id'` (DELETE) : pour supprimer un post

```

/*
-----Déclaration des librairies-----
*/
var express = require('express')
  , nStore = require('nstore')
  , app = module.exports = express.createServer();

nStore = nStore.extend(require('nstore/query'))();

/*
-----Paramétrages de fonctionnement d'Express-----
*/
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.static(__dirname + '/public'));
app.use(express.cookieParser('ilovebackbone'));
app.use(express.session({ secret: "ilovebackbone" }));

```

```

/*
----- Définition des "bases" posts & users -----
*/
var posts, users;

posts = nStore.new("blog.db", function() {
    users = nStore.new("users.db", function() {
        /*
            une fois les bases ouvertes, on passe
            en attente de requête http (cf. code de
            la fonction Routes())
            Si les bases n'existent pas,
            elles sont créées automatiquement
        */
        Routes();
        app.listen(3000);
        console.log('Express app started on port 3000');

    });
});

function Routes() {

/*
    Obtenir la liste de tous les posts lorsque
    l'on appelle http://localhost:3000/blogposts
    en mode GET
*/
app.get('/blogposts',function(req, res){
    console.log("GET (ALL) : /blogposts");
    posts.all(function(err, results) {

        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        } else {
            var posts = [];
            for(var key in results) {
                var post = results[key]; post.id = key;
                posts.push(post);
            }
            res.json(posts);
        }
    });
});

/*
    Obtenir la liste de tous les posts correspondant à un critère
    lorsque l'on appelle http://localhost:3000/blogposts/query/
    en

```

```

mode GET avec une requête en paramètre
ex : query : { "title" : "Mon 1er post" } }
*/
app.get('/blogposts/query/:query',function(req, res){
  console.log("GET (QUERY) : /blogposts/query/" + req.params.query);

  posts.find(JSON.parse(req.params.query), function(err, results) {
    if(err) {
      console.log("Erreur : ",err);
      res.json(err);
    } else {
      var posts = [];
      for(var key in results) {
        var post = results[key]; post.id = key;
        posts.push(post);
      }
      res.json(posts);
    }
  });
});

/*
Retrouver un post par sa clé unique lorsque
L'on appelle http://localhost:3000/blogposts/identifiant_du_post
en mode GET
*/
app.get('/blogposts/:id', function(req, res){
  console.log("GET : /blogposts/" + req.params.id);
  posts.get(req.params.id, function(err, post, key) {
    if(err) {
      console.log("Erreur : ",err);
      res.json(err);
    } else {
      post.id = key;
      res.json(post);
    }
  });
});

/*
Créer un nouveau post lorsque
L'on appelle http://localhost:3000/blogpost
avec en paramètre le post au format JSON
en mode POST
*/
app.post('/blogposts',function(req, res){
  console.log("POST CREATE ", req.body);
}

```

```

var d = new Date(), model = req.body;
model.saveDate = (d.valueOf());

posts.save(null,model, function (err, key){
    if(err) {
        console.log("Erreur : ",err);
        res.json(err);
    } else {
        model.id = key;
        res.json(model);
    }
});

/*
Mettre à jour un post lorsque
l'on appelle http://localhost:3000/blogpost
avec en paramètre le post au format JSON
en mode PUT
*/
app.put('/blogposts/:id',function(req, res){
    console.log("PUT UPDATE", req.body, req.params.id);

    var d = new Date(), model = req.body;
    model.saveDate = (d.valueOf());

    posts.save(req.params.id,model, function (err, key){
        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        } else {
            res.json(model);
        }
    });
});

/*
supprimer un post par sa clé unique lorsque
l'on appelle http://localhost:3000/blogpost/identifiant_du_post
en mode DELETE
*/
app.delete('/blogposts/:id',function(req, res){
    console.log("DELETE : /delete/"+req.params.id);

    posts.remove(req.params.id, function(err){
        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        }
    });
});

```

```

    } else {
        //petit correctif de contournement (bug ds nStore) :
        //ré-ouvrir la base lorsque la suppression a été faite
        posts = nStore.new("blog.db", function() {
            res.json(req.params.id);
            //Le modèle est vide si on ne trouve rien
        });
    });
});
}

```

Maintenant, nous allons faire un dernier travail avant de revenir à Backbone : Nous allons vérifier que notre serveur d'application fonctionne. Pour le lancer : en mode console, allez dans le répertoire blog et lancez la commande node app.js.

**Astuce :** plutôt que de devoir arrêter et relancer votre application à chaque modification, installez **nodemon** : `npm install -g nodemon`, dorénavant pour lancer votre application web, tapez `nodemon app.js` au lieu de `node app.js`, et elle se relancera toute seule à chaque fois que nodemon détectera un changement dans vos fichiers (au moment de la sauvegarde).

## 6.4 Testons notre application serveur

Une fois notre application lancée, ouvrez un navigateur, appelez l'url <http://localhost:3000> et ouvrez la console de debug du navigateur. Et c'est parti pour les tests, où nous allons utiliser intensivement les fonctionnalités ajax de jQuery. Rappelez-vous, nous l'avons inclus(e) dans notre projet, via notre page `index.html`, et au début de notre code dans `app.js`, nous avons la ligne suivante : `app.use(express.static(__dirname + '/public'));`, donc si à l'appel de <http://localhost:3000>, le serveur ne trouve pas de route `"/"`, il nous dirigera directement vers `index.html`.

### 6.4.1 Ajoutons un enregistrement

Dans la console tapez ceci (et validez) :

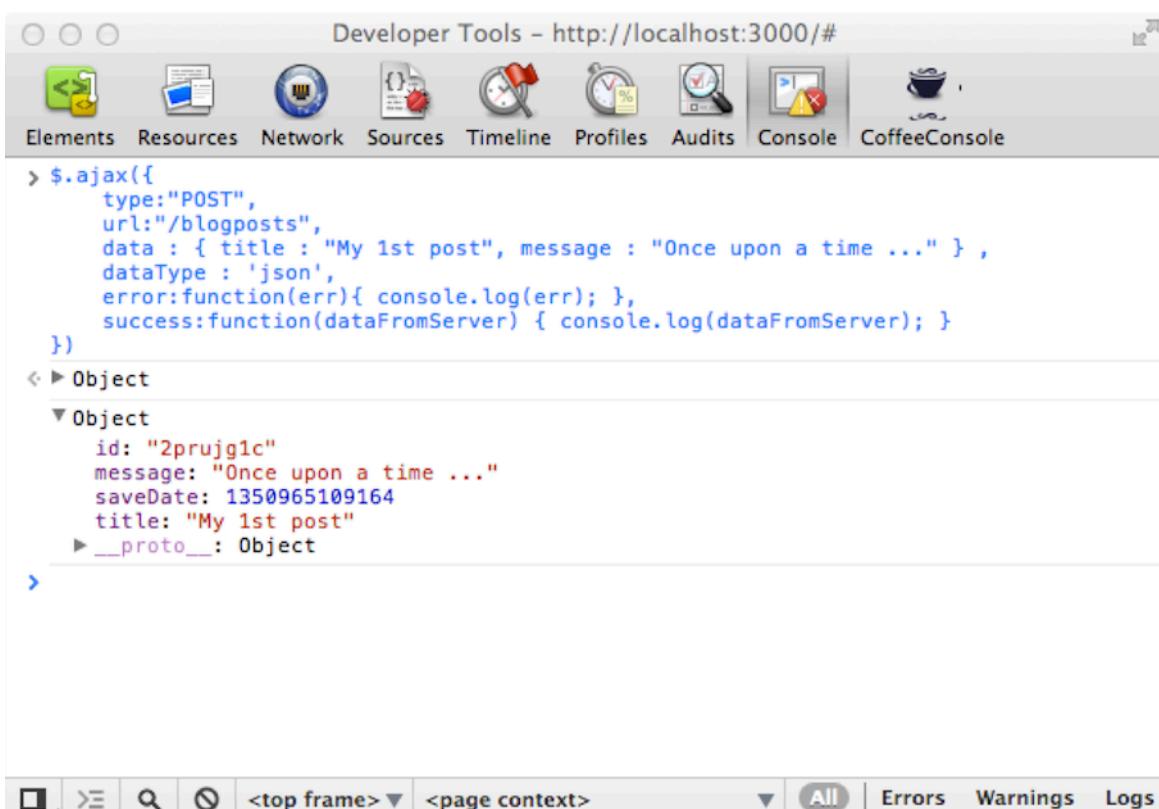
*Requête http de type POST :*

```

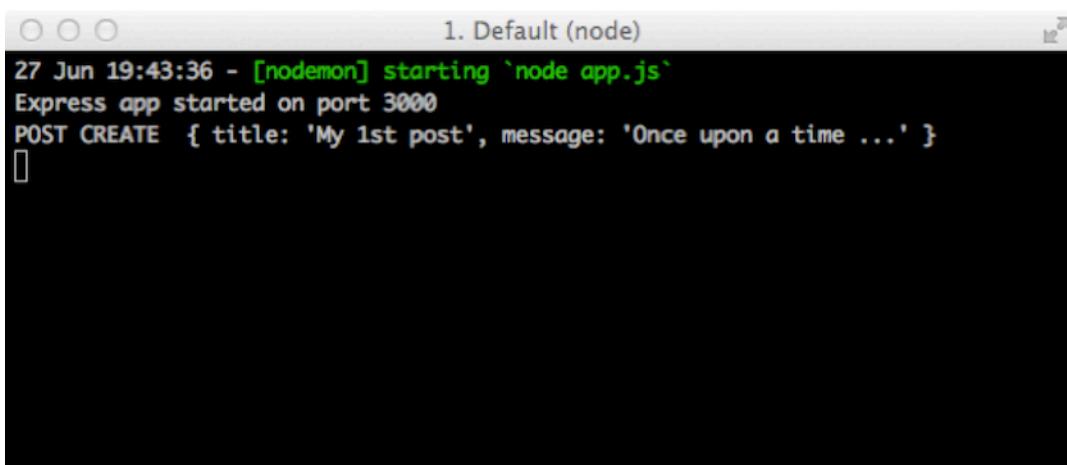
$.ajax({
    type:"POST",
    url:"/blogposts",
    data : { title : "My 1st post", message : "Once upon a time ..." } ,
    dataType : 'json',
    error:function(err){ console.log(err); },
    success:function(dataFromServer) { console.log(dataFromServer); }
})

```

Vous venez donc de créer votre tout 1er post (vous avez appelé la route '`/blogposts`' de type POST), et vous devriez obtenir ceci dans la console du navigateur :



Vous noterez que vous obtenez en retour le numéro de clé unique de votre enregistrement (généré par la base nStore) et aussi une date de sauvegarde. **Renouvelez l'opération plusieurs fois pour ajouter plusieurs enregistrements (si, si, faites le)**. De même dans le terminal, vous noterez l'apparition du message POST CREATE, nous avons donc bien appelé la “route” `/blogposts` de type POST :



#### 6.4.2 Obtenir tous les enregistrements

Dans la console tapez ceci :

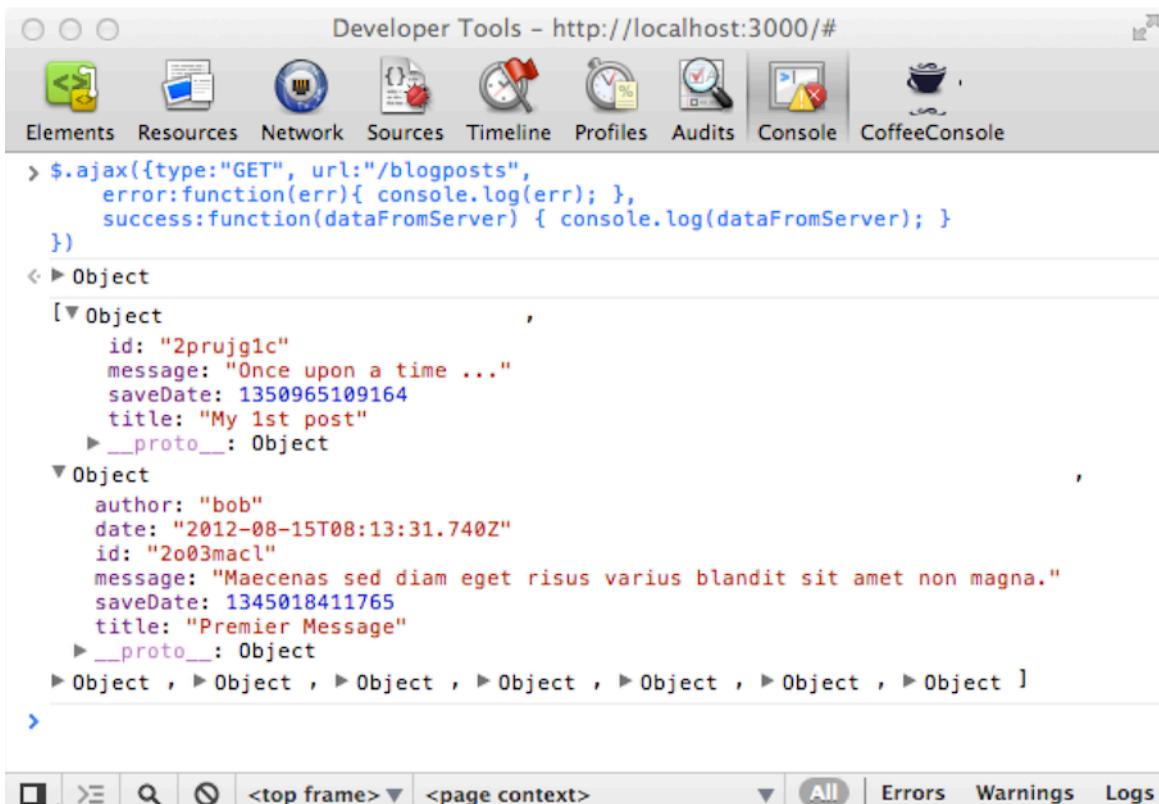
*Requête http de type GET :*

```

$.ajax({type:"GET", url:"/blogposts",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})

```

Vous obtenez un tableau d'objets correspondant à nos enregistrements :



```

Developer Tools - http://localhost:3000/#

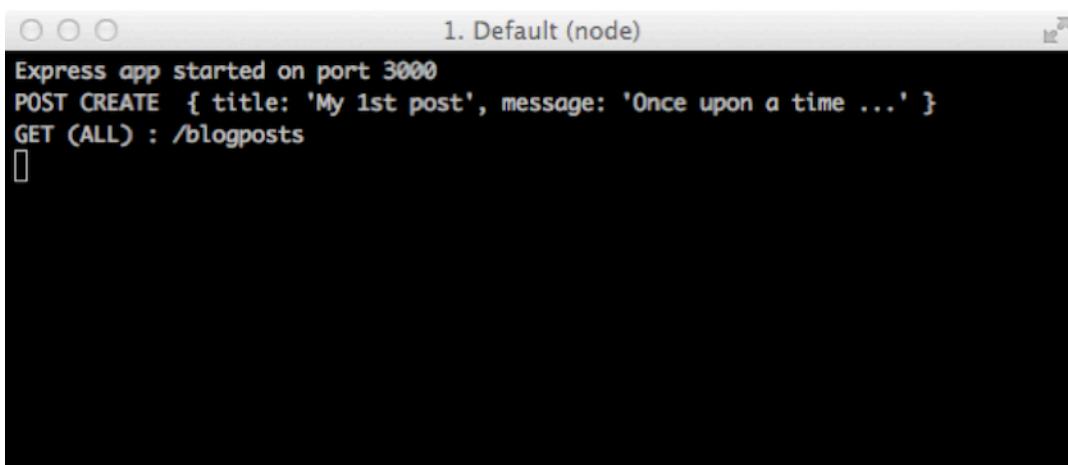
Elements Resources Network Sources Timeline Profiles Audits Console CoffeeConsole

> $.ajax({type:"GET", url:"/blogposts",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})
< ▶ Object
  [▼ Object
    id: "2prujg1c"
    message: "Once upon a time ..."
    saveDate: 1350965109164
    title: "My 1st post"
    ► __proto__: Object
  ▼ Object
    author: "bob"
    date: "2012-08-15T08:13:31.740Z"
    id: "2o03macl"
    message: "Maecenas sed diam eget risus varius blandit sit amet non magna."
    saveDate: 1345018411765
    title: "Premier Message"
    ► __proto__: Object
  ▶ Object , ▶ Object ]
>

```

The screenshot shows the Developer Tools Console tab with the URL `http://localhost:3000/#`. The console output displays an array of objects representing blog posts. Each object has properties: `id`, `message`, `saveDate`, and `title`. The first object's `__proto__` is shown, and the second object's `__proto__` is also shown. The array ends with seven more entries, indicated by the ellipsis.

De même dans le terminal, vous noterez l'apparition du message GET (ALL), nous avons donc bien appelé la “route” `/blogposts` de type GET :



```

1. Default (node)

Express app started on port 3000
POST CREATE  { title: 'My 1st post', message: 'Once upon a time ...' }
GET (ALL) : /blogposts

```

The screenshot shows a terminal window titled "1. Default (node)". It displays the output of an Express.js application. The application has started on port 3000. A POST request was made to create a new blog post with the title "My 1st post" and message "Once upon a time ...". A GET request was then made to retrieve all blog posts, resulting in the response "/blogposts".

#### 6.4.3 Retrouver un enregistrement particulier (par sa clé)

Dans la console tapez ceci (vous remarquerez que j'utilise une des clés d'enregistrement):

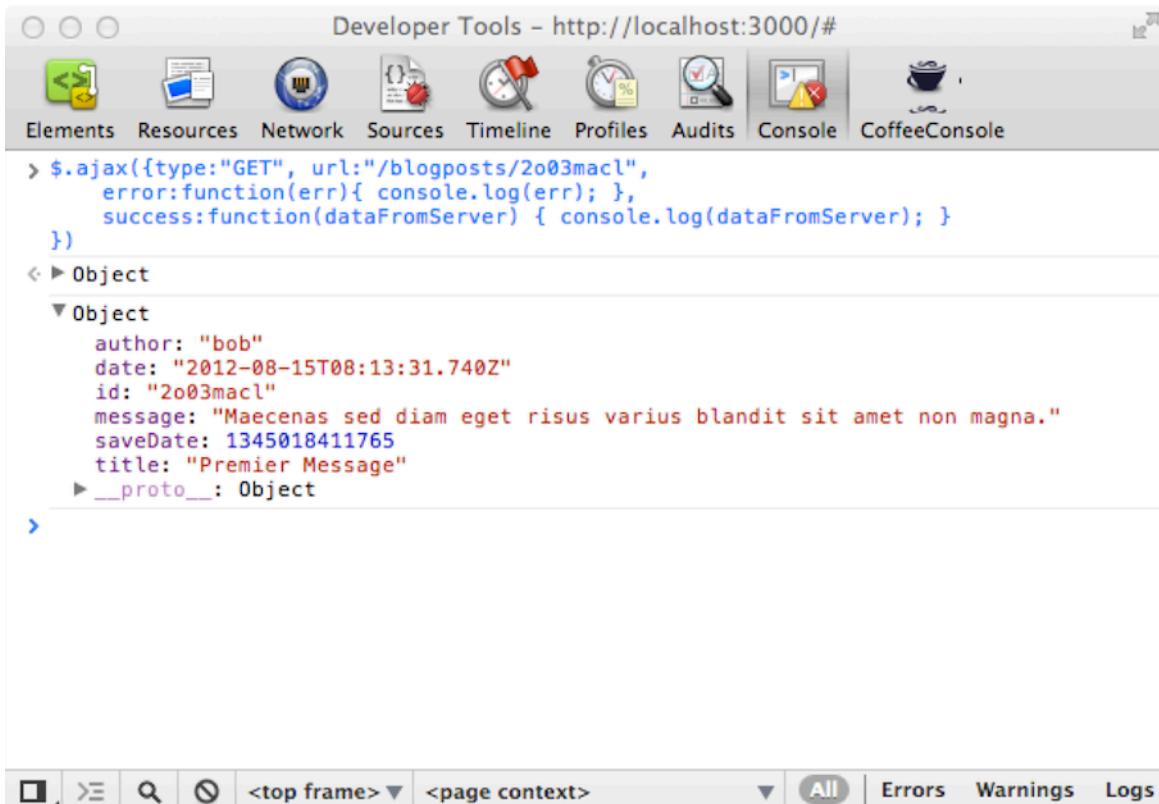
*Requête http de type GET :*

```

$.ajax({type:"GET", url:"/blogposts/2o03macl",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})

```

Vous obtenez :



```
Developer Tools - http://localhost:3000/#

Elements Resources Network Sources Timeline Profiles Audits Console CoffeeConsole

> $.ajax({type:"GET", url:"/blogposts/2o03macl",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})
< ► Object
  ▼ Object
    author: "bob"
    date: "2012-08-15T08:13:31.740Z"
    id: "2o03macl"
    message: "Maecenas sed diam eget risus varius blandit sit amet non magna."
    saveDate: 1345018411765
    title: "Premier Message"
    ► __proto__: Object
>
```

De même dans le terminal, vous obtiendrez le message GET : /blogposts/2o03macl, nous avons donc bien appelé la “route” /blogposts de type GET avec la clé du modèle en paramètre.

#### 6.4.4 Mettre à jour un enregistrement

Dans la console tapez ceci :

*Requête http de type PUT :*

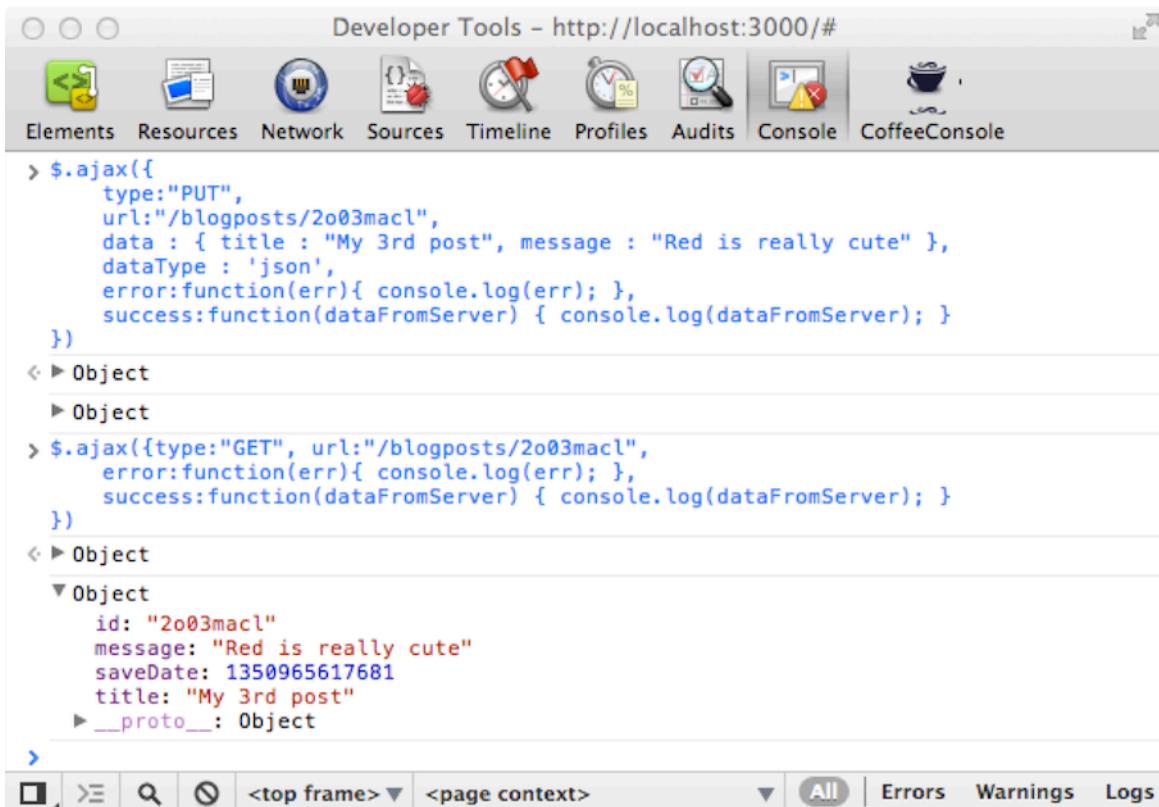
```
$.ajax({
  type:"PUT",
  url:"/blogposts/2o03macl",
  data : { title : "My 3rd post", message : "Red is really cute" },
  dataType : 'json',
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})
```

Puis appelez à nouveau pour vérifier :

*Requête http de type GET :*

```
$.ajax({type:"GET", url:"/blogposts/2o03macl",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})
```

La mise à jour a bien été prise en compte :



The screenshot shows the Developer Tools console in Google Chrome. The title bar says "Developer Tools - http://localhost:3000/#". Below it is a toolbar with icons for Elements, Resources, Network, Sources, Timeline, Profiles, Audits, Console, and CoffeeConsole. The main area of the console shows the following JavaScript code and its execution results:

```

> $.ajax({
    type:"PUT",
    url:"/blogposts/2o03macl",
    data : { title : "My 3rd post", message : "Red is really cute" },
    dataType : 'json',
    error:function(err){ console.log(err); },
    success:function(dataFromServer) { console.log(dataFromServer); }
})
<▶ Object
▶ Object
> $.ajax({type:"GET", url:"/blogposts/2o03macl",
    error:function(err){ console.log(err); },
    success:function(dataFromServer) { console.log(dataFromServer); }
})
<▶ Object
▼ Object
  id: "2o03macl"
  message: "Red is really cute"
  saveDate: 1350965617681
  title: "My 3rd post"
  ▶ __proto__: Object
>

```

At the bottom of the console, there are buttons for <top frame>, <page context>, and tabs for All, Errors, Warnings, and Logs. The "All" tab is selected.

De même, vous pouvez vérifier dans le terminal, l'apparition des messages correspondant à chacune de nos requêtes.

#### 6.4.5 Faire une requête

Dans la console tapez la commande “ajax” ci-dessous (*je veux les posts dont le titre est égal à “My 3rd post”*) :

*Requête http de type GET :*

```

$.ajax({
  type:"GET",
  url:'/blogposts/query/{ "title" : "My 3rd post"}',
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})

```

Vous obtenez :



Une fois de plus, vous pouvez vérifier dans le terminal, l'apparition du message GET (QUERY), nous avons donc bien appelé la “route” /blogposts de type GET (avec les paramètres de requête en paramètres).

#### 6.4.6 Supprimer un enregistrement

Supprimons l'enregistrement qui a la clé d'id égale à `2o03macl` (*chez vous c'est peut-être autre chose*). Dans la console tapez ceci :

*Requête http de type DELETE :*

```

$.ajax({
  type:"DELETE",
  url:"/blogposts/2o03macl",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})

```

Puis recherchez à nouveau l'enregistrement :

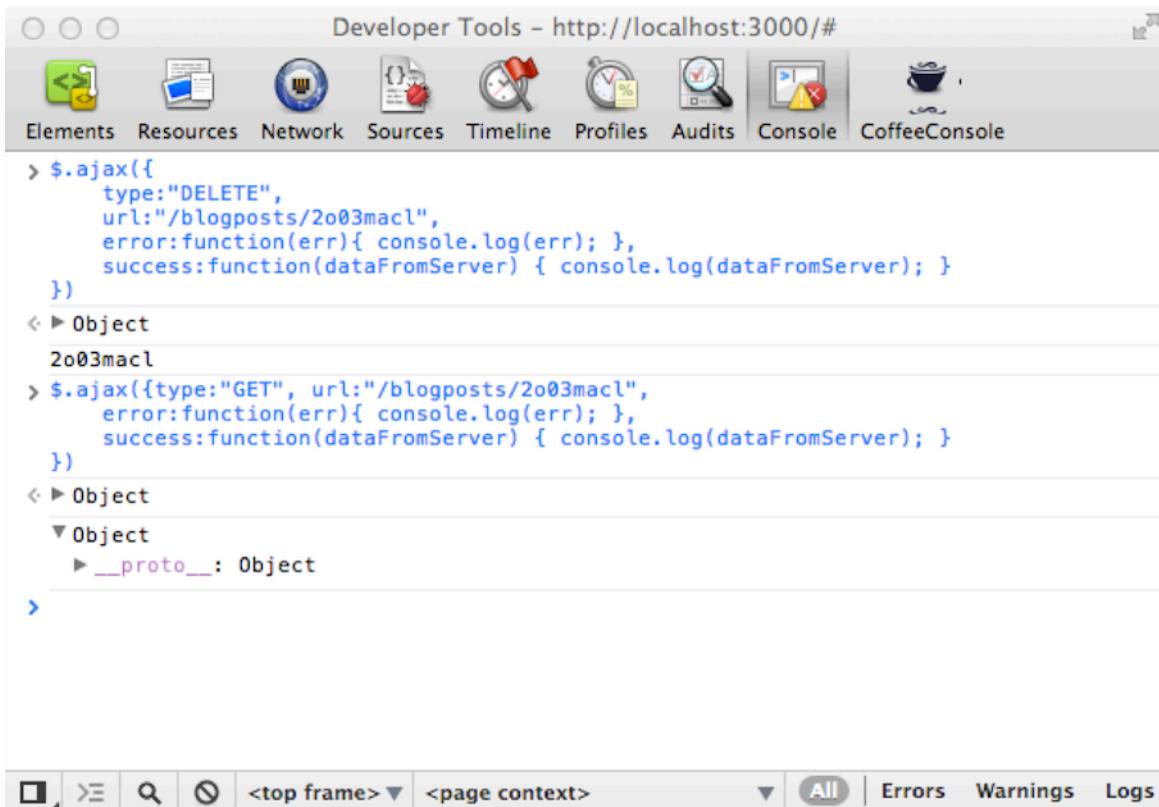
*Requête http de type GET :*

```

$.ajax({type:"GET", url:"/blogposts/2o03macl",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})

```

Vous obtenez un objet vide :



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The console output displays the following JavaScript code and its execution results:

```

> $.ajax({
  type:"DELETE",
  url:"/blogposts/2o03macl",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})
< Object
2o03macl
> $.ajax({type:"GET", url:"/blogposts/2o03macl",
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})
< Object
  < Object
    > __proto__: Object
>

```

The browser's status bar at the bottom shows various developer tools icons and tabs, with 'All' selected.

De même dans le terminal, vous noterez l'apparition du message DELETE puis GET, nous avons donc bien appelé la “route” /blogpost de type DELETE (puis GET) avec la clé du modèle en paramètre. Puis un message d’erreur apparaît car le document (post) n’existe plus.

La base de notre application côté serveur est donc posée. Nous la ferons évoluer en fonction de nos besoins. **Maintenant, passons aux choses sérieuses :-).**

**Remarque :** Vous avez remarqué que l'url dans le cas de la création, la sauvegarde, et la suppression d'un modèle, ne changeait pas. Ce qui fait la distinction c'est le verbe http (GET, POST, PUT, DELETE).

## 7 Les modèles et les collections en détail

*Sommaire*

- Fonctionnement général
- Les modèles
- Les collections

Nous allons voir comment définir nos modèles, jouer avec, interagir avec le serveur. Nous allons étudier l'intérêt d'une collection de modèles. **Attention, pas un gramme d'HTML (ou presque) dans ce chapitre, nous faisons tout en « mode commande », pour l'HTML il faudra patienter jusqu'au chapitre sur les Vues.**

Dans une application « de gestion », les modèles sont le cœur de l'application, ils représentent des concepts « informatisés » de la « vraie vie » : Les articles d'un catalogue, le client d'une entreprise, ... Ils

peuvent avoir des interactions entre eux : un client a plusieurs facture, la commande d'un fournisseur ; ... Nous avons là 4 modèles : client, facture, fournisseur, commande ... sans parler des articles de la facture et de la commande. D'ailleurs en parlant d'article, on pense catalogue, et dans notre cas le catalogue pourrait être une collection d'articles. Et tout cela doit être sauvegardé, doit pouvoir être retrouvé facilement, etc. ... Mais voyons donc le fonctionnement intrinsèque des modèles et collections de Backbone.

## 7.1 Fonctionnement général

Un modèle Backbone (`Backbone.Model`) représente une entité unique (une instance du modèle), par exemple nous avons la définition du modèle « Client », et si Bob est un client, alors c'est une instance de client. En général, il est lié à une vue (un composant d'affichage) qui changera (modifiera son affichage) lorsque que le modèle changera. Mais les changements du modèle sont aussi synchronisés avec le serveur. La synchronisation avec le serveur se fait avec la méthode `Backbone.sync()`, à chaque fois qu'un modèle fait une opération « CRUD », `Backbone.sync()` est appelée pour « discuter » avec le serveur (pour le moment cela va fonctionner tout seul, mais nous reviendrons plus tard à `Backbone.sync()` pour comprendre son fonctionnement et même modifier celui-ci).

**Remarque :** l'acronyme CRUD signifie Create, Read, Update, Delete (Créer, Lire, Mettre à jour, Supprimer). Si vous faites le lien avec le chapitre précédent, lorsque nous allons sauvegarder un nouveau modèle, ce sera une création et une requête de type POST sera envoyée au serveur, dans le cas de la lecture ce sera une requête de type GET, PUT pour les mises à jour de modèles et enfin DELETE pour la suppression. Et c'est la méthode `Backbone.sync()` qui va se charger de faire la bonne requête au serveur en fonction de l'action du modèle.

La collection Backbone (`Backbone.Collection`) sert à stocker (en mémoire) un ensemble de modèles de même type. Elle permettra de les trier par exemple, les filtrer, etc. ... Elle aussi est généralement liée à une vue et permet de « récupérer » un ensemble de modèles en provenance du serveur. Là aussi, c'est `Backbone.sync()` qui s'occupe de faire le travail.

## 7.2 Modèles

Si vous avez bien suivi les chapitres précédents vous devez disposer d'un squelette d'application et « client » et « serveur ». Nous allons donc, dans un premier temps définir notre modèle en javascript, puis nous le manipulerons directement dans la console du navigateur. Vous pouvez d'ores et déjà lancer la partie serveur avec la commande `node app.js` (ou `nodemon app.js`).

### 7.2.1 Définition du modèle

Ouvrez la page `index.html` du répertoire `public` et à l'intérieur de la fonction de chargement (`$(function(){})`) vous allez saisir le code de votre premier modèle :

```
$(function (){

  window.Post = Backbone.Model.extend({
    urlRoot : "/blogposts"
  })
})()
```

```
});  
});
```

Cela fait peu de code mais nous avons déjà une « mécanique utilisable » avec un grand nombre de possibilités. Nous avons donc un modèle `Post` pour lequel nous avons juste précisé l'url (`urlRoot`) à appeler (par `Backbone.sync`) lors d'actions/traitement de type « CRUD ». En effet un modèle « arrive » avec entre autres les méthodes suivantes : `save` (pour créer et sauvegarder), `fetch` (pour lire des données en provenance du serveur) et `destroy` (pour supprimer le modèle du serveur).

**Remarque :** j'ai préfixé `Post` par `window` pour y avoir accès en tant que variable globale (dans ma console par exemple).

Nous n'avons pas défini de « champs » comme on peut le faire en java, nous allons voir qu'il existe diverses manières de le faire.

Lancez votre navigateur et connectez-vous sur <http://localhost:3000> (la page `index.html` est chargée par défaut si on ne le précise pas) et ouvrez votre console (celle du navigateur).

Puis saisissez ceci dans la console (et validez) :

*Nouvelle instance d'un modèle “Post” :*

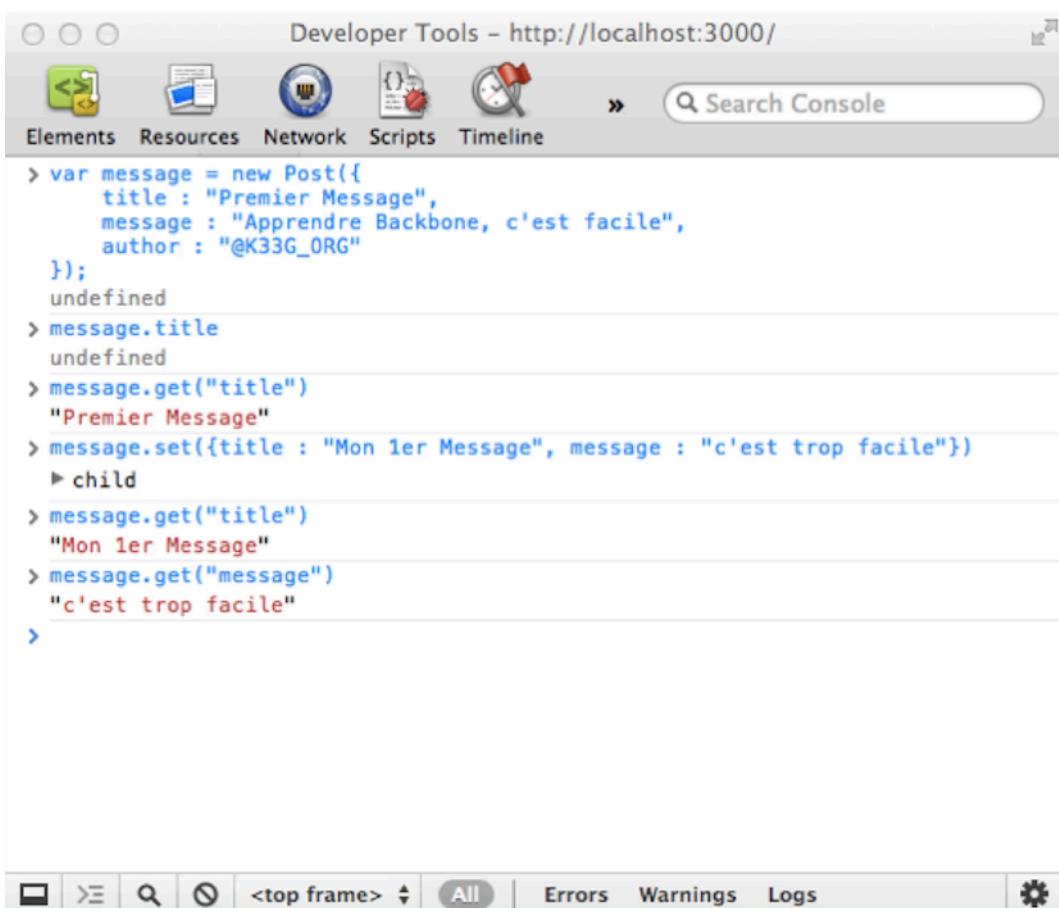
```
var message = new Post({  
    title : "Premier Message",  
    message : "Apprendre Backbone, c'est facile",  
    author : "@K33G_ORG"  
});
```

Nous avons maintenant une instance d'un modèle, jouons avec. Vous avez noté que la définition des champs du modèle (`title`, `message`, `author`) se fait à l'instanciation du modèle, personnellement je trouve ce la pratique et très lisible (un début programmation fonctionnelle), mais je comprends que cela puisse perturber (nous verrons comment faire autrement si vous le souhaitez).

**Remarque (importante) :** la propriété `urlRoot` du modèle n'est utile que si l'on se « sert » d'un modèle hors d'une collection de modèles (le modèle est indépendant), sinon, si le modèle appartient à une collection est qu'il n'a pas de propriété `urlRoot` renseignée, il « utilise » la propriété `url` de la collection à laquelle il appartient.

### 7.2.2 Getters et Setters

Pour lire ou modifier les valeurs des propriétés de notre `message`, le réflexe serait pour par exemple obtenir la valeur du titre de taper la commande `message.title`, et bien cela ne fonctionne pas ! Backbone a une mécanique différente, si vous souhaitez obtenir la valeur du titre il faudra taper la commande `message.get("title")` et pour la modifier `message.set("title", "mon nouveau titre")` ou `message.set({title : "mon nouveau titre"})`, cette dernière notation permet de changer plusieurs propriétés en une seule passe. Cela peut surprendre mais cela a beaucoup d'avantages d'utiliser des méthodes plutôt qu'une simple affectation, on peut ainsi s'abonner aux changements des valeurs pour déclencher automatiquement un traitement (comme le rafraîchissement d'une vue par exemple). Faites donc l'exercice dans la console :



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. At the top, there are five icons: Elements, Resources, Network, Scripts, and Timeline. Below the tabs is a search bar labeled 'Search Console'. The main area contains a series of JavaScript console commands and their results:

```

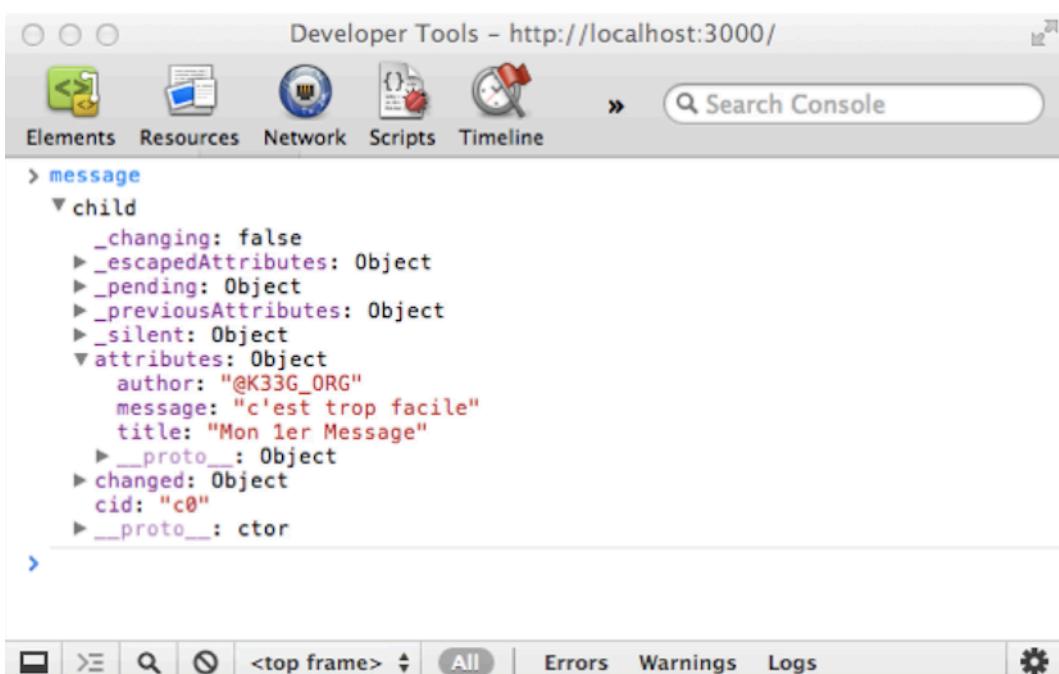
> var message = new Post({
  title : "Premier Message",
  message : "Apprendre Backbone, c'est facile",
  author : "@K33G_ORG"
});
undefined
> message.title
undefined
> message.get("title")
"Premier Message"
> message.set({title : "Mon 1er Message", message : "c'est trop facile"})
▶ child
> message.get("title")
"Mon 1er Message"
> message.get("message")
"c'est trop facile"
>

```

At the bottom of the console window, there is a toolbar with icons for back, forward, search, and refresh, followed by '<top frame>' and a dropdown menu. To the right of the toolbar are buttons for 'All', 'Errors', 'Warnings', and 'Logs', and a gear icon for settings.

### 7.2.3 Structure d'un modèle

Mais allons voir comment est structuré un modèle : dans la console, tapez `message` (notre instance de modèle `Post`) et déroulez la structure « `child` » (child pour instance de modèle) :



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The structure of the `message` object is being expanded. The first command is `> message`, followed by `> child`. The expanded structure of `child` is as follows:

```

> message
  ▼ child
    _changing: false
    ▶ _escapedAttributes: Object
    ▶ _pending: Object
    ▶ _previousAttributes: Object
    ▶ _silent: Object
    ▶ attributes: Object
      author: "@K33G_ORG"
      message: "c'est trop facile"
      title: "Mon 1er Message"
    ▶ __proto__: Object
    ▶ changed: Object
    ▶ cid: "c0"
    ▶ __proto__: ctor
  ▶ 

```

At the bottom of the console window, there is a toolbar with icons for back, forward, search, and refresh, followed by '<top frame>' and a dropdown menu. To the right of the toolbar are buttons for 'All', 'Errors', 'Warnings', and 'Logs', and a gear icon for settings.

Vous notez que les propriétés de notre modèle sont contenues dans un objet `attributes` (c'est pour

cela que nous n'y avons pas accès directement et que l'on utilise `get` et `set`).

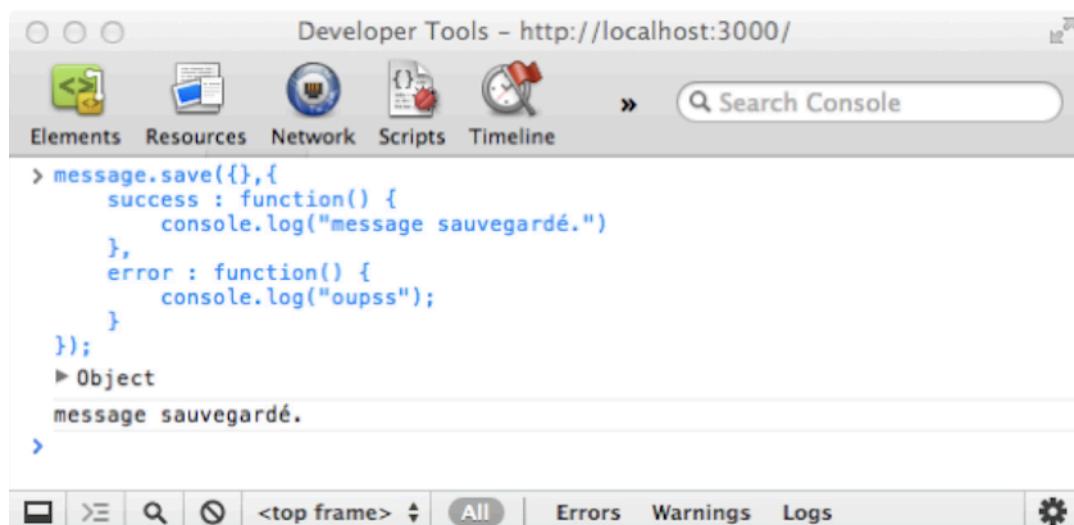
#### 7.2.4 Méthodes “CRUD” du modèle

##### Méthodes `save()` : création & mise à jour

Saisissez le code javascript ci-dessous dans la console du navigateur. Cela va déclencher une requête ajax vers le serveur pour sauvegarder votre modèle (instance de modèle). Si tout se passe bien, c'est la méthode `success()` qui est appelée.

*Appel de la méthode `save()` du modèle :*

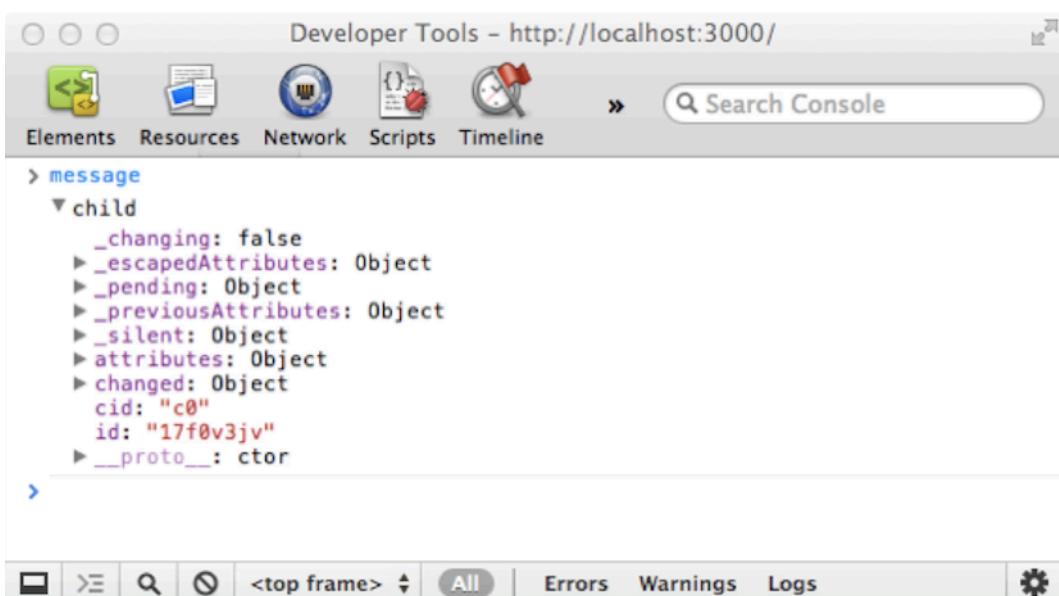
```
message.save({}, {
  success : function() {
    console.log("message sauvégarde.");
  },
  error : function() {
    console.log("oupss");
  }
});
```



Vous pouvez noter au passage que côté serveur, c'est bien une méthode de type POST qui a été faite et qu'un modèle a été créé en base de données.



Si dans la console du navigateur, vous saisissez `message` (notre modèle) et validez, vous aurez l'opportunité de pouvoir dérouler l'ensemble des membres (propriétés et méthodes) de message. Vous noterez l'apparition de la propriété `id`, avec une valeur (valeur unique qui a été affectée par le serveur).



Modifions maintenant le modèle en cours et sauvegardons le :

*Appel de la méthode save() du modèle :*

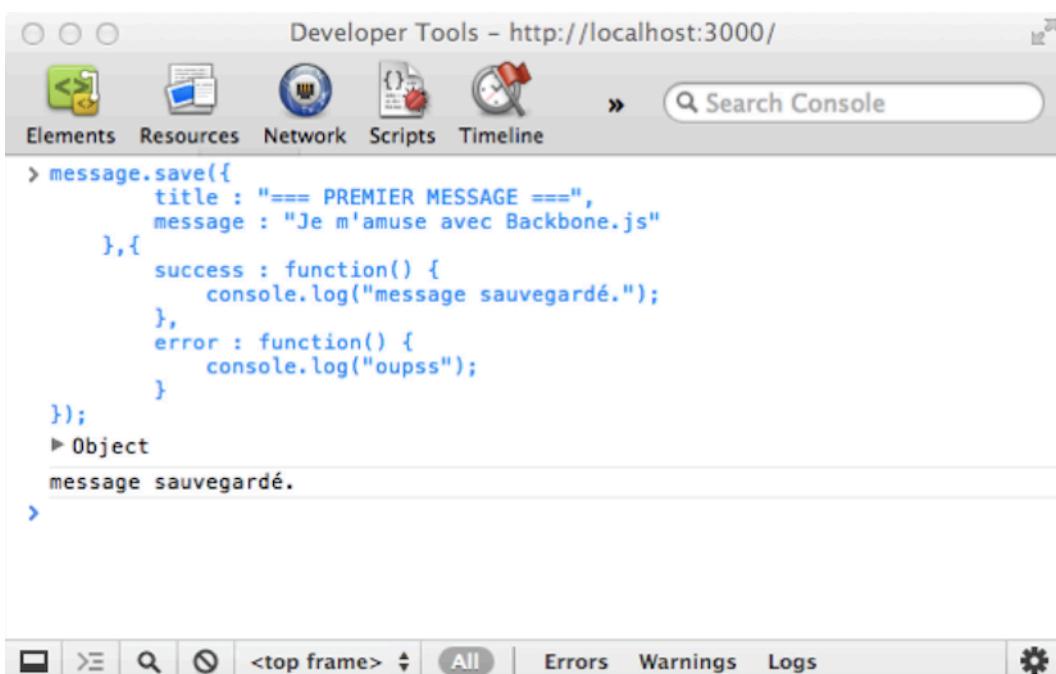
```
message.save({
    title : "==== PREMIER MESSAGE ====",
    message : "Je m'amuse avec Backbone.js"
}, {
    success : function() {
        console.log("message sauvégarde.");
    },
    error : function() {
        console.log("oupss");
    }
});
```

Ce qui est équivalent à ceci (nous avons juste utilisé un raccourci) :

```
message.set({
    title : "==== PREMIER MESSAGE ====",
    message : "Je m'amuse avec Backbone.js"
});

message.save({}, {
    success : function() {
        console.log("message sauvégarde.");
    },
    error : function() {
        console.log("oupss");
    }
});
```

Et vous obtiendrez :

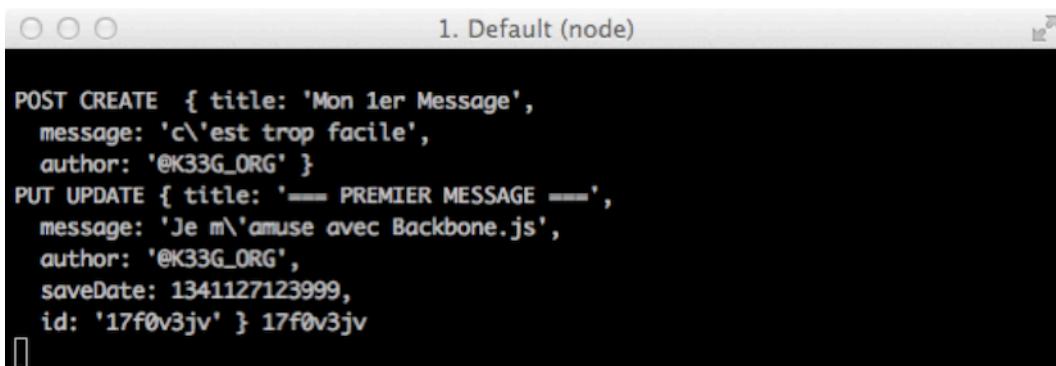


The screenshot shows the Network tab of the Developer Tools. A single request is listed:

```
> message.save({
    title : "==== PREMIER MESSAGE ===",
    message : "Je m'amuse avec Backbone.js"
}, {
    success : function() {
        console.log("message sauvé.");
    },
    error : function() {
        console.log("oups");
    }
});
▶ Object
message sauvé.
```

Below the list, there are several buttons: a refresh icon, a search icon, a magnifying glass icon, <top frame>, All (highlighted), Errors, Warnings, Logs, and a gear icon.

Cette fois ci, côté serveur, la requête ajax a été détectée comme une requête de type PUT, donc une requête de mise à jour du modèle.



The terminal window shows the server logs for a PUT request:

```
1. Default (node)
POST CREATE { title: 'Mon 1er Message',
  message: 'c\'est trop facile',
  author: '@K33G_ORG' }
PUT UPDATE { title: '--- PREMIER MESSAGE ---',
  message: 'Je m\'amuse avec Backbone.js',
  author: '@K33G_ORG',
  saveDate: 1341127123999,
  id: '17f0v3jv' } 17f0v3jv
```

### Méthode fetch() : retrouver un modèle

Notez bien le numéro d'id affecté par le serveur (dans notre exemple : “17fov3jv”, attention c'est une valeur aléatoire unique affectée par le serveur) et rafraîchissez votre page dans le navigateur (le modèle message disparaît donc de la mémoire).

Créons maintenant une nouvelle fois un modèle avec simplement comme champ, un id prenant la valeur de la clé du modèle sauvegardé en base :

```
var message = new Post({id:"17f0v3jv"});
```

Puis appelons à nouveau la méthode `fetch()` du modèle :

*Appel de la méthode `fetch()` pour charger les données du serveur :*

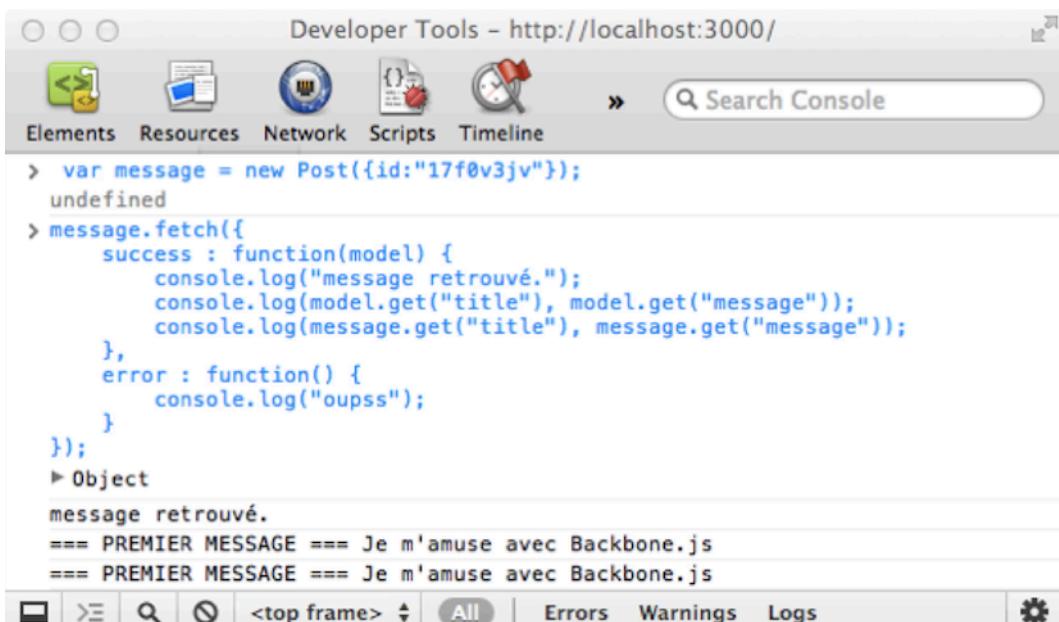
```
message.fetch({
  success : function(model) {
    console.log("message retrouvé.");
  }
});
```

```

        console.log(model.get("title"), model.get("message"));
        console.log(message.get("title"), message.get("message"));
    },
    error : function() {
        console.log("oupss");
    }
});

```

Nous récupérons bien les données du modèle sauvegardé :



Et côté serveur, on peut vérifier le message affiché dans le terminal : nous avons bien une requête de type GET avec en paramètre la clé unique (identifiant) du modèle.

### Méthode `destroy()` : supprimer un modèle du serveur

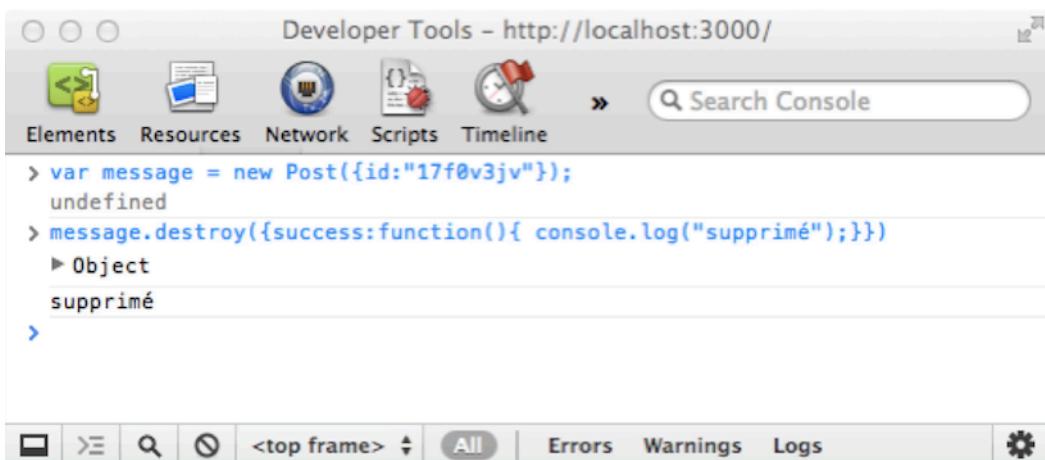
Le principe est le même que pour la méthode `fetch()`, à partir d'un modèle en cours d'utilisation ou un nouveau modèle créé avec un id existant, il suffit ensuite d'appeler la méthode `destroy()` du modèle pour le supprimer de la base de données :

*Appel de la méthode `destroy()` :*

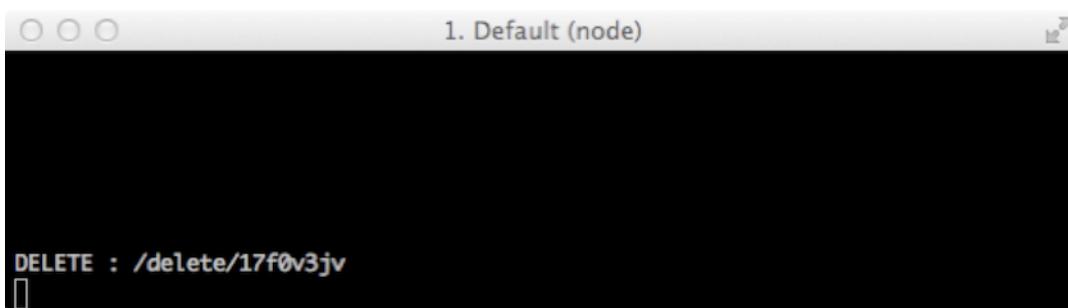
```

var message = new Post({id:"17f0v3jv"});
message.destroy({success:function(){ console.log("supprimé");}})

```



Et côté serveur, nous avons bien une requête de type `DELETE` avec en paramètre la clé unique (identifiant) du modèle :



### 7.2.5 Evénements

Il est possible de “s’abonner” aux changements effectués sur un modèle grâce à la méthode `on()` (anciennement `bind()`) de l’instance du modèle :

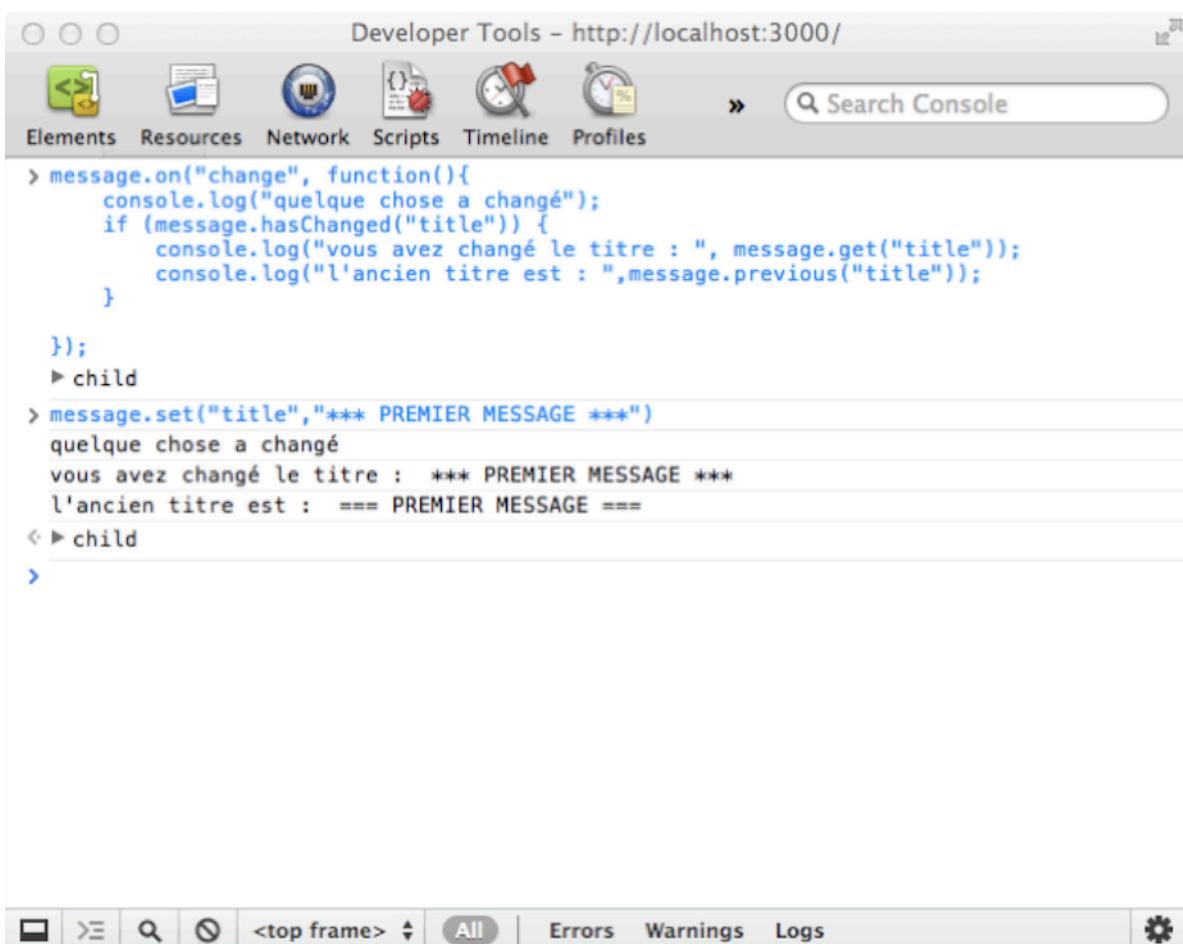
*Abonnement à un événement avec la méthode `on()` :*

```

message.on("change", function(){
  console.log("quelque chose a changé");
  if (message.hasChanged("title")) {
    console.log("vous avez changé le titre : ", message.get("title"));
    console.log("l'ancien titre est : ", message.previous("title"));
  }
});


```

Nous venons de nous abonner aux changements de valeurs des champs de l’instance de modèle `message`. C’est à dire que nous serons notifiés dès qu’une valeur d’un champ de `message` est modifiée :



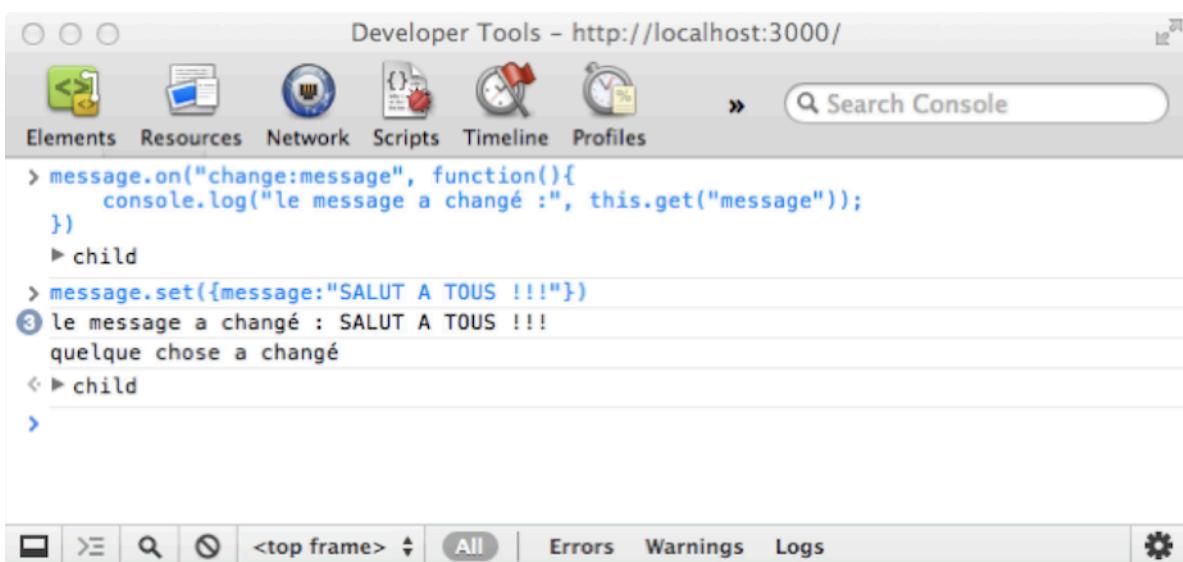
The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. At the top, there are tabs for 'Elements', 'Resources', 'Network', 'Scripts', 'Timeline', and 'Profiles'. A search bar labeled 'Search Console' is located at the top right. Below the tabs, the console output is displayed in blue text:

```
> message.on("change", function(){
  console.log("quelque chose a changé");
  if (message.hasChanged("title")) {
    console.log("vous avez changé le titre : ", message.get("title"));
    console.log("l'ancien titre est : ", message.previous("title"));
  }
});
▶ child
> message.set("title","*** PREMIER MESSAGE ***")
quelque chose a changé
vous avez changé le titre : *** PREMIER MESSAGE ***
l'ancien titre est : === PREMIER MESSAGE ===
< ▶ child
>
```

At the bottom of the console window, there are several icons: a refresh button, a search icon, a clear icon, a dropdown menu for 'top frame', and buttons for 'All', 'Errors', 'Warnings', and 'Logs'.

Vous pouvez aussi “écouter” les changements spécifiques à un attribut bien particulier de cette façon :

```
message.on("change:message", function(){
  console.log("le message a changé :", this.get("message"));
})
```



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. At the top, there are tabs for 'Elements', 'Resources', 'Network', 'Scripts', 'Timeline', and 'Profiles'. A search bar labeled 'Search Console' is located at the top right. Below the tabs, the console output is displayed in blue text:

```
> message.on("change:message", function(){
  console.log("le message a changé :", this.get("message"));
})
▶ child
> message.set({message:"SALUT A TOUS !!!"})
③ le message a changé : SALUT A TOUS !!!
quelque chose a changé
< ▶ child
>
```

At the bottom of the console window, there are several icons: a refresh button, a search icon, a clear icon, a dropdown menu for 'top frame', and buttons for 'All', 'Errors', 'Warnings', and 'Logs'.

On s’aperçoit que les abonnements se cumulent. Ce que nous venons de faire n’est pas contre valable que pour une instance de modèle de type Post. Comment faire pour que cela soit valable pour tous les

Posts ? En utilisant le constructeur du modèle, ou plus spécifiquement la méthode `initialize()` qui est appelée par le constructeur du modèle.

```
//TODO: parler de off()
```

### 7.2.6 Constructeur : initialize

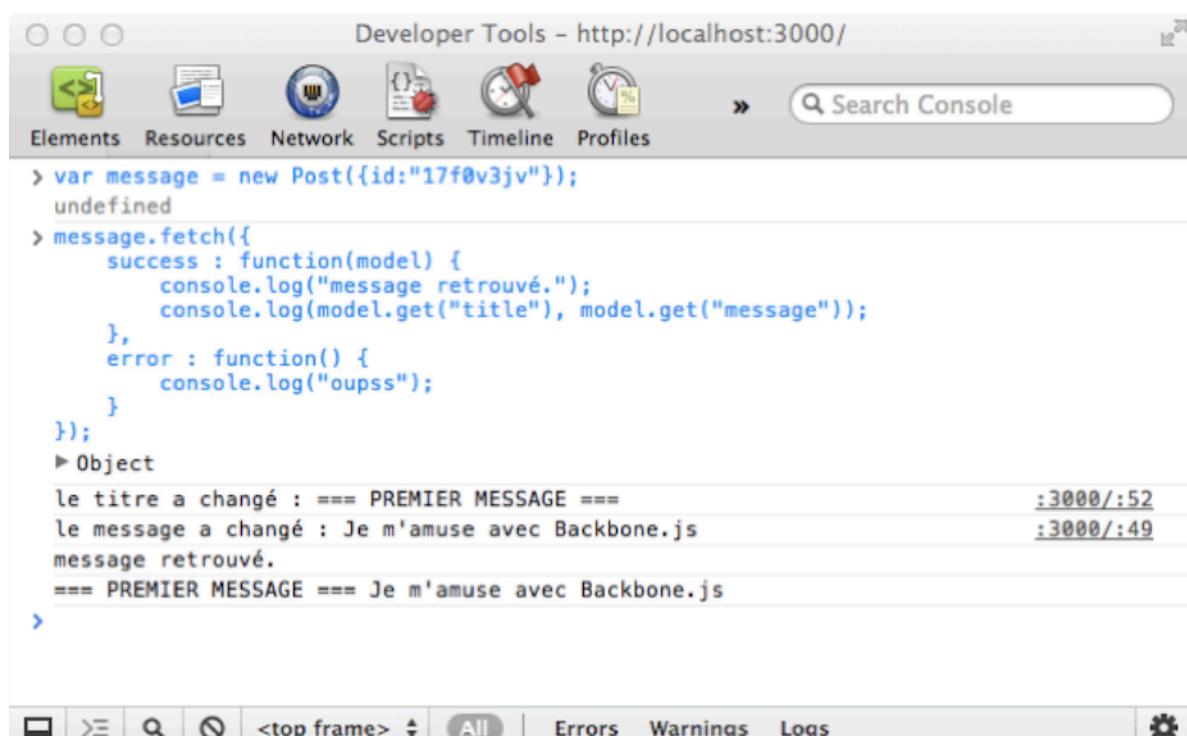
Retournez modifier le code de notre modèle dans la page `index.html` du répertoire `public` :

*Méthode `initialize()` et utilisation de `on()` :*

```
window.Post = Backbone.Model.extend({
  urlRoot :"/blogposts",

  initialize : function () {
    this.on("change:message", function(){
      console.log("le message a changé :", this.get("message"));
    });
    this.on("change:title", function(){
      console.log("le titre a changé :", this.get("title"));
    });
  }
});
```

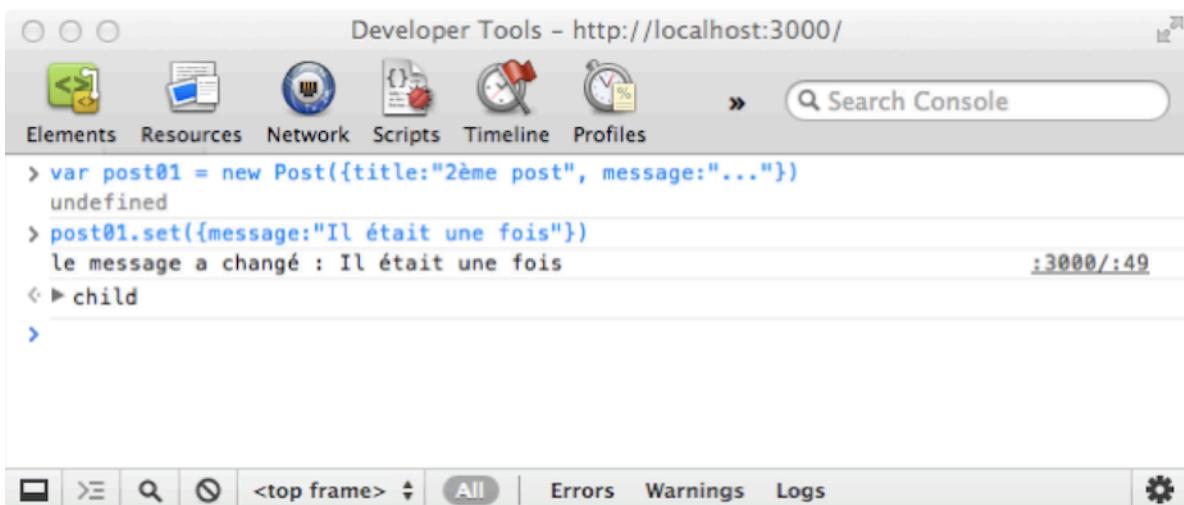
Nous venons d'expliquer que pour chacun des modèles de type `Post`, si son titre ou son message change, alors nous sommes notifiés. Rechargez votre page, et rappelez par un `fetch` votre modèle (les données du serveur) :



The screenshot shows the Developer Tools console in Google Chrome. The URL is `http://localhost:3000/`. The console output is as follows:

```
> var message = new Post({id:"17f0v3jv"});
undefined
> message.fetch({
  success : function(model) {
    console.log("message retrouvé.");
    console.log(model.get("title"), model.get("message"));
  },
  error : function() {
    console.log("oupss");
  }
});
▶ Object
le titre a changé : === PREMIER MESSAGE === :3000/:52
le message a changé : Je m'amuse avec Backbone.js :3000/:49
message retrouvé.
== PREMIER MESSAGE == Je m'amuse avec Backbone.js
```

Mais vous pouvez essayer avec un nouveau Post :



### 7.2.7 “Augmenter” le modèle : ajouter des valeurs par défaut et des méthodes au modèle

Vous pouvez éprouver le besoin de coder de manière plus classique (à la java), nous allons en profiter pour ajouter des getters et des setters “à l’ancienne” ainsi que des valeurs par défaut au modèle :

*Ajouts de propriétés et de méthodes au model :*

```
window.Post = Backbone.Model.extend({
  urlRoot :"/blogposts",

  /* valeurs par défaut du modèle */
  defaults : {
    title : "???", 
    message : "...",
    author : "John Doe"
  },

  initialize : function () {
    this.on("change:message", function(){
      console.log("le message a changé :", this.get("message"));
    });
    this.on("change:title", function(){
      console.log("le titre a changé :", this.get("title"));
    });
  },

  /* les getters et les setters à l'ancienne */
  getTitle : function () {
    return this.get("title");
  },
  setTitle : function (value) {
    this.set("title", value);
  },
  getMessage : function () {
    return this.get("message");
  }
});
```

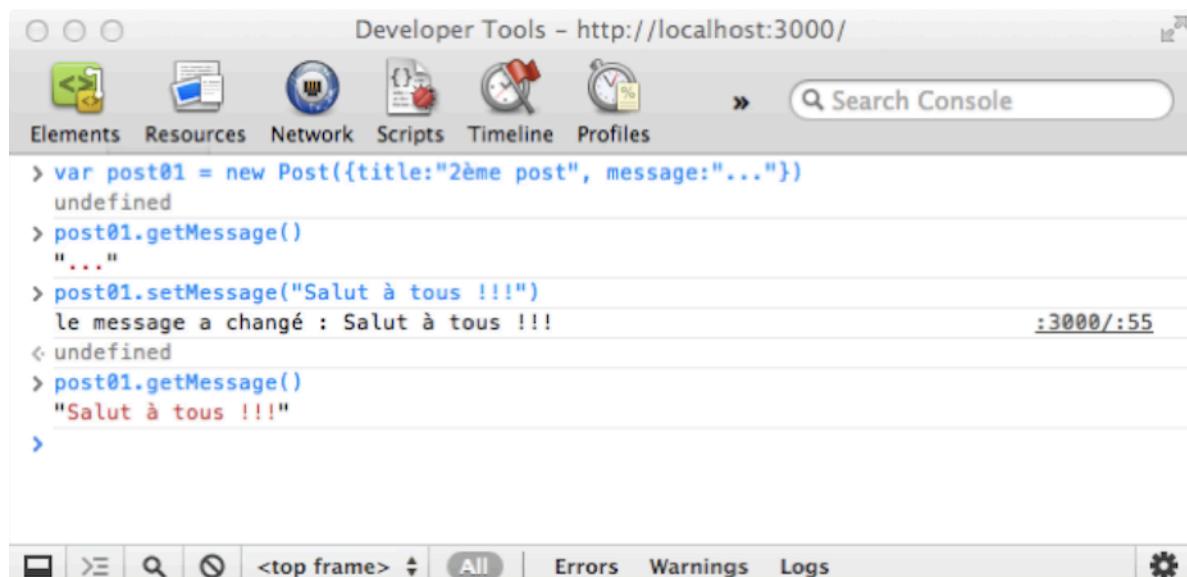
```

},
setMessage : function (value) {
    this.set("message", value);
},
getAuthor : function () {
    return this.get("author");
},
setAuthor : function (value) {
    this.set("author", value);
}
};

});

```

Que nous pouvons utiliser de la manière suivante :



The screenshot shows a browser developer tools console window titled "Developer Tools - http://localhost:3000/". The tabs at the top are Elements, Resources, Network, Scripts, Timeline, and Profiles. Below the tabs is a search bar labeled "Search Console". The main area contains the following JavaScript code:

```

> var post01 = new Post({title:"2ème post", message:""})
undefined
> post01.getMessage()
"..."
> post01.setMessage("Salut à tous !!!")
le message a changé : Salut à tous !!!
< undefined
> post01.getMessage()
"Salut à tous !!!"
>

```

At the bottom of the console, there are several icons: a refresh button, a search icon, a stop icon, a dropdown menu, and buttons for "All", "Errors", "Warnings", and "Logs".

###Validation

//TODO: à faire

### 7.2.8 Comment détecter qu'un modèle a été changé par quelqu'un d'autre ?

Dans la “rubrique trucs & astuces”, il est possible de détecter un changement effectué côté serveur (cet exemple est à titre démonstratif et mérite d'être optimisé ou d'utiliser d'autres moyens tels les websockets par exemple). Tapez ceci dans la console de votre navigateur :

*Appel de la méthode fetch() à intervalles réguliers :*

```

var message = new Post({id:"17f0v3jv"});

setInterval(function(){
    message.fetch();
},1000)

```

Nous venons de créer un nouveau Post, en lui renseignant son id (car nous savons qu'il existe côté serveur) et demandons à javascript d'aller chercher les données toutes les 1000 millisecondes (donc toutes les 1 secondes).

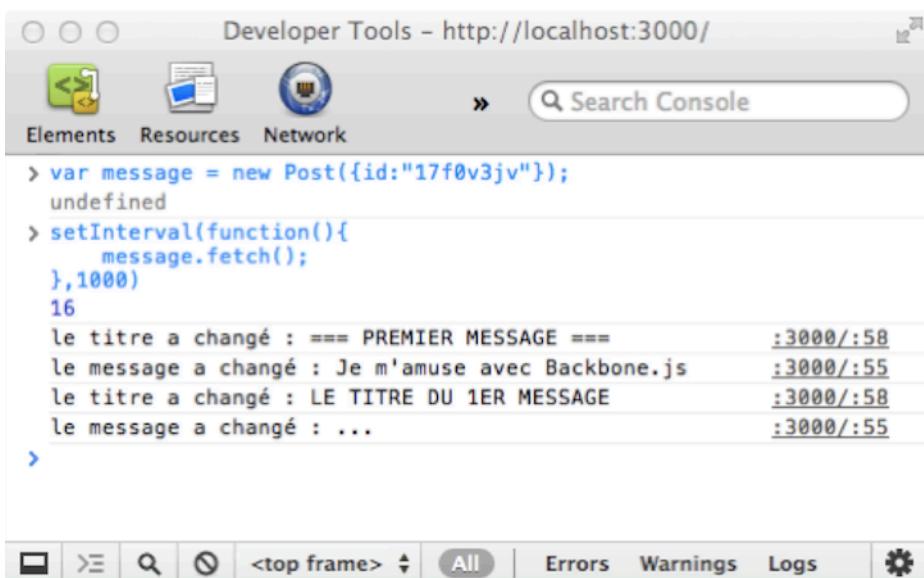
Ensuite, ouvrez un autre navigateur (par exemple FireFox) et connectez-vous sur <http://localhost:3000>, puis dans la console de ce navigateur tapez ceci :

```
var message = new Post({id:"17f0v3jv"});
message.set({title:"LE TITRE DU 1ER MESSAGE"});
```

puis sauvegardez :

```
message.save()
```

Et dans la console du précédent navigateur, vous allez voir s'afficher un message expliquant que le modèle a changé. La méthode `change()` que nous avons définie dans `initialize` est aussi appelée/déclenchée lorsque que Backbone détecte un changement (lors d'un `fetch()`) entre les données clientes et les données serveurs :



The screenshot shows the Developer Tools console in Google Chrome. The title bar says "Developer Tools - http://localhost:3000/". Below it are three tabs: "Elements", "Resources", and "Network". A search bar labeled "Search Console" is to the right. The main area contains the following code and its execution results:

```
> var message = new Post({id:"17f0v3jv"});
undefined
> setInterval(function(){
  message.fetch();
},1000)
16
le titre a changé : === PREMIER MESSAGE === :3000/:58
le message a changé : Je m'amuse avec Backbone.js :3000/:55
le titre a changé : LE TITRE DU 1ER MESSAGE :3000/:58
le message a changé : ... :3000/:55
>
```

At the bottom of the console are several icons: a square, a magnifying glass, a circle, a dropdown arrow, and a gear. To the right of these are buttons for "All", "Errors", "Warnings", and "Logs".

Pour le moment nous avons fait le tour de l'essentiel du fonctionnement des modèles (je vous engage cependant à lire la documentation de Backbone (<http://backbonejs.org/#Model>), pour en découvrir toutes les possibilités). Passons donc aux collections.

### 7.3 Collections

Changeons à nouveau le code de notre modèle dans notre page `index.html` :

```
window.Post = Backbone.Model.extend({
  urlRoot :"/blogposts"
});
```

Puis définissons une collection : `Backbone.Collection`

```
window.Posts = Backbone.Collection.extend({
  url :"/blogposts",
  model : Post
});
```

Nous avons précisé une url pour les collections (c'est ce qui sera utilisé lorsque nous ferons des requêtes au serveur), et le type de modèle de la collection. Et rechargeons notre page, pour une nouvelle fois passer en mode commande (et donc ouvrir la console du navigateur).

### 7.3.1 Comment ajouter des modèles à une collection

Tout d'abord, il faut créer(instancier) une nouvelle collection :

```
var postsList = new Posts();
```

Puis créer(instancier) 3 nouveaux modèles Post :

```
var post1 = new Post({title: "Titre Post1", message: "message 1"})
var post2 = new Post({title: "Titre Post2", message: "message 2"})
var post3 = new Post({title: "Titre Post3", message: "message 3"})
```

Que vous pouvez ajouter à la collection de cette manière :

```
postsList.add(post1)
```

Ou de cette façon :

```
postsList.add([post2, post3])
```

Il est possible de créer directement un modèle dans la collection de cette façon :

```
postsList.add(new Post({title: "Titre Post4", message: "message 4"}))
```

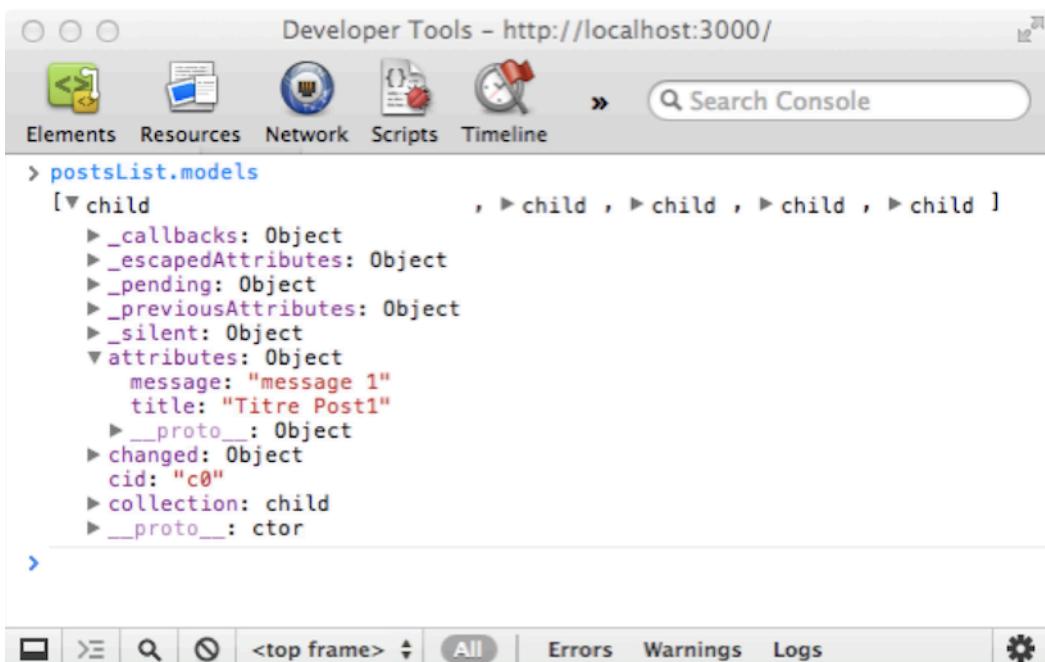
ou de cette manière :

```
postsList.create({title: "Titre Post5", message: "message 5"})
```

Et enfin, saisissez ceci :

```
postsList.models
```

Et vous obtenez un tableau des 5 modèles de la collection :



Et vous pouvez accéder aux modèles et à leurs propriétés par exemple de cette façon-ci :

```
postsList.models[0].get("title")
```

Ou bien de cette manière :

```
postsList.at(0).get("title")
```

`at()` utilise le numéro d'index du modèle dans la collection.

Ou bien comme ceci :

```
postsList.get("3chk5h1").get("title")
```

`get()` utilise le numéro d'id affecté (par le serveur) au modèle lorsqu'il est sauvegardé.

Ou encore :

```
postsList.getByCid("c18").get("title")
```

`getByCid()` utilise la propriété `cid` du modèle automatiquement affectée lorsqu'il est créé/instancié (`new`), cela peut être pratique pour les modèles qui ne sont pas encore sauvegardés et qui donc n'ont pas encore d'id.

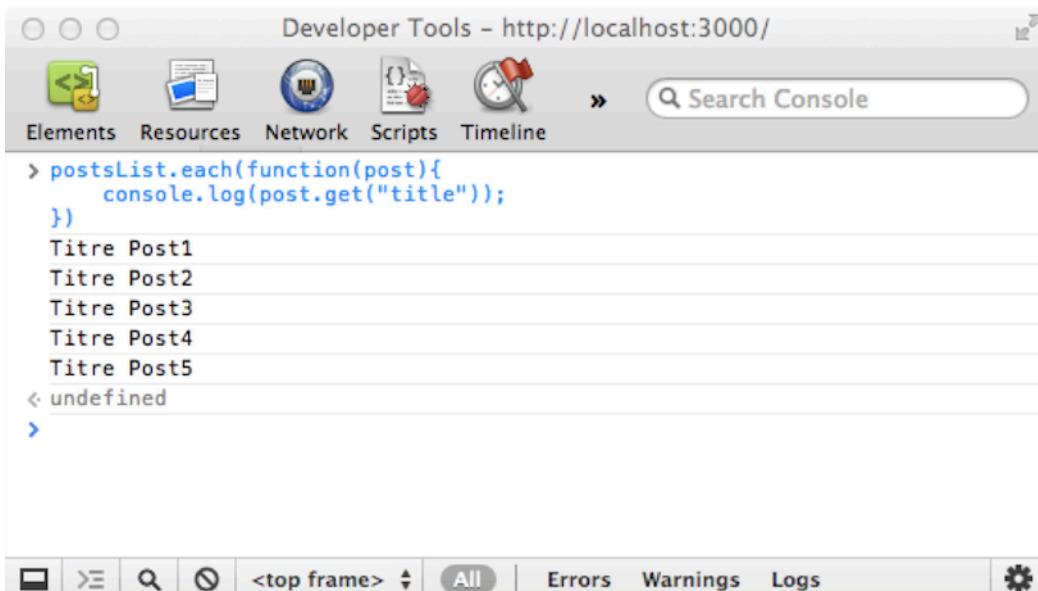
### 7.3.2 Parcourir les modèles d'une collection

Les collections Backbone dispose d'une méthode `each()` (issue de la librairie Underscore.js) qui permet de parcourir chacun des éléments de la collection et de faire un traitement pour chacun de ces éléments. Par exemple, nous souhaitons afficher le titre de chacun des Posts de la collection, pour cela saisissez le code suivant dans la console :

*Appel de la méthode each() :*

```
postsList.each(function(post){
    console.log(post.get("title"));
})
```

Vous obtiendrez le résultat suivant :



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The title bar says 'Developer Tools - http://localhost:3000/'. Below it are tabs for 'Elements', 'Resources', 'Network', 'Scripts', and 'Timeline'. The 'Console' tab has a search bar labeled 'Search Console'. The main area contains the following code and output:

```
> postsList.each(function(post){
    console.log(post.get("title"));
})
Titre Post1
Titre Post2
Titre Post3
Titre Post4
Titre Post5
< undefined
>
```

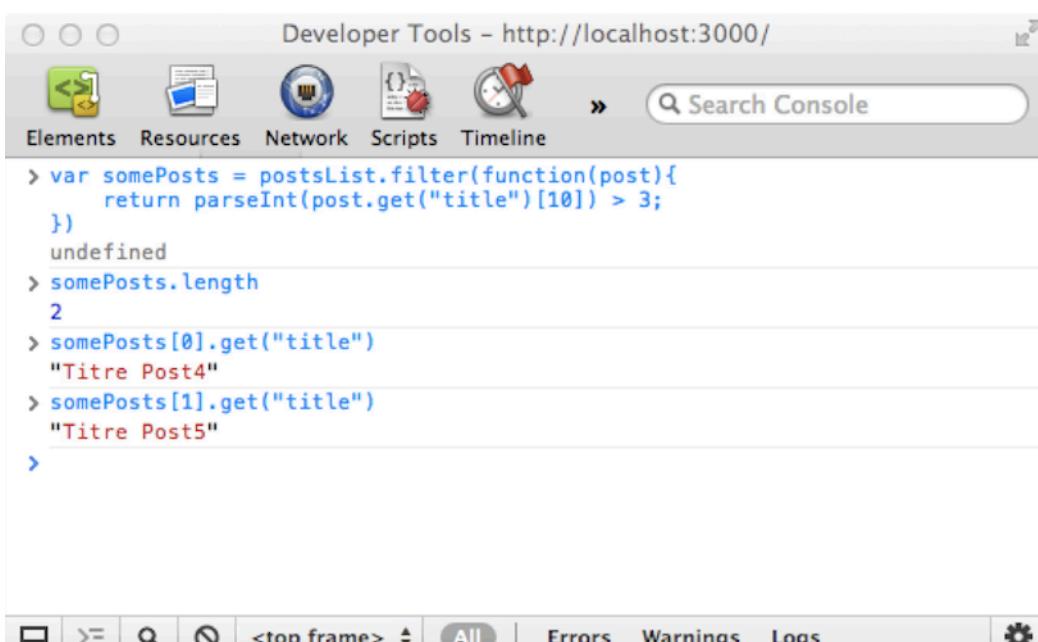
At the bottom of the console window, there are several icons: a magnifying glass, a gear, and buttons for 'All', 'Errors', 'Warnings', and 'Logs'.

### 7.3.3 Filtrer les modèles d'une collection

De la même façon que `each()` il existe une méthode `filter()` (toujours issue de Underscore.js) qui permet de retourner l'ensemble des modèles d'une collection répondant à un critère. Dans l'exemple ci-dessous je souhaite obtenir la liste des modèles dont le titre se termine par un chiffre supérieur à 3 :

*Appel de la méthode `filter()` :*

```
var somePosts = postsList.filter(function(post){
    return parseInt(post.get("title")[10]) > 3;
})
```



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The title bar says 'Developer Tools - http://localhost:3000/'. Below it are tabs for 'Elements', 'Resources', 'Network', 'Scripts', and 'Timeline'. The 'Console' tab has a search bar labeled 'Search Console'. The main area contains the following code and output:

```
> var somePosts = postsList.filter(function(post){
    return parseInt(post.get("title")[10]) > 3;
})
undefined
> somePosts.length
2
> somePosts[0].get("title")
"Titre Post4"
> somePosts[1].get("title")
"Titre Post5"
>
```

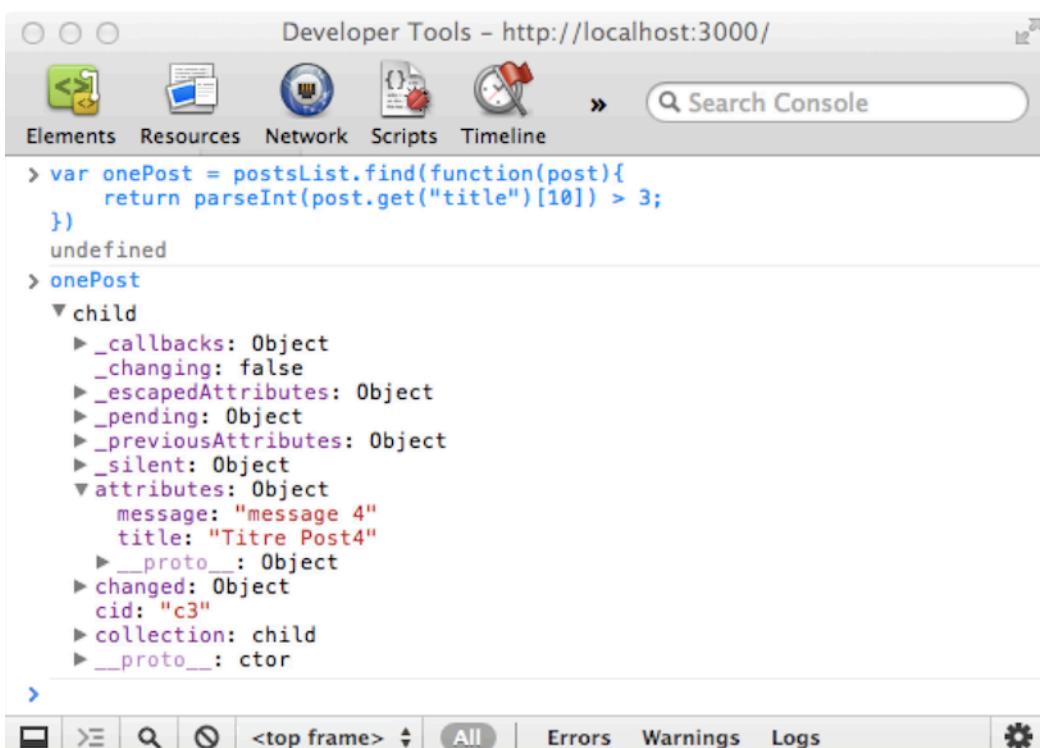
At the bottom of the console window, there are several icons: a magnifying glass, a gear, and buttons for 'All', 'Errors', 'Warnings', and 'Logs'.

### 7.3.4 Trouver le 1er modèle d'une collection correspondant à un critère

La méthode `find()` fonctionne comme `filter()` mais retourne le premier élément correspondant aux critères de recherche :

*Appel de la méthode `find()` :*

```
var onePost = postsList.find(function(post){
    return parseInt(post.get("title")[10]) > 3;
})
```



### 7.3.5 Autres méthodes de la collection

```
//TODO: à faire ... en attendant lisez la doc de Backbone (ou faites moi une PR ;)
```

## 7.4 Les collections “parlent” au serveur

Sauvegardons d'abord nos modèles (en utilisant `each()` pour aller plus vite) pour avoir un jeu d'essai de données en base côté serveur. Pour cela, tapez donc ceci dans votre console :

*Sauvegarder tous les modèles :*

```
postsList.each(function(post){
    post.save({}, {
        success : function (post) {
```

```

        console.log(post.get("title")," sauvegardé");
    },
    error : function () { console.log("Oupss"); }
});

})

```

```

Developer Tools - http://localhost:3000/
Elements Resources Network Scripts Timeline » Search Console
> postsList.each(function(post){
  post.save({},{
    success : function (post) {
      console.log(post.get("title")," sauvegardé");
    },
    error : function () { console.log("Oupss"); }
  });
})

undefined
Titre Post5 sauvegardé
Titre Post4 sauvegardé
Titre Post3 sauvegardé
Titre Post2 sauvegardé
Titre Post1 sauvegardé
>

```

Toolbar icons: Full screen, Zoom in, Zoom out, Refresh, <top frame>, All (highlighted), Errors, Warnings, Logs, Settings.

#### 7.4.1 Charger les données

Maintenant nous souhaitons interroger le serveur pour qu'il nous fournisse l'ensemble des modèles de type Post. Rechargez la page pour être sûr de remettre toutes les variables en mémoire à zéro. Ensuite créez une nouvelle instance de collection de Post :

```
var postsList = new Posts();
```

Et chargez celle-ci à partir des données du serveur :

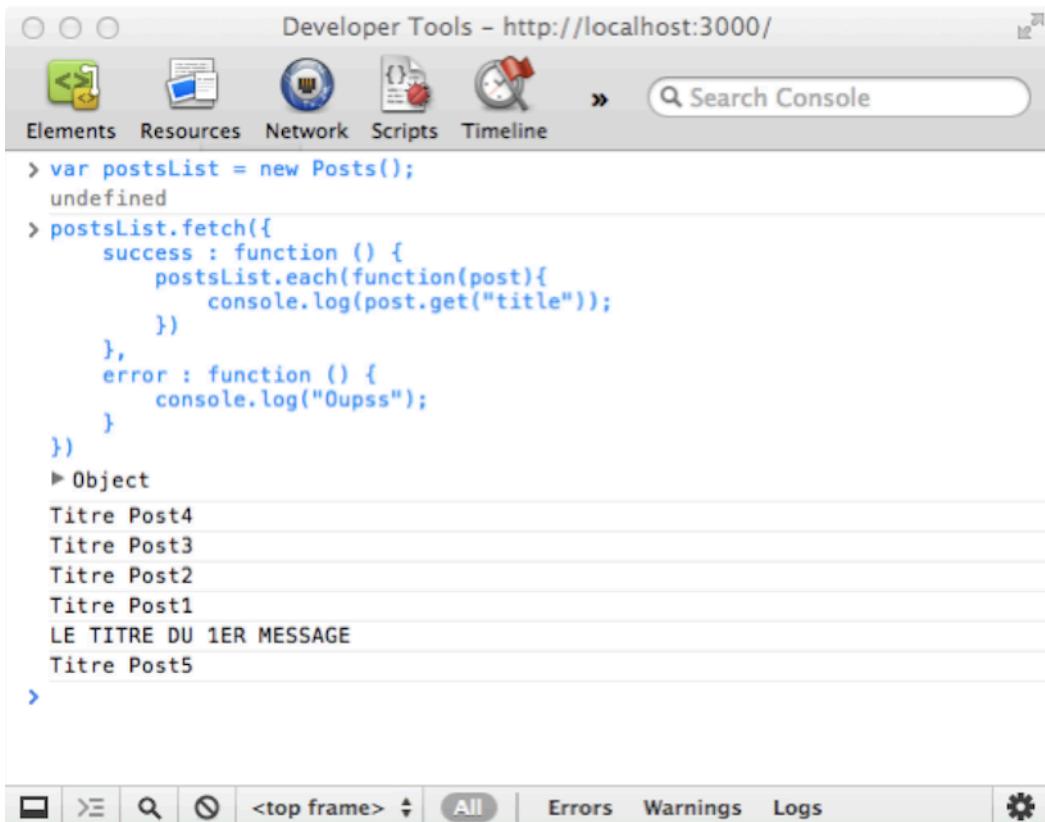
*Appel de la méthode fetch()* :

```

postsList.fetch({
  success : function () {
    postsList.each(function(post){
      console.log(post.get("title"));
    })
  },
  error : function () {
    console.log("Oupss");
  }
})

```

Nous avons donc bien une collection avec les modèles initialisés à partir des données du serveur :



```

Developer Tools - http://localhost:3000/
Elements Resources Network Scripts Timeline » Search Console

> var postsList = new Posts();
undefined
> postsList.fetch({
    success : function () {
        postsList.each(function(post){
            console.log(post.get("title"));
        })
    },
    error : function () {
        console.log("Oupss");
    }
})
▶ Object
Titre Post4
Titre Post3
Titre Post2
Titre Post1
LE TITRE DU 1ER MESSAGE
Titre Post5
>

```

The screenshot shows the Google Chrome Developer Tools console. At the top, there are tabs for Elements, Resources, Network, Scripts, and Timeline. Below the tabs, a search bar says "Search Console". The main area contains a block of JavaScript code. When run, it creates a new instance of the `Posts` collection, calls its `fetch` method, and logs each post's title to the console. The output shows five titles: "Post4", "Post3", "Post2", "Post1", and "LE TITRE DU 1ER MESSAGE". Below the code, there is a toolbar with icons for back, forward, search, and refresh, followed by buttons for "All", "Errors", "Warnings", and "Logs", and a gear icon for settings.

#### 7.4.2 Requêtes

Lorsque nous avons créé notre application côté serveur, nous avions prévu de pouvoir requêter les données. Nous voudrions pouvoir faire ça à partir de la collection que nous avons créée. Pour cela il faudra pouvoir changer la propriété `url` de la collection. Modifions alors le code source de notre collection de la façon suivante (dans la page `index.html`) :

*Ajouter des méthodes à la collection :*

```

window.Posts = Backbone.Collection.extend({
    model : Post,
    all : function () {
        this.url = "/blogposts";
        return this;
    },
    query : function (query) {
        this.url = "/blogposts/query/" + query;
        return this;
    }
});

```

Rechargez ensuite la page, puis créez une nouvelle collection (dans la console du navigateur) :

```
var postsList = new Posts();
```

puis faites une requête :

*Ne charger que certains titres dans la collection :*

```
postsList.query('{"title" : "Titre Post2"}')
  .fetch({
    success:function(result){
      console.log(result);
    }
})
```

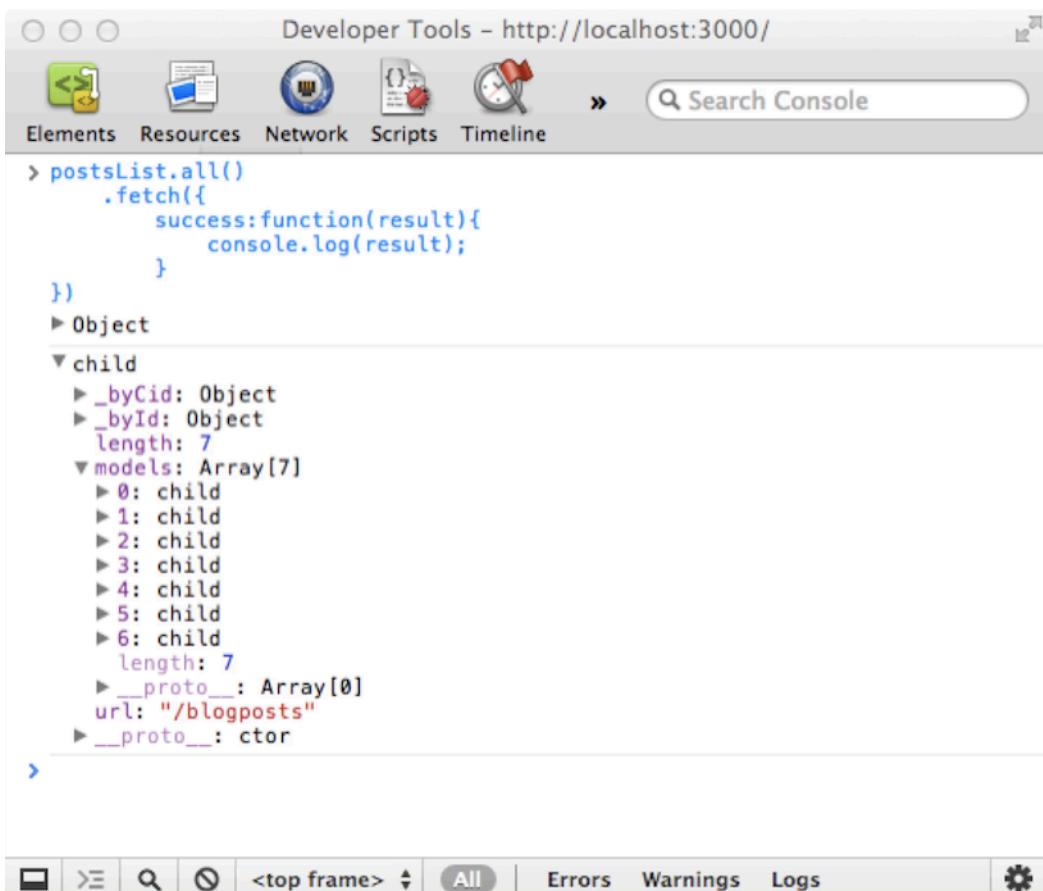
Vous obtenez ceci :

```
Developer Tools - http://localhost:3000/
Elements Resources Network Scripts Timeline » Search Console
> postsList.query('{"title" : "Titre Post2"}')
  .fetch({
    success:function(result){
      console.log(result);
    }
})
▶ Object
▶ child
> postsList.at(0).get("title")
"Titre Post2"
>
```

Ensuite si vous souhaitez charger tous les modèles (toujours dans la console de votre navigateur) :

*Charger tous les modèles dans la collection :*

```
postsList.all()
  .fetch({
    success:function(result){
      console.log(result);
    }
})
```



**A Noter :** il se trouve que les collections dans Backbone ont une méthode `url()` qui est appelée si la propriété `url` n'existe pas, cela peut être un autre moyen d'adresser la problématique de changement d'url. De même si les modèles ajoutés à une collection, n'ont pas de propriété `url` (ni `urlRoot`), ils héritent de celle de la collection (mais cela demande à revoir la politique de « routage » utilisée côté serveur ou de modifier `Backbone.sync()` qui construira les requêtes http en fonction du type d'objet faisant une requête (modèle ou collection)). Pour plus d'information sur le sujet, aller voir : <http://backbonejs.org/#Model-url> ainsi que <http://backbonejs.org/#Collection-url>.

## 7.5 Evénements

Comme pour les modèles, il est possible de s'abonner à des événements issus des collections. Nous souhaitons être notifié de tout changement dans un modèle de la collection. Saisissez donc ceci (dans la console) :

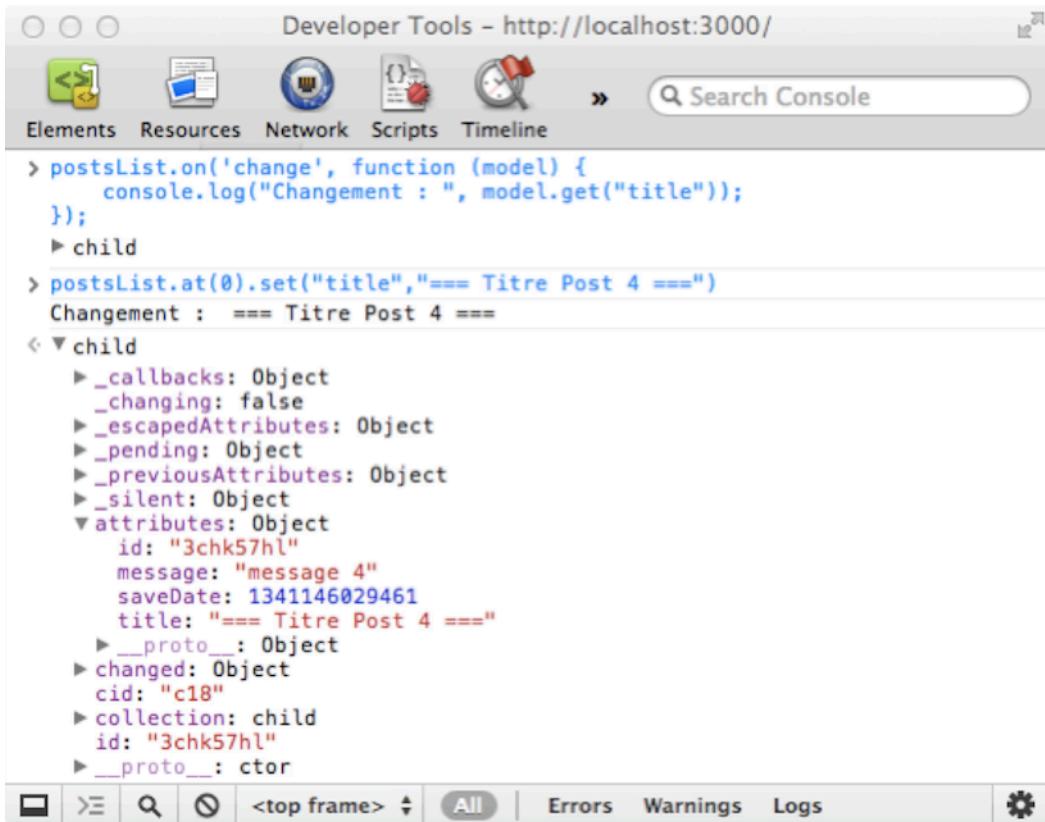
*Affecter un événement à tous les modèles :*

```
'javascript postsList.on('change', function (model) { console.log("Changement : ", model.get("title")); });
```

Puis :

```
postsList.at(0).set("title", "== Titre Post 4 ==")
```

Vous obtenez ceci :



The screenshot shows the Developer Tools Network tab for the URL `http://localhost:3000/`. A search bar at the top right contains the text "Search Console". Below the tabs, a code snippet is displayed:

```
> postsList.on('change', function (model) {
  console.log("Changement : ", model.get("title"));
});
▶ child
> postsList.at(0).set("title","== Titre Post 4 ===")
Changement : === Titre Post 4 ===
< ▾ child
  ▶ _callbacks: Object
  ▶ _changing: false
  ▶ _escapedAttributes: Object
  ▶ _pending: Object
  ▶ _previousAttributes: Object
  ▶ _silent: Object
  ▾ attributes: Object
    id: "3chk57hl"
    message: "message 4"
    saveDate: 1341146029461
    title: "== Titre Post 4 ==="
    ▶ __proto__: Object
  ▶ changed: Object
  ▶ cid: "c18"
  ▶ collection: child
  ▶ id: "3chk57hl"
  ▶ __proto__: ctor
```

Below the code, there is a toolbar with icons for refresh, search, and other developer tools functions. To the right of the toolbar, there are buttons for "All", "Errors", "Warnings", and "Logs", with "All" being the active tab.

**Il n'est donc plus obligatoire d'affecter un événement de façon individuelle aux modèles, tous les modèles appartenant à une collection héritent de l'événement (trigger) associé à celle-ci.**

Voilà nous en connaissons assez sur les modèles et les collections pour passer à la suite.

## 8 Vues & Templating

### Sommaire

- *1ère vue*
- *Mise à jour automatique de l'affichage*
- *Sous-vues*
- *Templating*
- *événements*

*Nous avons joué avec les données dans le chapitre précédent, nous allons maintenant voir comment les afficher dynamiquement dans notre page web.*

Le composant View de Backbone est peut-être celui qui génère le plus de polémiques. Est-ce vraiment une vue ? Ne serait-ce pas plutôt un contrôleur ? Il se trouve que dans une version plus ancienne de Backbone, le composant Controller existait, aujourd'hui il est le devenu le composant Router que nous verrons par la suite ... Cependant, un routeur est-il réellement un contrôleur ?... Mais, rappelez-vous que l'on est dans un contexte client (navigateur) et que le concept MVC « classique » n'est pas forcément « portable » en l'état. L'essentiel est que cela fonctionne, et si les contrôleurs vous manquent à ce point, nous verrons comment en créer quelques chapitres plus loin.

## 8.1 Préparons le terrain

Pour repartir sur de bonnes bases, nous allons supprimer la base de données avec laquelle nous avons déjà bien joué. Donc supprimez le fichier `blog.db` de la racine de votre application. Ensuite, modifiez le code javascript de la page `index.html` dans le répertoire `/public`, donc dans la partie `<script></script>`, pour instancier une collection : (on ajoute : `window.blogPosts = new Posts();`)

*Instancier une collection :*

```
$($.function (){

    window.Post = Backbone.Model.extend({
        urlRoot :"/blogposts"

    });

    window.Posts = Backbone.Collection.extend({
        model : Post,
        all : function () {
            this.url = "/blogposts";
            return this;
        },
        query : function (query) {
            this.url = "/blogposts/query/" +query;
            return this;
        }
    });

    window.blogPosts = new Posts();
});
```

Sauvegardez, puis relancez votre application (`node app.js` ou `nodemon app.js`), dans le navigateur accédez à la page principale (<http://localhost:3000/>), pour enfin ouvrir la console de votre navigateur. Nous allons créer des modèles, que nous ajouterons à la collection `blogposts`.

### 8.1.1 Création et sauvegarde des modèles

Commencez par saisir ceci dans la console du navigateur :

*Ajouter des modèles à la collection :*

```
var messages = [
    "Maecenas sed diam eget risus varius blandit sit amet non magna.",
    "Integer posuere erat a ante venenatis dapibus posuere velit aliquet.",
    "Donec id elit non mi porta gravida at eget metus.",
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
    "Cras mattis consectetur purus sit amet fermentum.",
```

```

        "Nulla vitae elit libero, a pharetra augue."
    ]);

blogPosts.add([
    new Post({
        title : "Premier Message",
        message : messages[0], date : new Date(2012, 10, 23, 7,4,0,0),
        author : "bob"
    }),
    new Post({
        title : "Backbone ???",
        message : messages[1], date : new Date(2012, 10, 23, 7,5,0,0),
        author : "bob"
    }),
    new Post({
        title : "Les Modèles",
        message : messages[2], date : new Date(2012, 10, 23, 7,6,0,0),
        author : "sam"
    }),
    new Post({
        title : "Les Vues",
        message : messages[3], date : new Date(2012, 10, 23, 7,7,0,0),
        author : "sam"
    }),
    new Post({
        title : "Les Routes",
        message : messages[4], date : new Date(2012, 10, 23, 7,8,0,0),
        author : "bob"
    }),
    new Post({
        title : "Mais où sont les contrôleurs ?",
        message : messages[5], date : new Date(2012, 10, 23, 7,9,0,0),
        author : "bob"
    })
])

```

**Remarque :** en javascript, pour les dates, le chiffre 10 correspond à Novembre (faire +1)

Nous avons donc maintenant 5 Posts dans notre collection. Pour ne pas avoir à tout re-saisir à chaque fois, sauvegardez vos posts (toujours dans la console du navigateur) :

*Sauvegarder les modèles en base :*

```

blogPosts.each(function(post){
    post.save({},{
        success : function (post) {
            console.log(post.get("title")," sauvegardé");
        },
    })
})

```

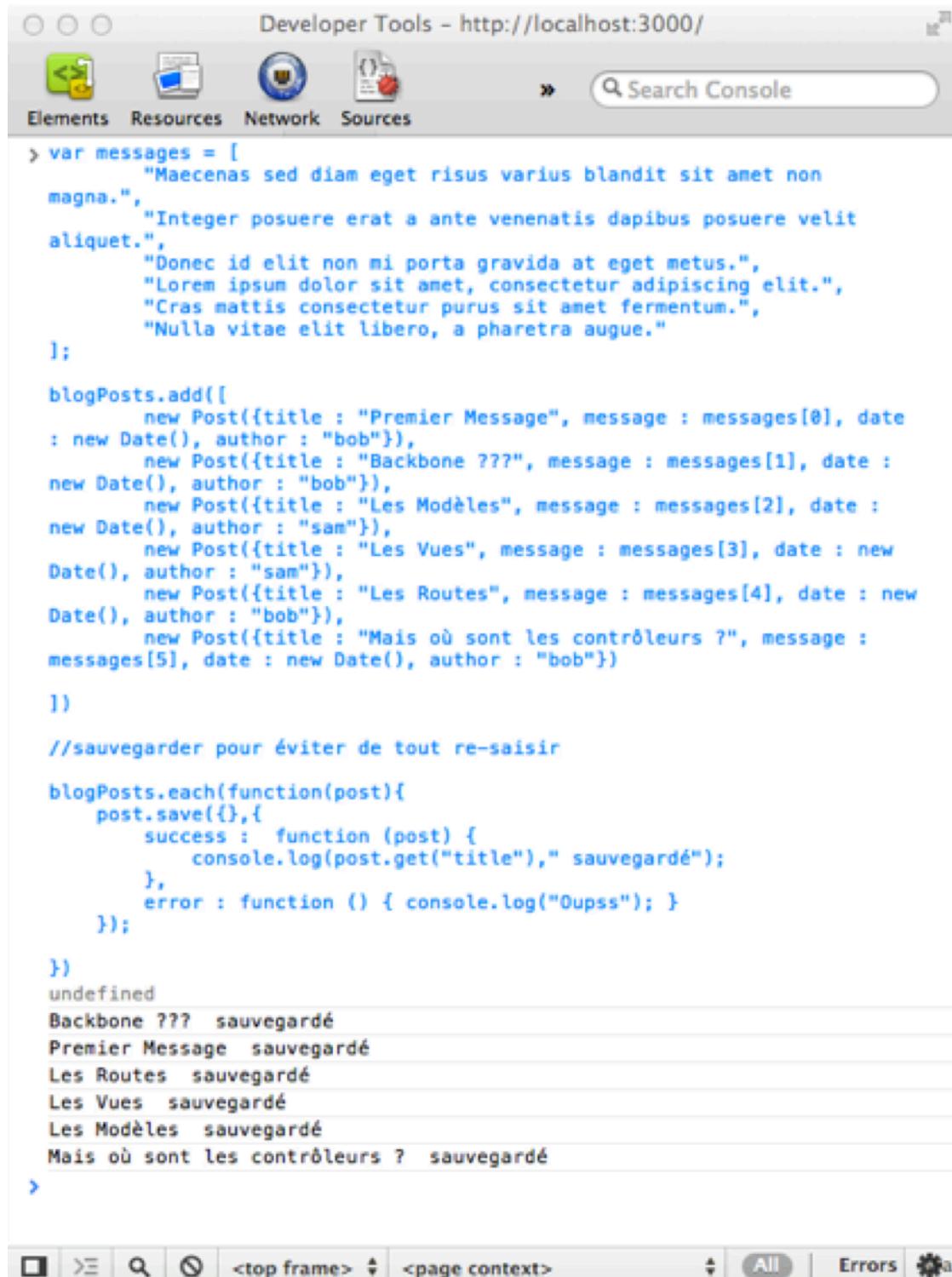
```

        error : function () { console.log("Oupss"); }
    });

})

```

Vous devriez au final obtenir ceci :



```

Developer Tools - http://localhost:3000/
Elements Resources Network Sources » Search Console

> var messages = [
    "Maecenas sed diam eget risus varius blandit sit amet non magna.",
    "Integer posuere erat a ante venenatis dapibus posuere velit aliquet.",
    "Donec id elit non mi porta gravida at eget metus.",
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
    "Cras mattis consectetur purus sit amet fermentum.",
    "Nulla vitae elit libero, a pharetra augue."
];

blogPosts.add([
    new Post({title : "Premier Message", message : messages[0], date : new Date(), author : "bob"}),
    new Post({title : "Backbone ???", message : messages[1], date : new Date(), author : "bob"}),
    new Post({title : "Les Modèles", message : messages[2], date : new Date(), author : "sam"}),
    new Post({title : "Les Vues", message : messages[3], date : new Date(), author : "sam"}),
    new Post({title : "Les Routes", message : messages[4], date : new Date(), author : "bob"}),
    new Post({title : "Mais où sont les contrôleurs ?", message : messages[5], date : new Date(), author : "bob"})
])

//sauvegarder pour éviter de tout re-saisir

blogPosts.each(function(post){
    post.save({},{
        success : function (post) {
            console.log(post.get("title")," sauvegardé");
        },
        error : function () { console.log("Oupss"); }
    });
});

undefined
Backbone ??? sauvegardé
Premier Message sauvegardé
Les Routes sauvegardé
Les Vues sauvegardé
Les Modèles sauvegardé
Mais où sont les contrôleurs ? sauvegardé
>

```

Pour vérifier que la sauvegarde a bien fonctionné, rafraîchissez votre page et lancez ce code dans la console du navigateur :

*Charger la collection avec les modèles sauvegardés en base :*

```
blogPosts.all()
  .fetch({
    success:function(result){
      console.log(result);
    }
})
```

Si tout va bien (il n'y a pas de raison), vous devriez obtenir ceci :

Developer Tools - http://localhost:3000/

Elements Resources Network Sources Search Console

```
> blogPosts.all()
  .fetch({
    success:function(result){
      console.log(result);
    }
})
▶ Object
  ▼ child
    ▶ _byCid: Object
    ▶ _byId: Object
    ▶ length: 6
    ▼ models: Array[6]
      ▷ 0: child
        ▶ _callbacks: Object
        ▶ _escapedAttributes: Object
        ▶ _pending: Object
        ▶ _previousAttributes: Object
        ▶ _silent: Object
        ▼ attributes: Object
          author: "bob"
          date: "2012-08-15T06:34:16.655Z"
          id: "lj42s2s"
          message: "Nulla vitae elit libero, a pharetra augue."
          saveDate: 1345012456700
          title: "Mais où sont les contrôleur?"
        ▶ __proto__: Object
        ▶ changed: Object
        ▶ cid: "c0"
        ▶ collection: child
        ▶ id: "lj42s2s"
        ▶ __proto__: ctor
      ▷ 1: child
      ▷ 2: child
      ▷ 3: child
      ▷ 4: child
      ▷ 5: child
      ▶ length: 6
      ▶ __proto__: Array[0]
      ▶ url: "/blogposts"
      ▶ __proto__: ctor
    ▷ 
```

All Errors

Ainsi, quoiqu'il se passe, vous disposez de tous vos messages et ne serez plus obligés de les ressaisir pour

la suite des exercices. Nous pouvons donc entrer dans le vif du sujet.

## 8.2 1ère vue

Un objet Vue dans Backbone (`Backbone.View`) et généralement composé (au minimum) par convention de :

- une propriété `el` : c'est l'élément du DOM (la partie de votre page html à laquelle on rattache l'objet `View`)
- une méthode `initialize` (déclenchée à l'instanciation de la vue)
- une méthode `render` (chargée d'afficher les données liées à la vue)

**Remarque :** Libre à vous de vous faire vos propres bonnes pratiques concernant les responsabilités de l'objet View afin de rendre votre code lisible et maintenable ... Vous trouverez toujours quelqu'un pour les discuter mais c'est comme cela que l'on apprend et s'améliore ... Et vous pouvez aussi avoir raison :-).

Dans notre page `index.html` nous allons ajouter un tag `<div id="posts_list"></div>` comme ceci :

```
<div class="container">
  <div class="hero-unit">
    <h1>Backbone rocks !!!</h1>
  </div>

  <div id="posts_list"></div>

</div>
```

Et modifier le code javascript de la manière suivante : entre la définition de la collection et son instantiation, ajoutez le code de notre première vue :

*1ère vue pour afficher le contenu de la collection :*

```
window.PostsListView = Backbone.View.extend({
  el : $("#posts_list"),
  initialize : function (data) {
    this.collection = data;
  },
  render : function () {
    var html = "";
    $(this.el).html(""); //on vide le div
    this.collection.each(function(model) {

      html += [
        '<h1>' + model.get("title") + '</h1><hr>',
        '<b>par : ' + model.get("author") + '</b> le : ' + model.get("date") + '<br>',
        '<p>' + model.get("message") + '</p>'
      ];
    });
    $(this.el).html(html);
  }
});
```

```

        ].join(""));

    });

    $(this.el).append(html);

}
));

```

### 8.2.1 Explications & utilisation

Notre vue `PostsListView` est reliée au tag `<div id="posts_list"></div>` par la propriété `el` qui n'est ni plus ni moins un objet **jQuery**. La méthode `initialize` (qui sera appelée à l'instanciation de la vue), prend en paramètre les données que nous souhaitons afficher, et les affecte à la propriété `collection` de la vue. La méthode `render`, vide le contenu du tag `<div id="posts_list"></div>`, parcourt la collection de données pour construire le code html, et enfin affiche celui-ci par la commande `$(this.el).append(html)`. Mais utilisons directement notre code, ce sera plus « parlant ».

Sauvegardez, rafraîchissez la page et en mode console, passez les étapes qui suivent :

*1] Chargez les données de la collection :*

```

blogPosts.all()
  .fetch({
    success:function(result){
      console.log(result);
    }
})

```

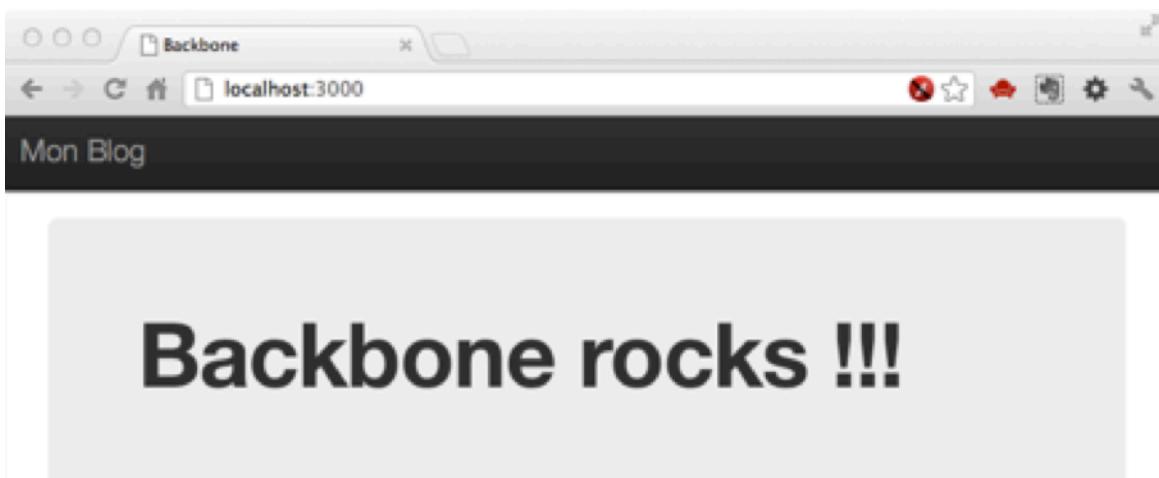
*2] Instanciez la vue en lui passant la collection en paramètre :*

```
postsListView = new PostsListView(blogPosts)
```

*3] Appelez la méthode render de la vue :*

```
postsListView.render()
```

Et vous obtenez la liste de vos messages :



## Backbone ???

par : bob le : 2012-08-15T06:34:16.655Z  
Integer posuere erat a ante venenatis dapibus posuere velit aliquet.

## Premier Message

par : bob le : 2012-08-15T06:34:16.654Z  
Maecenas sed diam eget risus varius blandit sit amet non magna.

## Les Routes

par : bob le : 2012-08-15T06:34:16.655Z  
Cras mattis consectetur purus sit amet fermentum.

## Les Vues

par : sam le : 2012-08-15T06:34:16.655Z  
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Souvenez vous, dans les chapitres précédents nous avions « donné » aux collections la possibilité de faire des requêtes sur les données avant de lancer un `fetch`. Essayez donc ceci dans la console de votre navigateur :

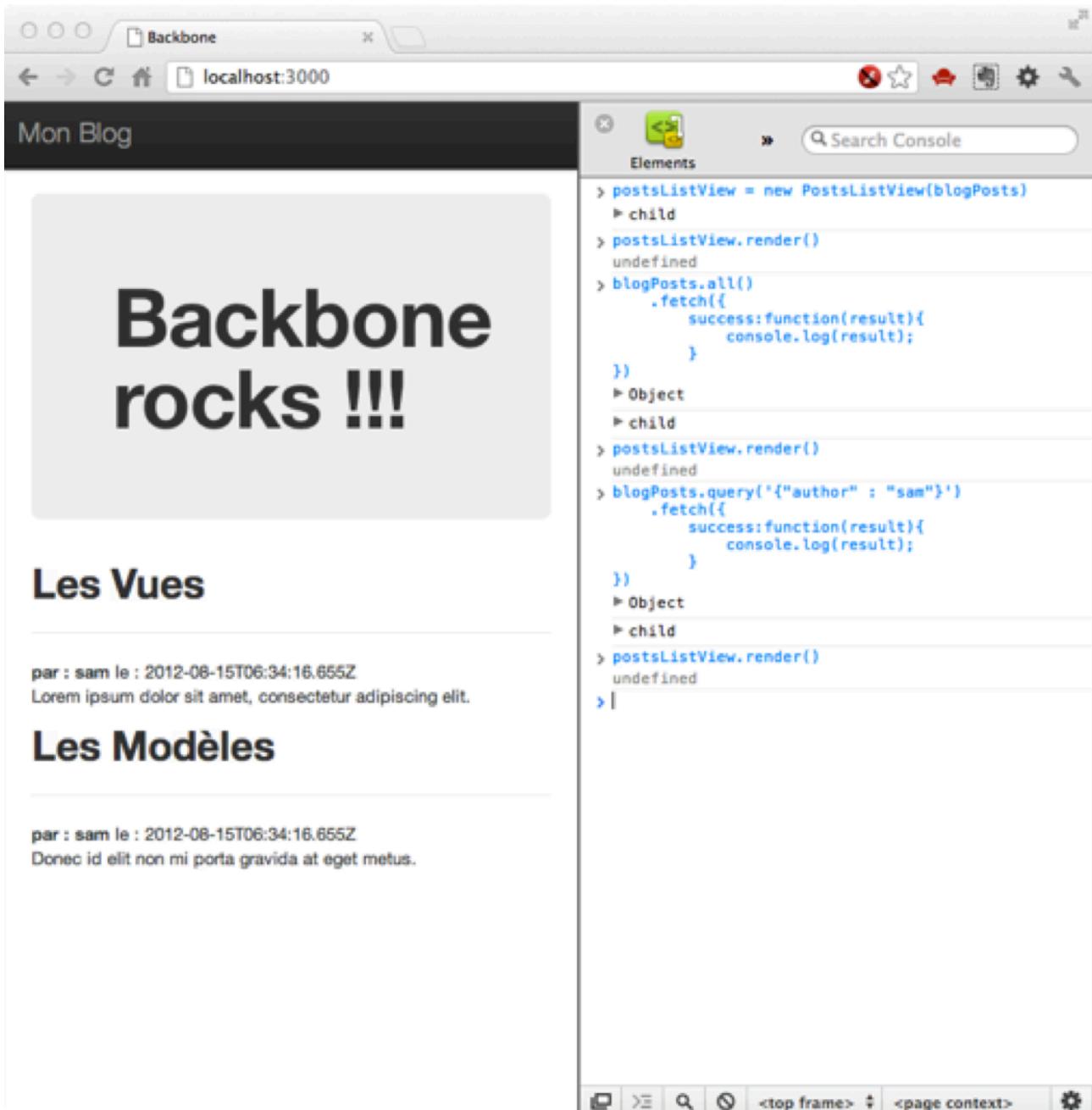
*Je ne veux que les posts de l'auteur “Sam” :*

```
blogPosts.query('{"author" : "sam"}')
  .fetch({
    success:function(result){
      console.log(result);
    }
  })
```

Puis faite à nouveau un :

```
postsListView.render()
```

Et là, l'affichage s'actualise automatiquement :



## 8.3 Maintenant, un peu de magie ...

### 8.3.1 S'abonner aux événements

Modifions une nouvelle fois notre vue en ajoutant le code suivant à la méthode `initialize` :

```

_.bindAll(this, 'render');
this.collection.bind('reset', this.render);

```

Nous venons d'expliquer que tous les évènements déclarés déclencheront la méthode `render` de la vue. Et ensuite nous avons expliqué que la méthode `reset` de la collection déclenchera la méthode `render` de la vue.

**Remarque :** Une collection Backbone déclenche un `reset` lors de l'appel d'un `fetch`. La méthode `reset` vide la collection.

```
//TODO: faire un chapitre à part sur '_.bindAll'
```

Le code de notre vue doit donc ressembler à ceci :

*PostListView :*

```
window.PostsListView = Backbone.View.extend({
  el : $('#posts_list'),
  initialize : function (data) {
    this.collection = data;

    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render);

  },
  render : function () {
    var html = '';
    $(this.el).html('');
    //on vide le div
    this.collection.each(function(model) {

      html += [
        '<h1>' + model.get("title") + '</h1><hr>',
        '<b>par : ' + model.get("author") + '</b> le : ' + model.get("date") + '<br>',
        '<p>' + model.get("message") + '</p>'
      ].join('');

    });

    $(this.el).append(html);
  }
});
```

Sauvegardez ensuite la page, puis retournez dans le navigateur, rafraîchissez la page et retournez dans la console du navigateur pour instancier une nouvelle vue :

```
postsListView = new PostsListView(blogPosts)
```

Puis essayez ceci :

```
blogPosts.all()
  .fetch({
    success:function(result){
      console.log(result);
    }
  })
```

et cela :

```
blogPosts.query('{"author" : "sam"}')
  .fetch({
    success:function(result){
      console.log(result);
    }
  })
})
```

Vous remarquez que votre vue se met à jour automatiquement à chaque changement, sans avoir à rappeler la méthode render de la vue. Mais il est possible de faire ceci aussi avec les changements sur les modèles.

### 8.3.2 S'abonner à d'autres évènements (modèles)

Toujours dans la méthode initialize de la vue, ajoutez le code suivant :

```
this.collection.bind('change', this.render);
this.collection.bind('add', this.render);
this.collection.bind('remove', this.render);
```

Maintenant, si vous changez la valeur d'un attribut d'un modèle, que vous ajoutez ou supprimez un modèle de la collection, la vue sera réactualisée à chaque fois.

Sauvegardez la page, puis retournez dans le navigateur, rafraîchissez la page et retournez dans la console du navigateur pour instancier une nouvelle vue et charger les données de la collection :

```
postsListView = new PostsListView(blogPosts)
blogPosts.all()
  .fetch({
    success:function(result){
      console.log(result);
    }
  })
})
```

Puis changez le titre du 1er post :

```
blogPosts.at(0).set("title","BACKBONE ???!!!")
```

ou ajoutez un post :

```
blogPosts.add(new Post({title:"HELLO",message : "salut", author : "k33g"}))
```

ou encore supprimez un post :

```
blogPosts.remove(blogPosts.at(0));
```

Là encore, votre page s'actualise instantanément.

### 8.3.3 Amélioration & Finalisation du code

Avant de passer à l'utilisation des templates dans les vue, nous allons apporter quelques modifications et améliorer un peu notre code pour nous préparer à la suite.

Dans ses dernières versions, Backbone a hérité d'un raccourcis concernant la propriété `el` de la vue qui consiste à remplacer (avec pour objectif l'optimisation d'exécution de code) le sélecteur `$(this.el)` par `this.$el`.

De plus, vous devez savoir qu'il n'est pas obligatoire de déclarer l'affectation de la collection dans la méthode initialize de la vue, mais que l'on peut faire ceci directement en paramètre du constructeur à l'instanciation de la vue. Comme ceci :

```
new PostsListView({collection : blogPosts})
```

On pourrait faire de même avec `el`, et utiliser ceci :

```
new PostsListView({el : $("#posts_list"), collection : blogPosts})
```

En fait tout dépend de vos besoins (et de vos habitudes).

En ce qui nous concerne, modifions le code de notre vue de la manière suivante :

*PostsListView* :

```
window.PostsListView = Backbone.View.extend({
  el : $("#posts_list"),
  initialize : function () {

    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render);
    this.collection.bind('change', this.render);
    this.collection.bind('add', this.render);
    this.collection.bind('remove', this.render);

  },
  render : function () {
    var html = "";
    this.$el.html(""); //on vide le div
    this.collection.each(function(model) {

      html += [
        '<h1>' + model.get("title") + '</h1><hr>',
        '<b>par : ' + model.get("author") + '</b> le : ' + model.get("date") + '<br>',
        '<p>' + model.get("message") + '</p>'
      ].join("");
    });

    this.$el.append(html);
  }
});
```

Puis à la fin du code javascript, ajoutez le code qui instancie la vue, ainsi que le code qui « charge » la collection (on se souvient que le render de la vue sera déclenché automatiquement une fois le `fetch` de la collection terminé.) :

```
window.postsListView = new PostsListView({collection : blogPosts})

blogPosts.all().fetch({
  success:function(result){
    console.log(result);
  }
});
```

Le code final du script dans la page devrait ressembler à ceci :

*Code final :*

```
<!-- === code applicatif === -->
<script>
$(function (){

  window.Post = Backbone.Model.extend({
    urlRoot :"/blogposts"

  });

  window.Posts = Backbone.Collection.extend({
    model : Post,
    all : function () {
      this.url = "/blogposts";
      return this;
    },
    query : function (query) {
      this.url = "/blogposts/query/"+query;
      return this;
    }
  });

  window.PostsListView = Backbone.View.extend({
    el : $("#posts_list"),
    initialize : function () {

      _.bindAll(this, 'render');
      this.collection.bind('reset', this.render);
      this.collection.bind('change', this.render);
      this.collection.bind('add', this.render);
      this.collection.bind('remove', this.render);

    },
    render : function () {
```

```

var html = '';
this.$el.html(''); //on vide le div
this.collection.each(function(model) {

    html += [
        '<h1>' + model.get("title") + '</h1><hr>',
        '<b>par : ' + model.get("author") + '</b> le : ' + model.get("date") + '<br>',
        '<p>' + model.get("message") + '</p>'
    ].join('');

});

this.$el.append(html);

}

});

window.blogPosts = new Posts();

window.postsListView = new PostsListView({collection : blogPosts})

blogPosts.all().fetch({
    success:function(result){
        //ça marche !!!
    }
});

});

</script>

```

Nous avons fait un peu de magie, passons donc à la sorcellerie ;) ...

## 8.4 Utilisation du templating ... 1ère fois

Vous vous souvenez ? Je vous avez parlé d'underscore avec les templates ? Eh bien il est temps de les mettre en œuvre.

### 8.4.1 Définition de notre 1er template

Dans la partie HTML de notre page, juste avant `<div id="posts_list"></div>`, ajoutez le code ci-dessous (ce sera notre template) :

```

<!-- template pour les posts -->
<script type="text/template" id="posts_list_template">

<% _.each(posts ,function(post){ %>
    <h1><%= post.get("title") %></h1><hr>

```

```

<b>par : <%= post.get("author") %></b> le : <%= post.get("date") %><br>
<p><%= post.get("message") %></p>
<% }); %>

</script>

```

**Remarque :** le fait de définir le template à l'intérieur de `<script type="text/template"></script>` fait que le modèle de template ne sera pas affiché dans la page.

En fait (grâce à underscore), nous venons de définir le template dont la vue backbone va se servir pour afficher les données. Il faudra lui passer pour cela un **tableau** de posts. Modifions donc notre vue de la façon suivante :

```

window.PostsListView = Backbone.View.extend({
  el : $("#posts_list"),
  initialize : function () {
    this.template = _.template($("#posts_list_template").html());

    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render);
    this.collection.bind('change', this.render);
    this.collection.bind('add', this.render);
    this.collection.bind('remove', this.render);

  },
  render : function () {
    var renderedContent = this.template({posts : this.collection.models});

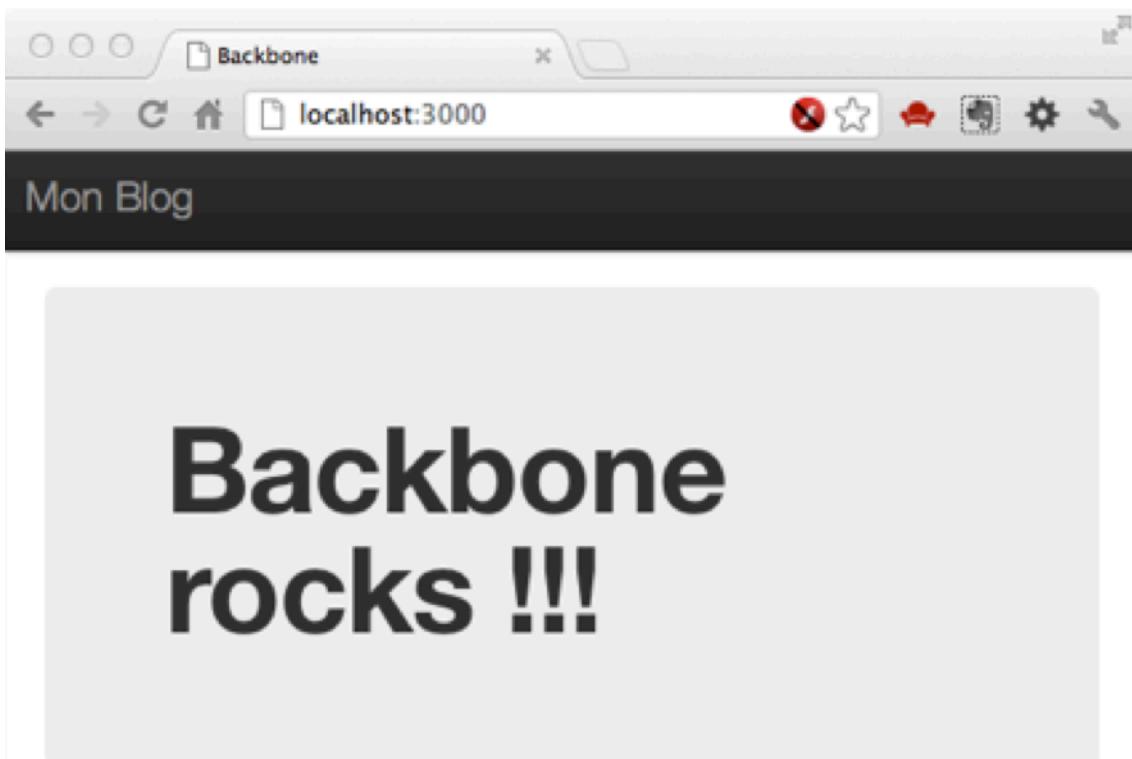
    this.$el.html(renderedContent);
  }
});

```

Nous avons inséré dans la méthode `initialize`: `this.template = _.template($("#posts_list_template").html());` où nous expliquons que la définition du template se trouve dans la zone ayant `posts_list_template` pour id. Nous avons aussi simplifié grandement le contenu de la méthode `render`, où nous faisons générer le contenu HTML à partir du template et des données.

**Remarque :** notez bien que `this.collection.models` est un tableau de modèles.

Vous pouvez sauvegarder et rafraîchir, les résultats sont identiques aux précédents, mais il est beaucoup plus facile de créer et modifier vos templates html.



## Premier Message

par : bob le : 2012-08-15T08:13:31.740Z  
Maecenas sed diam eget risus varius blandit sit amet non magna.

## Les Vues

par : sam le : 2012-08-15T08:13:31.741Z  
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

## Les Modèles

par : sam le : 2012-08-15T08:13:31.741Z  
Donec id elit non mi porta gravida at eget metus.

## Backbone ???

par : bob le : 2012-08-15T08:13:31.741Z  
Integer posuere erat a ante venenatis dapibus posuere velit aliquet.

## Les Routes

## 8.5 Sous-vue(s)

Il est possible de faire des sous vues pour gérer différentes parties de votre page web. En fait il s'agit de vues encapsulées dans une autre vue, ce qui peut être pratique en termes d'organisation, mais aussi dans les cas où les comportements de chacunes des vues dépendent les uns des autres.

Nous allons donc profiter des possibilités de Twitter Bootstrap pour revoir un peu la mise en page de notre « site » et du même coup mettre en œuvre le concept de sous-vue.

### 8.5.1 Réorganisation du code html

Modifions notre code html de la manière suivante :

```

<div class="navbar navbar-fixed-top">
    <div class="navbar-inner">
        <div class="container">
            <a class="brand">Mon Blog</a>
        </div>
    </div>
</div>

<div class="container-fluid">

    <div class="row-fluid">

        <div class="span3">
            <script type="text/template" id="blog_sidebar_template">
                <h2>Les 3 derniers :</h2>
                <ul>
                    <% _.each(posts ,function(post){ %>
                        <li><%= post.get("title") %></li>
                    <% }); %>
                </ul>
            </script>
            <div class="sidebar" id="blog_sidebar">
                <!-- Last 3 posts -->
            </div>
        </div>

        <div class="span9">
            <div class="hero-unit">
                <h1>Backbone rocks !!!</h1>
            </div>

            <!-- template pour Les posts -->
            <script type="text/template" id="posts_list_template">

                <% _.each(posts ,function(post){ %>
                    <h1><%= post.get("title") %></h1><hr>
                    <b>par : <%= post.get("author") %></b> le : <%= post.get("date") %><br>
                <% }); %>
            </script>
        </div>
    </div>
</div>
```

```

        <p><%= post.get("message") %></p>
    <% } ); %>

</script>
<div class="row-fluid" id="posts_list"></div>

</div>

</div>
</div>

```

**Explications :** nous avons donc 2 templates, un pour afficher les 3 derniers posts (`id="blog_sidebar_template"`), un pour afficher tous les posts (`id="posts_list_template"`).

### 8.5.2 Crédation & Modification des vues

Nous allons créer une vue principale (`MainView`) qui se chargera de “piloter” 2 sous-vues (`SidebarView` et `PostsListView`) :

1] Simplifions `PostsListView` :

```

window.PostsListView = Backbone.View.extend({
    el : $("#posts_list"),
    initialize : function () {
        this.template = _.template($("#posts_list_template").html());
    },
    render : function () {
        var renderedContent = this.template({posts : this.collection.models});
        this.$el.html(renderedContent);
    }
});

```

2] Crédation de `SidebarView` :

```

window.SidebarView = Backbone.View.extend({
    el : $("#blog_sidebar"),
    initialize : function () {
        this.template = _.template($("#blog_sidebar_template").html());
    },
    render : function () {
        var renderedContent = this.template({posts : this.collection.models});
        this.$el.html(renderedContent);
    }
});

```

3] Crédation de la vue principale :

```

window.MainView = Backbone.View.extend({
  initialize : function () {

    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render);
    this.collection.bind('change', this.render);
    this.collection.bind('add', this.render);
    this.collection.bind('remove', this.render);

    this.sidebarView = new SidebarView();
    this.postsListView = new PostsListView({collection : blogPosts});

  },
  render : function () {
    this.sidebarView.collection = new Posts(this.collection.first(3));
    this.sidebarView.render();
    this.postsListView.render();
  }
});

```

C'est donc maintenant la vue MainView qui s'abonne aux changements de la collection et déclenche le rendu des 2 autres vues.

Et enfin instancions la collection ainsi que la vue principale (qui se chargera d'instancier les deux sous-vues). Donc à la place de :

```

window.blogPosts = new Posts();
window.postsListView = new PostsListView({collection : blogPosts})

```

**Nous aurons ceci :**

```

window.blogPosts = new Posts();
window.mainView = new MainView({collection : blogPosts});

```

Vous pouvez sauvegarder votre code et rafraîchir votre page :

**Les 3 derniers :**

- Premier Message
- Les Vues
- Les Modèles

# Backbone rocks

## !!!

### Premier Message

par : bob le : 2012-08-15T08:13:31.740Z  
Maecenas sed diam eget risus varius blandit sit amet non magna.

### Les Vues

par : sam le : 2012-08-15T08:13:31.741Z  
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

### Les Modèles

par : sam le : 2012-08-15T08:13:31.741Z  
Donec id elit non mi porta gravida at eget metus.

### Backbone ???

par : bob le : 2012-08-15T08:13:31.741Z  
Integer posuere erat a ante venenatis dapibus posuere velit aliquet.

### Les Routes

Et si vous faites ceci en mode console :

```
blogPosts.at(0).set("title","Bonjour à tous !")
```

Vous verrez que les modifications sont bien propagées dans les 2 vues simultanément.

#### 8.5.3 Un dernier petit réglage : tri des collections

Nous souhaitons avant d'aller plus loin trier la collection de posts pour avoir les message en ordre décroissant. Pour cela nous allons créer ce que l'on appelle un « **comparator** » dans la méthode `initialize` de la vue principale `MainView` :

*Trier la collection par ordre décroissant de date :*

```
this.collection.comparator = function (model) {
    return -(new Date(model.get("date")).getTime());
}
```

Donc le code final de MainView sera celui-ci :

*Vue principale :*

```
window.MainView = Backbone.View.extend({
  initialize : function () {

    this.collection.comparator = function (model) {
      return -(new Date(model.get("date"))).getTime();
    }

    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render);
    this.collection.bind('change', this.render);
    this.collection.bind('add', this.render);
    this.collection.bind('remove', this.render);

    this.sidebarView = new SidebarView();
    this.postsListView = new PostsListView({collection : this.collection});

  },
  render : function () {

    this.sidebarView.collection = new Posts(this.collection.first(3));
    this.sidebarView.render();
    this.postsListView.render();
  }
});
```

Et le rendu dans le navigateur devrait vous donner ceci :

//TODO : faire un paragraphe sur le comparator dans le chapitre sur les collections

## 8.6 Utilisation d'autre(s) moteur(s) de template

Vous lirez souvent que Backbone est “framework agnostic”, donc vous pouvez par exemple l'utiliser avec **zepto**, plutôt que **jQuery** en ce qui concerne la gestion du DOM et des appels Ajax. Il en est de même avec le moteur de template. Rien ne vous oblige à utiliser celui d'**Underscore**. Un des plus utilisé est Mustache.js (<http://mustache.github.com/>). Vous pouvez récupérer le code ici : <https://github.com/janl/mustache.js/>. En fait, plus précisément, enregistrez le fichier <https://raw.github.com/janl/mustache.js/master/mustache.js> dans votre répertoire `public/libs/vendors`. Puis faites y référence dans votre page `index.html` :

```
<script src="libs/vendors/mustache.js"></script>
```

Et nous allons une fois de plus “casser” notre code html.

### 8.6.1 Redéfinissons donc nos templates

1] Avant pour la partie concernant la vue `SidebarView` nous avions ceci :

```
<script type="text/template" id="blog_sidebar_template">
  <h2>Les 3 derniers :</h2>
  <ul>
    <% _.each(posts ,function(post){ %>
      <li><%= post.get("title") %></li>
    <% }); %>
  </ul>
</script>
```

2] Que vous allez remplacer par ceci :

```
<script type="text/template" id="blog_sidebar_template">
  <h2>Les 3 derniers :</h2>

  <ul>{{#posts}}
    <li>{{title}}</li>
  {{/posts}}</ul>

</script>
```

3] En ce qui concerne le template lié à la vue `PostsListView`, remplacez :

```
<script type="text/template" id="posts_list_template">

  <% _.each(posts ,function(post){ %>
    <h1><%= post.get("title") %></h1><hr>
    <b>par : <%= post.get("author") %></b> le : <%= post.get("date") %><br>
    <p><%= post.get("message") %></p>
  <% }); %>

</script>
```

4] Par :

```
<script type="text/template" id="posts_list_template">

  {{#posts}}
    <h1>{{title}}</h1>
    <b>par : {{author}}</b> le : {{date}}<br>
    <p>{{message}}</p>
  {{/posts}}

</script>
```

Nous obtenons donc des templates html plus lisibles, utilisable moyennant une petite modification de nos vues :

5] Avant (avec le moteur de template d'underscore) :

```
window.PostsListView = Backbone.View.extend({
  el : $("#posts_list"),
  initialize : function () {
    this.template = _.template($("#posts_list_template").html());
  },
  render : function () {
    var renderedContent = this.template({posts : this.collection.models});
    this.$el.html(renderedContent);
  }
});

window.SidebarView = Backbone.View.extend({
  el : $("#blog_sidebar"),
  initialize : function () {
    this.template = _.template($("#blog_sidebar_template").html());
  },
  render : function () {
    var renderedContent = this.template({posts : this.collection.models});
    this.$el.html(renderedContent);
  }
});
```

6] Après (en utilisant Mustache) nous aurons ceci :

```
window.PostsListView = Backbone.View.extend({
  el : $("#posts_list"),
  initialize : function () {
    this.template = $("#posts_list_template").html();
  },
  render : function () {
    var renderedContent = Mustache.to_html(
      this.template,
      {posts : this.collection.toJSON()});
    this.$el.html(renderedContent);
  }
});

window.SidebarView = Backbone.View.extend({
  el : $("#blog_sidebar"),
  initialize : function () {
    this.template = $("#blog_sidebar_template").html();
  },
  render : function () {
```

```

    var renderedContent = Mustache.to_html(
      this.template,
      {posts : this.collection.toJSON()})
    );

    this.$el.html(renderedContent);
  }
});

```

Vous noterez l'utilisation de `this.collection.toJSON()` plutôt que `this.collection.models`. En effet Mustache a besoin d'objets au format JSON, et (cela tombe bien), les collections Backbone dispose d'une méthode d'exportation/mise en forme au format JSON.

Sauvegardez, lancez, il n'y a pas de changement, l'affichage est identique (heureusement), vous avez juste utilisé une autre façon de travailler.

## 8.7 Gestion des événements dans les vues

Les objets de type `Backbone.View` peuvent aussi gérer les événements (`mouseover`, `click`, etc. ...). Nous allons donc profiter de ce paragraphe pour mettre en œuvre un système d'authentification dans notre application, qui utilisera donc cette possibilité. Il est temps de retourner travailler côté serveur quelques instants.

Si vous voulez en savoir plus sur les événements dans les vues Backbone, je vous engage fortement à lire la documentation : <http://backbonejs.org/#View-delegateEvents>.

Donc ...

## 8.8 Authentification (côté serveur) : les utilisateurs

*Ce paragraphe ne parle pas des vues, mais est nécessaire pour la mise en place des paragraphes suivants.*

Nous aurons besoin d'une liste des utilisateurs connectés (je vous le rappelle, nous sommes côté serveur, donc dans le fichier `app.js`) que nous représenterons sous la forme d'un tableau de variables (ou d'objets) :

```
var connectedUsers = [];
```

Nous aurons besoin d'ajouter des utilisateurs dans notre base de données, donc nous allons nous créer de quoi rajouter au moins une fois quelques utilisateurs pour notre application :

*Ajouter un utilisateur en base :*

```

function addUser (user) {
  users.save(null,user, function (err, key){
    if(err) {
      console.log("Erreur : ",err);
    } else {
      user.id = key;
    }
  });
}

```

```

        console.log(user);
    }
});
}

```

Nous appellerons n fois cette fonction pour ajouter des utilisateurs :

*Ajouter des utilisateurs :*

```

function addUsers () {
    addUser({
        email : "bob@morane.com",
        password : "backbone",
        isAdmin : true,
        firstName : "Bob",
        lastName : "Morane"
    });
    addUser({
        email : "sam@lepirate.com",
        password : "underscore",
        isAdmin : false,
        firstName : "Sam",
        lastName : "Le Pirate"
    });

    //etc. ...
}

```

Et pour déclencher l'ajout des utilisateurs, nous créons une « route » `addusers` :

```

app.get('/addusers', function(req, res){
    addUsers();
    res.json({MESSAGE:"Users added."});
})

```

Qu'il suffira d'appeler comme ceci dans le navigateur : <http://localhost:3000/addusers/>

**Remarque :** notez bien que mon système d'authentification est très « léger ». En production, il vous faudrait quelque chose de plus abouti, mais ce n'est pas le propos de cet ouvrage. Nous avions besoin de quelque chose de simple.

### 8.8.1 S'authentifier – Se déconnecter

Nous aurons besoin de nous authentifier. Il nous faut donc d'abord une fonction « utilitaire » qui nous permette de vérifier si l'email de l'utilisateur n'est pas déjà pris (utilisateur déjà connecté sous une autre session) :

*Vérifier si un utilisateur est déjà connecté :*

```

function findUserByEmail (email) {
    /*
        Permet de vérifier si un utilisateur est déjà Loggé
    */
    return connectedUsers.filter(function(user) {
        return user.email == email;
    })[0];
}

```

Nous allons donc créer une route `authenticate` avec le code suivant :

*Code pour s'authentifier :*

```

app.post('/authenticate', function(req, res){
    console.log("POST authenticate ", req.body);
    //Je récupère les information de connexion de l'utilisateur
    var user = req.body;

    //est ce que l'email est déjà utilisé ?
    if(findUserByEmail(user.email)) {
        res.json({infos:"Utilisateur déjà connecté"})
    } else { //si l'email n'est pas utilisé
        //Je cherche l'utilisateur dans la base de données
        users.find({email: user.email, password: user.password },
            function(err, results) {
                if(err) {
                    res.json({error:"Oups, Houson, on a un problème"});
                } else {
                    //J'ai trouvé l'utilisateur
                    var key = Object.keys(results)[0]
                    ,   authenticatedUser = results[key];

                    //Je rajoute l'id de session à l'objet utilisateur

                    authenticatedUser.key = key;
                    authenticatedUser.sessionID = req.sessionID;

                    //Ajouter l'utilisateur authentifié à la liste des utilisateurs connectés
                    connectedUsers.push(authenticatedUser);

                    //Je renvoie au navigateur les informations de l'utilisateur
                    // ... sans le mot de passe bien sûr
                    res.json({
                        email:authenticatedUser.email,
                        firstName : authenticatedUser.firstName,
                        lastName : authenticatedUser.lastName,
                        isAdmin : authenticatedUser.isAdmin
                    });
                }
            });
    }
}

```

```
});
```

**Remarque :** La « bienséance » (d'un point de vue architecture) voudrait que ne mette pas tout ce code au niveau de la route mais dans la méthode d'un contrôleur qui serait appelée par la route. Une fois de plus je vais au plus court, mais gardez à l'esprit : toujours un code lisible et maintenable.

Il faudra aussi pouvoir se déconnecter. Nous ajoutons donc une route `logoff` qui nous permettra de déconnecter l'utilisateur.

Nous avons tout d'abord besoin d'une fonction nous permettant de retrouver un utilisateur par son id de session parmi les utilisateurs connectés :

```
function findUserBySession (sessionId) {
  /*
    Permet de retrouver un utilisateur par son id de session
  */
  return connectedUsers.filter(function(user) {
    return user.sessionID == sessionId;
  })[0];
}
```

Que nous allons utiliser ensuite dans notre route `logoff` :

*Se déconnecter :*

```
app.get('/logoff', function(req, res){
  //Je recherche l'utilisateur courant parmi les utilisateurs connectés
  var alreadyAuthenticatedUser = findUserBySession(req.sessionID);

  if(alreadyAuthenticatedUser) {
    //Je l'ai trouvé, je le supprime de la liste des utilisateurs connectés
    var posInArray = connectedUsers.indexOf(alreadyAuthenticatedUser);
    connectedUsers.splice(posInArray, 1);
    res.json({state:"disconnected"});
  } else {
    res.json({});
  }
});
```

Nous aurons aussi besoin d'un moyen pour savoir (côté client) si un utilisateur est déjà connecté. Donc nous allons créer une route `alreadyauthenticated` que la page web pourra « appeler » pour vérification (par exemple au rechargement de la page) :

*Est-ce que je suis déjà authentifié ?*

```

app.get('/alreadyauthenticated', function(req, res){
    var alreadyAuthenticatedUser = findUserBySession(req.sessionID);

    //Si je suis déjà authentifié, renvoyer les informations utilisateur
    if(alreadyAuthenticatedUser) {
        res.json({
            email:alreadyAuthenticatedUser.email,
            firstName : alreadyAuthenticatedUser.firstName,
            lastName : alreadyAuthenticatedUser.lastName,
            isAdmin : alreadyAuthenticatedUser.isAdmin
        });
    } else {
        res.json({}); 
    }
});

});

```

Nous sommes maintenant prêts à utiliser tout cela côté client. Le code définitif de `app.js` devrait ressembler à ceci :

```

/*
----- Déclaration des librairies -----
var express = require('express')
, nStore = require('nstore')
, app = module.exports = express.createServer();

nStore = nStore.extend(require('nstore/query'))();

/*
----- Paramétrages de fonctionnement d'Express -----
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.static(__dirname + '/public'));
app.use(express.cookieParser('ilovebackbone'));
app.use(express.session({ secret: "ilovebackbone" }));

/*
----- Définition des "bases" posts & users -----
var posts, users;

posts = nStore.new("blog.db", function() {
    users = nStore.new("users.db", function() {
        Routes();
        app.listen(3000);
        console.log('Express app started on port 3000');
    });
});

```

```

        });
    });

/*===== Authentification =====*/
var connectedUsers = [];

function addUser (user) {
    users.save(null,user, function (err, key){
        if(err) {
            console.log("Erreur : ",err);
        } else {
            user.id = key;
            console.log(user);
        }
    });
}

function addUsers () {
    addUser({
        email : "bob@morane.com",
        password : "backbone",
        isAdmin : true,
        firstName : "Bob",
        lastName : "Morane"
    });
    addUser({
        email : "sam@lepirate.com",
        password : "underscore",
        isAdmin : false,
        firstName : "Sam",
        lastName : "Le Pirate"
    });
}

//etc. ...
}

function findUserBySession (sessionID) {
/*
    Permet de retrouver un utilisateur par son id de session
*/
    return connectedUsers.filter(function(user) {
        return user.sessionID == sessionID;
    })[0];
}

function findUserByMail (email) {
/*

```

```

Permet de vérifier si un utilisateur est déjà loggé
*/
return connectedUsers.filter(function(user) {
    return user.email == email;
})[0];
}

function Routes() {
    /*===== Routes pour authentification =====*/
    app.get('/addusers', function(req, res){
        addUsers();
        res.json({MESSAGE:"Users added."});
    });

    app.get('/alreadyauthenticated', function(req, res){
        var alreadyAuthenticatedUser = findUserBySession(req.sessionID);

        /* Si je suis déjà authentifié,
           renvoyer les informations utilisateur
           sans le mot de passe bien sûr
        */
        if(alreadyAuthenticatedUser) {
            res.json({
                email:alreadyAuthenticatedUser.email,
                firstName : alreadyAuthenticatedUser.firstName,
                lastName : alreadyAuthenticatedUser.lastName,
                isAdmin : alreadyAuthenticatedUser.isAdmin
            });
        } else {
            res.json({});
        }
    });

    app.post('/authenticate', function(req, res){
        console.log("POST authenticate ", req.body);
        //Je récupère les information de connexion de l'utilisateur
        var user = req.body;

        //est ce que l'email est déjà utilisé ?
        if(findUserByMail(user.email)) {
            res.json({infos:"Utilisateur déjà connecté"})
        } else { //si l'email n'est pas utilisé
            //Je cherche l'utilisateur dans la base de données
            users.find({email: user.email, password: user.password }, function(err, result)
            if(err) {
                res.json({error:"Oups, Houson, on a un problème"});
            }
        }
    });
}

```

```

    } else {
        //J'ai trouvé l'utilisateur
        var key = Object.keys(results)[0]
        , authenticatedUser = results[key];

        //Je rajoute l'id de session à l'objet utilisateur

        authenticatedUser.key = key;
        authenticatedUser.sessionID = req.sessionID;

        //J'ajoute l'utilisateur authentifié à la liste des utilisateurs connexes
        connectedUsers.push(authenticatedUser);

        //Je renvoie au navigateur les informations de l'utilisateur
        // ... sans le mot de passe bien sûr
        res.json({
            email:authenticatedUser.email,
            firstName : authenticatedUser.firstName,
            lastName : authenticatedUser.lastName,
            isAdmin : authenticatedUser.isAdmin
        });
    }
});

//);
}

app.get('/logoff',function(req, res){

    //Je recherche l'utilisateur courant parmi les utilisateurs connectés
    var alreadyAuthenticatedUser = findUserBySession(req.sessionID);

    if(alreadyAuthenticatedUser) {
        //Je l'ai trouvé, je le supprime de la liste des utilisateurs connectés
        var posInArray = connectedUsers.indexOf(alreadyAuthenticatedUser);
        connectedUsers.splice(posInArray, 1);
        res.json({state:"disconnected"});
    } else {
        res.json({});
    }

});

/*===== Fin des routes "authentification" =====*/
/*
Obtenir la liste de tous les posts lorsque
l'on appelle http://localhost:3000/blogposts
en mode GET
*/

```

```

app.get('/blogposts',function(req, res){
    console.log("GET (ALL) : /blogposts");
    posts.all(function(err, results) {

        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        } else {
            var posts = [];
            for(var key in results) {
                var post = results[key]; post.id = key;
                posts.push(post);
            }
            res.json(posts);
        }
    });
});

/*
Obtenir la liste de tous les posts correspondant à un critère
Lorsque l'on appelle http://localhost:3000/blogposts/ en
mode GET avec une requête en paramètre
ex : query : { "title" : "Mon 1er post" } */
app.get('/blogposts/:query',function(req, res){
    console.log("GET (QUERY) : /blogposts/" +req.params.query);

    posts.find(JSON.parse(req.params.query), function(err, results) {
        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        } else {
            var posts = [];
            for(var key in results) {
                var post = results[key]; post.id = key;
                posts.push(post);
            }
            res.json(posts);
        }
    });
});

/*
Retrouver un post par sa clé unique lorsque
l'on appelle http://localhost:3000/blogpost/identifiant_du_post
en mode GET */

```

```

app.get('/blogpost/:id', function(req, res){
  console.log("GET : /blogpost/" + req.params.id);
  posts.get(req.params.id, function(err, post, key) {
    if(err) {
      console.log("Erreur : ",err);
      res.json(err);
    } else {
      post.id = key;
      res.json(post);
    }
  });
});

/*
Créer un nouveau post Lorsque
l'on appelle http://localhost:3000/blogpost
avec en paramètre le post au format JSON
en mode POST
*/
app.post('/blogpost',function(req, res){
  console.log("POST CREATE ", req.body);

  var d = new Date(), model = req.body;
  model.saveDate = (d.valueOf());

  posts.save(null,model, function (err, key){
    if(err) {
      console.log("Erreur : ",err);
      res.json(err);
    } else {
      model.id = key;
      res.json(model);
    }
  });
});

/*
Mettre à jour un post lorsque
l'on appelle http://localhost:3000/blogpost
avec en paramètre le post au format JSON
en mode PUT
*/
app.put('/blogpost/:id',function(req, res){
  console.log("PUT UPDATE", req.body, req.params.id);

  var d = new Date(), model = req.body;
  model.saveDate = (d.valueOf());
}

```

```

posts.save(req.params.id,model, function (err, key){
    if(err) {
        console.log("Erreur : ",err);
        res.json(err);
    } else {
        res.json(model);
    }
});

/*
supprimer un post par sa clé unique lorsque
l'on appelle http://localhost:3000/blogpost/identifiant_du_post
en mode DELETE
*/
app.delete('/blogpost/:id',function(req, res){
    console.log("DELETE : /delete/" +req.params.id);

    posts.remove(req.params.id, function(err){
        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        } else {
            //petit correctif de contournement (bug ds nStore) :
            //ré-ouvrir la base lorsque la suppression a été faite
            posts = nStore.new("blog.db", function() {
                res.json(req.params.id);
                //Le modèle est vide si on ne trouve rien
            });
        }
    });
});
}

```

## 8.9 Authentification (côté client)

Nous repassons enfin au code client et nous allons pouvoir vérifier comment sont gérés les évènements dans une vue en implémentant l'authentification côté client.

### 8.9.1 Formulaire d'authentification

Nous allons donc commencer par créer le template du formulaire d'authentification dans notre page `index.html` (je choisis de le placer juste après la liste des 3 derniers posts) :

```

<div class="sidebar" id="blog_sidebar">
    <!-- Last 3 posts -->
</div>

```

```
<!-- ===== Formulaire d'authentification ===== -->
<script type="text/template" id="blog_login_form_template">
  <h3>Login :</h3>
  <input name="email" type="text" placeholder="email"/><br>
  <input name="password" type="password" placeholder="password"/><br>
  <a href="#" class="btn btn-primary">Login</a>
  <a href="#" class="btn btn-inverse">Logoff</a><br>
  <b>{{message}} {{firstName}} {{lastName}}</b>

</script>
<form class="container" id="blog_login_form">

</form>
```

### 8.9.2 L'objet Backbone.View : Login.View

Notre composant d'authentification aura 2 zones de saisie (email et mot de passe), un bouton de login, un bouton pour se déconnecter, une zone pour afficher un message (bienvenue, erreur, ...). Le composant devra aussi pouvoir vérifier si l'utilisateur est toujours connecté en cas de rafraîchissement de la page.

```
/*===== Authentication =====*/
window.LoginView = Backbone.View.extend({
  el : $("#blog_login_form"),

  initialize : function () {
    var that = this;
    this.template = $("#blog_login_form_template").html();

    //on vérifie si pas déjà authentifié
    $.ajax({type:"GET", url:"/alreadyauthenticated",
      error:function(err){ console.log(err); },
      success:function(dataFromServer) {

        if(dataFromServer.firstName) {
          that.render("Bienvenue",dataFromServer);
        } else {
          that.render("????",{firstName:"John", lastName:"Doe"});
        }
      }
    })
  },

  render : function (message, user) {

    var renderedContent = Mustache.to_html(this.template, {
      message : message,
      firstName : user ? user.firstName : "",
      lastName : user ? user.lastName : ""
    })
  }
});
```

```

    });
    this.$el.html(renderedContent);
}

});

window.loginView = new LoginView();
/*===== Fin authentication =====*/

```

A l'initialisation (`initialize`) la vue va vérifier si l'utilisateur en cours est déjà authentifié (par exemple vous vous êtes signé, mais vous avez rafraîchi la page), en appelant la route `/alreadyauthenticated`, si l'utilisateur est déjà authentifié, la méthode `render` de la vue est appelée avec un message de bienvenue et les informations de l'utilisateur (`that.render("Bienvenue", dataFromServer);`) dans le cas contraire la méthode `render` est aussi appelée mais avec un message signifiant que l'utilisateur n'est pas connecté (`that.render("???", {firstName:"John", lastName:"Doe"});`).

### 8.9.3 Ajoutons une gestion des évènements

Une propriété de l'objet `Backbone.View` permet de gérer les évènements spécifiques à la vue. Si vous vous souvenez, notre template de formulaire ressemble à ceci :

```

<!-- /*===== Formulaire d'authentification =====*/ -->
<script type="text/template" id="blog_login_form_template">
  <h3>Login :</h3>
  <input name="email" type="text" placeholder="email"/><br>
  <input name="password" type="password" placeholder="password"/><br>
  <a href="#" class="btn btn-primary">Login</a>
  <a href="#" class="btn btn-inverse">Logoff</a><br>
  <b>{{message}} {{firstName}} {{lastName}}</b>
</script>

```

Je souhaite pouvoir déclencher des évènements lorsque je clique sur les boutons du formulaire. Pour cela il suffit d'ajouter à l'objet `Backbone.View` la propriété `events` :

```

events : {
  "click .btn-primary" : "onClickBtnLogin",
  "click .btn-inverse" : "onClickBtnLogoff"
},

```

En fait je demande à mon objet `Backbone.View` d'intercepter tous les évènements de type `click` sur les éléments html (de la vue considérée) dont la classe css est `.btn-primary` ou `.btn-inverse` et de déclencher respectivement les méthodes `onClickBtnLogin` ou `onClickBtnLogoff`.

**Remarque :** nous aurions très bien pu affecter des id aux boutons :

```

<a href="#" id="btnLogIn" class="btn btn-primary">Login</a>
<a href="#" id="btnLogOff" class="btn btn-inverse">Logoff</a><br>

```

et relier les évènements aux ids :

```
events : {
    "click #btnLogIn" : "onClickBtnLogin",
    "click #btnLogOff" : "onClickBtnLogoff"
},
```

Il ne nous reste plus qu'à écrire les méthodes `onClickBtnLogin` ou `onClickBtnLogoff` au sein de notre objet de type `Backbone.View` qui vont respectivement appeler les routes que nous avions définies précédemment :

```
onClickBtnLogin : function (domEvent) {

    var fields = $("#blog_login_form :input")
    ,   that = this;

    $.ajax({
        type:"POST",
        url:"/authenticate",
        data : { email : fields[0].value, password : fields[1].value } ,
        dataType : 'json',
        error:function(err){ console.log(err); },
        success:function(dataFromServer) {

            if(dataFromServer.infos) {
                that.render(dataFromServer.infos);
            } else {
                if(dataFromServer.error) {
                    that.render(dataFromServer.error);
                } else {
                    that.render("Bienvenue",dataFromServer);
                }
            }
        }

    });
},
onClickBtnLogoff : function() {

    var that = this;
    $.ajax({type:"GET", url:"/logoff",
        error:function(err){ console.log(err); },
        success:function(dataFromServer) {
            console.log(dataFromServer);
            that.render("???",{firstName:"John", lastName:"Doe"});
        }
    })
}
```

### 8.9.4 Vérification

Si vous avez bien suivi, nous devons avoir au moins 2 utilisateurs en base de données :

```
function addUsers () {
  addUser({
    email : "bob@morane.com",
    password : "backbone",
    isAdmin : true,
    firstName : "Bob",
    lastName : "Morane"
  });
  addUser({
    email : "sam@lepirate.com",
    password : "underscore",
    isAdmin : false,
    firstName : "Sam",
    lastName : "Le Pirate"
  });

  //etc. ...
}
```

Lançons donc notre page web :



Authentifiez vous en tapant n'importe quoi :



### Les 3 derniers :

- Mais où sont les contrôleurs ?
- Les Routes
- Backbone ???

Login :



[Login](#) [Logoff](#)

Ouups !!! Loupé !

# Backbone rocks !!!

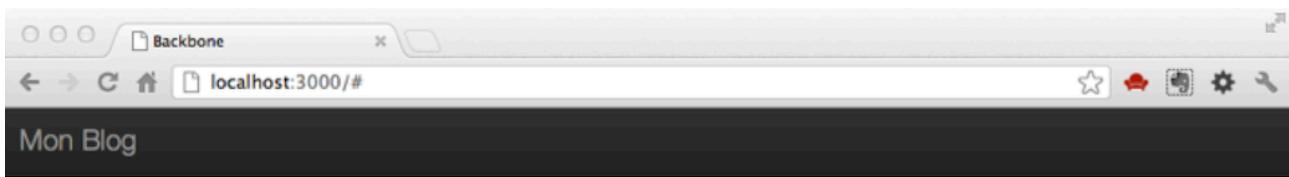
## Mais où sont les contrôleurs ?

par : bob le : 2012-08-15T08:13:36.741Z

Nulla vitae elit libero, a pharetra augue.

Vous obtenez le message “**Ouups loupé !!!**”

Authentifiez vous en utilisant un des utilisateurs existant :



### Les 3 derniers :

- Mais où sont les contrôleurs ?
- Les Routes
- Backbone ???

Login :



[Login](#) [Logoff](#)

Bienvenue Bob Morane

# Backbone rocks !!!

## Mais où sont les contrôleurs ?

par : bob le : 2012-08-15T08:13:36.741Z

Nulla vitae elit libero, a pharetra augue.

Vous obtenez un message de bienvenue.

Vous pouvez essayer de rafraîchir la page, vous restez connecté.

Si vous ouvrez un autre navigateur (une autre marque de navigateur pour être sûr de ne pas partager la session), vous vous apercevez qu'il ne considère pas que vous êtes authentifié :

Essayez de vous connecter avec un utilisateur déjà loggé sur une autre session :

Vous obtenez le message “**Utilisateur déjà connecté**”

Vous disposez maintenant de suffisamment d’élément pour jouer avec les vues. Nous allons pouvoir passer au composant `Backbone.Router`. Nous reviendrons ensuite sur la sécurisation de notre application dans le chapitre sur l’organisation du code.

## 9 Le Routeur

*Sommaire*

- *Modifier (un peu) la vue principale*
- *Ajouter un template*
- *“Maîtriser” les urls*

*Parmi les composants principaux de Backbone, il y a le Routeur (`Backbone.Router`), qui n'est pas obligatoire pour construire une application Backbone, mais qui néanmoins est très pratique. Son rôle principal est de déclencher des actions en fonction de l'url saisie dans votre navigateur, ou des liens cliqués.*

Dans une SPA (Single Page Application), la barre d'url est un composant à part entière de votre application. C'est une partie à laquelle l'utilisateur accède facilement, vous ne pouvez donc pas l'ignorer, et votre application devra pouvoir réagir en fonction des actions de l'utilisateur. Nous allons donc mettre en œuvre `Backbone.Router` pour pouvoir « réagir » aux changements d'url. Nous allons transformer notre vue d'affichages de posts, pour qu'elle n'affiche plus le détail des messages, mais à la place un lien qui lorsqu'il sera clické affichera le détail du message. Nous verrons qu'ensuite cette méthode nous permettra d'accéder directement à partir de la barre de l'url à des fonctionnalités de notre application.

### 9.1 Modifions notre vue

Dans la page index.html, modifions le template « `posts_list_template` » afin qu'il n'affiche plus le contenu du post `({{message}})`. A la place nous ajoutons un lien dont l'url sera post-fixée de l'id du post `(#post/{{id}})` :

```
<script type="text/template" id="posts_list_template">

  {{#posts}}
    <h2>{{title}}</h2>
    <b>par : {{author}}</b> le : {{date}}
    <a href="#post/{{id}}">Lire ...</a>

  {{/posts}}

</script>
```

Ensuite nous ajoutons un nouveau template (pour voir le détail du post) qui sera utilisé lorsque nous cliquerons sur le lien « Lire ... », avec un lien `(#/)` qui déclenchera l'affichage de l'ensemble de la liste des posts :

```
<script type="text/template" id="post_details_template">
  <h2>{{post.title}}</h2>
  <b>par : {{post.author}}</b> le : {{post.date}}
  <p>{{post.message}}</p>

  <a href="#">Tous les messages</a>
</script>
```

Nous créons ensuite un nouvel objet de type `Backbone.View` qui sera « chargé » d'afficher le détail du message du post, à partir du nouveau template (`#post_details_template`), en lieu et place de la liste des posts (`#posts_list`) :

```
window.PostView = Backbone.View.extend({
  el : $("#posts_list"),
  initialize : function () {
    this.template = $("#post_details_template").html();
  },
  render : function (post) {
    var renderedContent = Mustache.to_html(this.template, {post : post.toJSON()});
    this.$el.html(renderedContent);
  }
});

window.postView = new PostView();
```

## 9.2 Crédit du routeur

Nous pouvons maintenant créer notre routeur. La propriété importante du routeur est `routes`. Dans notre exemple (juste en dessous), je lui ai affecté trois routes :

- `post/:id_post` : lorsque le click sur un lien de type `<a href="#post/{{id}}>Lire ...</a>` la méthode `displayPost` du routeur sera appelée avec l'id du post en paramètre
- `hello`, qui appellera la méthode `hello` si par exemple on saisit `http://localhost:3000/#hello` dans la barre d'url du navigateur (notez le “#”, nous y reviendrons plus tard)
- et enfin `*path` qui appellera la méthode `root` pour toute autre url comme `#/`, `/`, ...

Le code qui sert à récupérer la liste des posts en provenance du serveur est déplacé dans la méthode `root` du routeur :

*Le routeur de notre application de blog :*

```
window.RoutesManager = Backbone.Router.extend({
  routes : {
    "post/:id_post" : "displayPost",
    "hello" : "hello",
    "*path" : "root"
  },
  root : function () {
    blogPosts.all().fetch({
      success:function(result){
        //ça marche !!!
      }
    });
  }
});
```

```

hello : function () {
    $(".hero-unit > h1").html("Hello World !!!");
},

displayPost : function (id_post) {

    var tmp = new Post({id:id_post});

    tmp.fetch({
        success : function(result) {
            postView.render(result);
        }
    });
}

window.router = new RoutesManager();

Backbone.history.start();

//TODO: faire un § sur Backbone.history.start({pushState: true});

```

Sauvegardez le tout, rafraîchissez la page, et testez :

**Les 3 derniers :**

- Mais où sont les contrôleurs ?
- Les Routes
- Backbone ???

**Login :**

email  
password

Login Logoff  
??? John Doe

**Mais où sont les contrôleurs ?**  
par : bob le : 2012-08-15T08:13:36.741Z [Lire ...](#)

**Les Routes**  
par : bob le : 2012-08-15T08:13:35.741Z [Lire ...](#)

**Backbone ???**  
par : bob le : 2012-08-15T08:13:34.741Z [Lire ...](#)

**Les Modèles**  
par : sam le : 2012-08-15T08:13:33.741Z [Lire ...](#)

Si vous cliquez sur un des liens « Lire ... », la liste des posts disparaît au profit du message relatif au post sélectionné :

The screenshot shows a web browser window with the title 'Backbone'. The address bar displays 'localhost:3000/#post/33kcjmrg'. The main content area has a dark header 'Mon Blog'. Below it, a large box contains the text 'Backbone rocks !!!'. To the left, there's a sidebar with a 'Les 3 derniers:' list and a 'Login:' form. The list includes 'Mais où sont les contrôleurs?', 'Les Routes', and 'Backbone ???'. The login form has fields for 'email' and 'password', and buttons for 'Login' and 'Logoff'. A user '??? John Doe' is logged in. Below the login form, there's a link 'Tous les messages'.

**Remarque IMPORTANTE :** Du coup il est maintenant possible de bookmarker les urls des posts pour pointer directement dessus.

Maintenant, essayer aussi de taper l'url directement dans la barre d'url : <http://localhost:3000/#hello>. Et là, le titre de notre blog change. Donc l'url peut bien déclencher directement des actions javascript. Vous pouvez donc réagir, prévenir, ... toute modification de l'url (comme le retour à la page précédente) pour déclencher l'action nécessaire (par exemple la sauvegarde des données en cours).

The screenshot shows a web browser window with the title 'Backbone'. The address bar displays 'localhost:3000/#hello'. The main content area has a dark header 'Mon Blog'. Below it, a large box contains the text 'Hello World !!!'. To the left, there's a sidebar with a 'Les 3 derniers:' list and a 'Login:' form. The list includes 'Mais où sont les contrôleurs?', 'Les Routes', and 'Backbone ???'. The login form has fields for 'email' and 'password', and buttons for 'Login' and 'Logoff'. A user '??? John Doe' is logged in. Below the login form, there are three links: 'Mais où sont les contrôleurs ?', 'Les Routes', and 'Backbone ???', each with a timestamp 'par : bob le : 2012-08-15T08:13:36.741Z' and a 'Lire ...' link.

Voilà. C'est un peu court pour le Routeur, mais cela devrait suffire pour le moment. Maintenant il est temps d'organiser notre code "comme les vrais" avant que notre projet devienne un sac de nouille.

# 10 Organiser son code

*Sommaire*

- *Namespaces & Modules, à l'ancienne*
- *“Loader” javascript, comme les vrais*

*Notre application (côté client) tient dans une seule page. Elle mélange du code HTML et du code Javascript, et cela dans un seul fichier commence à devenir difficilement lisible, difficilement modifiable et donc difficilement maintenable. J'aimerais ajouter la possibilité d'ajouter ou de modifier des posts, mais ... Faisons d'abord le ménage et rangeons notre code..*

Il y a de nombreuses querelles de chapelle autour du sujet de l'organisation du code (des façons de le faire), et je vous avoue que je n'ai pas encore fait complètement mon choix, mais l'essentiel est de produire quelque chose qui fonctionne (correctement) et que vous pourrez facilement faire évoluer. Je vais donc vous présenter 2 méthodes, la méthode « à l'ancienne » qui a le mérite de fonctionner, d'être rapide, et la méthode « hype » pour les champions qui est intéressante à connaître tout particulièrement dans le cadre de gros projets.

**Remarque :** pour la méthode “hype”, j'utilise le(s) outil(s) (yepnope) que je préfère, il est tout à fait possible de suivre le principe décrit avec d'autres comme require.js et d'autres ... Je n'ai pas la science infuse, je me sens plus à l'aise avec YepNope ... c'est tout, et tant que ça marche ;) ...

## 10.1 “A l'ancienne”

### 10.1.1 Namespace

Créer une application Backbone, c'est écrire des modèles, des vues, des templates, etc. ... Et une des bonnes pratiques pour parvenir à garder ceci bien organisé et d'utiliser le **namespacing** (comme en .Net, Java, ...) de la façon suivante :

*Namespace Blog :*

```
var Blog = {
  Models : {},
  Collections : {},
  Views : {},
  Router : {}
}
```

Vous enregistrez ceci dans un fichier **Blog.js**, que vous pensez à référencer dans votre page html :

```
<script src="Blog.js"></script>
```

Ainsi par la suite vous pourrez déclarer et faire référence à vos composants de la manière suivante :

```

Blog.Models.Post = Backbone.Model.extend({
    ...
});

Blog.Collections.Posts = Backbone.Collection.extend({
    ...
});

Blog.Views.PostForm = Backbone.View.extend({
    ...
});

//etc...

```

Puis les utiliser comme ceci :

```

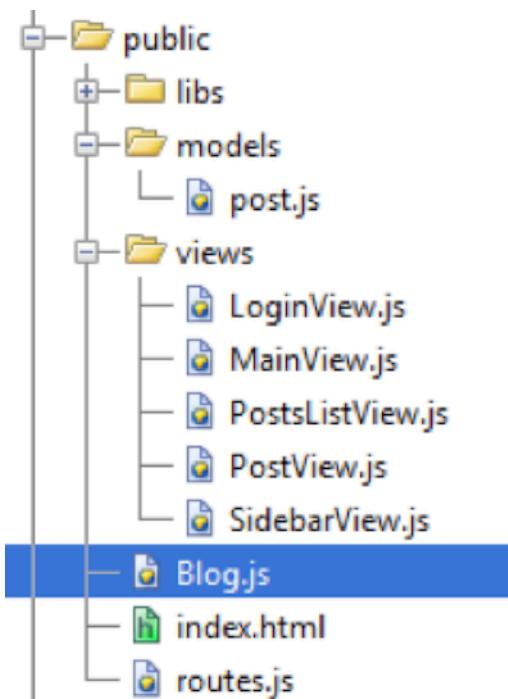
var myPost = new Blog.Models.Post();
var posts = new Blog.Collections.Posts();
var postForm = new Blog.Views.PostForm();

```

Du coup, à la lecture du code, on voit tout de suite que `myPost` est un modèle, `posts` une collection et `postForm` une vue.

Cela va donc permettre de créer d'autres fichiers javascript spécifiques à chacun de vos composants, par exemple un fichier `post.js` avec le code de votre modèle `post` (*en général j'y ajoute aussi la collection correspondante*), puis autant de fichiers javascript que de vues. Cela va augmenter le nombre de vos fichiers, mais à chaque modification vous n'aurez à vous concentrer que sur une seule portion de code.

Puis vous pouvez aussi organiser vos fichiers javascript dans des répertoires. Voici comment je procède :



Je crée un répertoire `libs/vendors` pour tous les scripts « qui ne sont pas de moi » (jquery, backbone, etc. ...), puis je range mes modèles dans un répertoire `models`, mes vues dans un répertoire `views`. Au même endroit que ma page `index.html`, je positionne le fichier `routes.js`(mon routeur) ainsi

que le fichier `Blog.js` qui contient mes « namespaces », et enfin je déclare tout ceci dans ma page `index.html`:

```
<!-- === Frameworks === -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>
<script src="libs/vendors/mustache.js"></script>

<!-- === code applicatif === -->

<script src="Blog.js"></script>
<script src="models/Post.js" ></script> <!-- Models & Collection -->

<!-- Backbone Views -->
<script src="views/SidebarView.js"></script>
<script src="views/PostsListView.js"></script>
<script src="views/MainView.js"></script>
<script src="views/LoginView.js"></script>
<script src="views/PostView.js"></script>

<script src="routes.js"></script>
```

Pour ensuite écrire mon code javascript de « lancement », clairement simplifié par rapport à ce que nous avons fait jusqu'ici :

```
<script>
$(function (){

    window.blogPosts = new Blog.Collections.Posts();

    window.mainView = new Blog.Views.MainView({collection : blogPosts});

    /*===== Authentification =====*/
    window.loginView = new Blog.Views.LoginView();
    /*===== Fin authentification =====*/

    window.postView = new Blog.Views.PostView();

    window.router = new Blog.Router.RoutesManager({collection:blogPosts});
    Backbone.history.start();

});
```

**Remarque :** Vous n'êtes pas obligés de faire comme moi, adaptez selon vos goûts ou les normes imposées sur les projets. Et n'hésitez pas à me contacter pour me donner des astuces pour améliorer mon organisation de code.

### 10.1.2 Modules

J'aime bien combiner la notion de module à la notion de namespace, ce qui permet d'associer à notre namespace une notion de variable ou de fonctions privées, mais aussi de créer un système de plugin. Je vous montre avec le code, ce sera plus « parlant » :

Avant nous avions donc ceci :

*Namespace Blog* :

```
var Blog = {
  Models : {},
  Collections : {},
  Views : {},
  Router : {}
}
```

Que nous allons changer par ceci :

*Module+Namespace* :

```
var Blog = (function () {
  var blog = {};

  blog.Models = {};
  blog.Collections = {};
  blog.Views = {};
  blog.Router = {};

  return blog;
}());
```

Cela signifie que seul la variable `blog` sera exposée par l'entremise de la variable `Blog`. Tout ce qui est entre `(function () { et })()`; sera exécuté. A l'intérieur de cette closure vous pouvez coder des variables et des méthodes privées.

Ensuite vous allez pouvoir déclarer des « plug-ins » à votre module de la manière suivante (par exemple):

```
var Blog = (function (blog) {

  blog.Models.Post = Backbone.Model.extend({
    urlRoot :"/blogposts"
  });

  blog.Collections.Posts = Backbone.Collection.extend({
    model : blog.Models.Post,
    all : function () {
      this.url = "/blogposts";
      return this;
    },
  });
});
```

```

    query : function (query) {
        this.url = "/blogposts/query/" + query;
        return this;
    }

});

return blog;
}(Blog));

```

### 10.1.3 Au final, nous aurons ...

Avec les principes décrits plus haut, nous allons donc pouvoir “découper” notre code afin de bien tout ordonner, et nous obtiendrons les fichiers suivants :

*Blog.js* :

```

var Blog = (function () {
    var blog = {};

    blog.Models = {};
    blog.Collections = {};
    blog.Views = {};
    blog.Router = {};

    return blog;
}());

```

*routes.js* :

```

var Blog = (function (blog) {

    blog.Router.RoutesManager = Backbone.Router.extend({
        initialize : function(args) {
            this.collection = args.collection;
        },
        routes : {
            "post/:id_post" : "displayPost",
            "hello" : "hello",
            "*path" : "root"
        },
        root : function () {
            this.collection.all().fetch({
                success:function(result){
                    //ça marche !!!
                }
            });
        },
        hello : function () {

```

```

        $(".hero-unit > h1").html("Hello World !!!");
    },

    displayPost : function (id_post) {

        var tmp = new blog.Models.Post({id:id_post});

        tmp.fetch({
            success : function(result) {
                postView.render(result);
            }
        });
    }

    return blog;
}(Blog));

```

*models/post.js :*

```

var Blog = (function (blog) {

    blog.Models.Post = Backbone.Model.extend({
        urlRoot :"/blogposts"
    });

    blog.Collections.Posts = Backbone.Collection.extend({
        model : blog.Models.Post,
        all : function () {
            this.url = "/blogposts";
            return this;
        },
        query : function (query) {
            this.url = "/blogposts/query/" +query;
            return this;
        }
    });

    return blog;
}(Blog));

```

*views/SidebarView.js :*

```

var Blog = (function (blog) {

    blog.Views.SidebarView = Backbone.View.extend({
        el : $("#blog_sidebar"),
        initialize : function () {
            this.template = $("#blog_sidebar_template").html();

```

```

},
render : function () {
    var renderedContent = Mustache.to_html(this.template,
        {posts : this.collection.toJSON()});
};

this.$el.html(renderedContent);
}
});

return blog;
}(Blog));

```

*views/PostsListViews.js :*

```

var Blog = (function (blog) {

blog.Views.PostsListView = Backbone.View.extend({
    el : $("#posts_list"),
    initialize : function () {
        this.template = $("#posts_list_template").html();
    },
    render : function () {
        var renderedContent = Mustache.to_html(this.template,
            {posts : this.collection.toJSON()});
    };

    this.$el.html(renderedContent);
}
});

return blog;
}(Blog));

```

*views/MainView.js :*

```

var Blog = (function (blog) {

blog.Views.MainView = Backbone.View.extend({
    initialize : function () {

        this.collection.comparator = function (model) {
            return -(new Date(model.get("date")).getTime());
        }

        _.bindAll(this, 'render');
        this.collection.bind('reset', this.render);
        this.collection.bind('change', this.render);
        this.collection.bind('add', this.render);
    }
});

```

```

    this.collection.bind('remove', this.render);

    this.sidebarView = new blog.Views_sidebarView();
    this.postsListView = new blog.Views_PostsListView({
        collection : this.collection
    });

},
render : function () {

    //this.collection.models = this.collection.models.reverse();
    this.sidebarView.collection =
        new blog.Collections_Posts(this.collection.first(3));
    this.sidebarView.render();
    this.postsListView.render();
}
});

return blog;
}(Blog));

```

*views/LoginView.js :*

```

var Blog = (function (blog) {

    blog.Views.LoginView = Backbone.View.extend({
        el : $("#blog_login_form"),

        initialize : function () {
            var that = this;
            this.template = $("#blog_login_form_template").html();

            //on vérifie si pas déjà authentifié
            $.ajax({type:"GET", url:"/alreadyauthenticated",
                error:function(err){ console.log(err); },
                success:function(dataFromServer) {

                    if(dataFromServer.firstName) {
                        that.render("Bienvenue",dataFromServer);
                    } else {
                        that.render("???", {firstName:"John", lastName:"Doe"});
                    }
                }
            })
        },
        render : function (message, user) {

            var renderedContent = Mustache.to_html(this.template, {

```

```

        message : message,
        firstName : user ? user.firstName : """",
        lastName : user ? user.lastName : """
    });
    this.$el.html(renderedContent);
},
events : {
    "click .btn-primary" : "onClickBtnLogin",
    "click .btn-inverse" : "onClickBtnLogoff"
},
onClickBtnLogin : function (domEvent) {

    var fields = $("#blog_login_form :input")
    ,   that = this;

    $.ajax({
        type:"POST",
        url:"/authenticate",
        data : { email : fields[0].value, password : fields[1].value } ,
        dataType : 'json',
        error:function(err){ console.log(err); },
        success:function(dataFromServer) {

            if(dataFromServer.infos) {
                that.render(dataFromServer.infos);
            } else {
                if(dataFromServer.error) {
                    that.render(dataFromServer.error);
                } else {
                    that.render("Bienvenue",dataFromServer);
                }
            }
        }
    });
},
onClickBtnLogoff : function() {

    var that = this;
    $.ajax({type:"GET", url:"/logoff",
        error:function(err){ console.log(err); },
        success:function(dataFromServer) {
            console.log(dataFromServer);
            that.render("???",{firstName:"John", lastName:"Doe"});
        }
    })
}
});

});

```

```
    return blog;
}(Blog));
```

*views/PostView.js :*

```
var Blog = (function (blog) {

    blog.Views.PostView = Backbone.View.extend({
        el : $("#posts_list"),
        initialize : function () {
            this.template = $("#post_details_template").html();
        },
        render : function (post) {
            var renderedContent = Mustache.to_html(this.template,
                {post : post.toJSON()});
        };

        this.$el.html(renderedContent);
    }
});

return blog;
})(Blog));
```

... Sauvegardez tout ça et essayez, normalement cela devrait fonctionner et vous verrez qu'à l'usage, le code en devient plus lisible. Mais passons donc à la 2ème méthode.

## 10.2 Méthode “hype”, comme les vrais

Plus les fichiers javascript se multiplient, plus la gestion des `<script src="...">` devient pénible, sans compter que dans certains cas il est nécessaire de gérer l'ordre d'inclusion, que l'on aimerait pouvoir déclencher un traitement uniquement lorsque l'on est sûr que notre script est chargé, etc. ...

Pour répondre à ces types de problématiques, il existe ce que l'on appelle des “loaders” (chargeurs) de script, tels :

- Require.js (probablement le plus connu et le plus utilisé) <http://requirejs.org/>
- Head.js <http://headjs.com/>
- YepNope <http://yepnopejs.com/> d'Alex Sexton
- Etc. ...

Il y a beaucoup de débats autour des « javascript resources loaders », « est-ce bien ou mal ? » « Cela ralentit le chargement de la page web », « c'est génial il faut généraliser son utilisation », ... Mon propos n'est pas de participer au débat mais de vous montrer « rapidement » de quelle façon on peut les utiliser. (Cependant si votre application est simple, cela ne vaut pas la peine d'en utiliser, si ce n'est à titre éducatif ou pour le plaisir).

**Remarque :** si vous souhaitez creuser le sujet je vous engage à lire « Non-onload-blocking async JS » de Stoyan Stefanov : <http://www.phpied.com/non-onload-blocking-async-js/>.

En ce qui nous concerne, j'ai choisi **YepNope** parce que nettement plus simple (plus léger aussi) que Require.js et développé par Alex Sexton, ce qui est un gage de qualité (excellent développeur javascript et qui prend le temps de répondre à vos questions, ce qui n'est pas négligeable).

### 10.2.1 Préparation

Commencez par télécharger la dernière version de YepNope ici :

- <https://github.com/SlexAxton/yepnope.js/archives/master>

Puis, dézipper et copier ensuite le fichier `yepnope.js` ou sa version minifiée (dans mon cas j'ai utilisé la version 1.5.4) dans le répertoire `public/libs/vendors` de votre application.

Nous allons maintenant supprimer toutes les références de script que nous avions dans la page `index.html` :

```
<!-- === Frameworks === -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>
<script src="libs/vendors/mustache.js"></script>

<!-- === code applicatif === -->

<script src="Blog.js"></script>
<script src="models/Post.js"></script> <!-- Models & Collection -->

<!-- Backbone Views -->
<script src="views/SidebarView.js"></script>
<script src="views/PostsListView.js"></script>
<script src="views/MainView.js"></script>
<script src="views/LoginView.js"></script>
<script src="views/PostView.js"></script>

<script src="routes.js"></script>
```

Et nous écrivons ceci à la place :

```
<script src="libs/vendors/yepnope.1.5.4-min.js"></script>
<script src="main.js"></script>
```

Et c'est donc dans le script `main.js` que nous allons procéder au chargement de nos différents scripts de la manière suivante :

1ère utilisation de `yepnope` :

```

yepnope({
  load: {
    jquery      : 'libs/vendors/jquery-1.7.2.js',
    underscore  : 'libs/vendors/underscore.js',
    backbone    : 'libs/vendors/backbone.js',
    mustache   : 'libs/vendors/mustache.js',

    //NameSpace
    blog        : 'Blog.js',

    //Models
    posts       : 'models/post.js',

    //Controllers
    sidebarview : 'views/SidebarView.js',
    postslistviews : 'views/PostsListView.js',
    mainview    : 'views/MainView.js',
    loginview   : 'views/LoginView.js',
    postview    : 'views/PostView.js',

    //Routes
    routes      : 'routes.js'

  },
  callback : {
    "routes" : function () {
      console.log("routes loaded ...");
    }
  },
  complete : function () {
    //...
  }
});

```

Vous l'aurez compris, le paramètre `load` sert à définir les scripts à charger. Vous notez aussi que l'on peut donner un alias à chacun des scripts (sinon YepNope le fera automatiquement à partir du nom du fichier javascript), alias que l'on peut ensuite utiliser dans le paramètre `callback` pour déclencher un traitement une fois que le script est inclus dans la page (dans notre exemple c'est au moment de l'inclusion du fichier `routes.js`).

Et enfin, il y a aussi le paramètre `complete` qui permet de lancer un traitement une fois tous les scripts inclus.

Nous allons donc déplacer le javascript restant dans notre page à l'intérieur de `complete`, pour finalement obtenir ceci :

*Code définitif :*

```

yepnope({
  load: {
    jquery      : 'libs/vendors/jquery-1.7.2.js',

```

```
underscore          : 'libs/vendors/underscore.js',
backbone           : 'libs/vendors/backbone.js',
mustache           : 'libs/vendors/mustache.js',

//NameSpace
blog               : 'Blog.js',

//Models
posts              : 'models/post.js',

//Controllers
sidebarview        : 'views/SidebarView.js',
postslistviews     : 'views/PostsListView.js',
mainview            : 'views/MainView.js',
loginview           : 'views/LoginView.js',
postview            : 'views/PostView.js',

//Routes
routes             : 'routes.js'

},

callback : {
  "routes" : function () {
    console.log("routes loaded ...");
  }
},
complete : function () {
  $(function (){

    console.log("Lauching application ...");

    window.blogPosts = new Blog.Collections.Posts();

    window.mainView = new Blog.Views.MainView({collection : blogPosts});

    /*===== Authentification =====*/
    window.loginView = new Blog.Views.LoginView();
    /*===== Fin authentication =====*/

    window.postView = new Blog.Views.PostView();

    window.router = new Blog.Router.RoutesManager({collection:blogPosts});

    Backbone.history.start();

  });
}
});
```

Et voilà ! Vous disposez maintenant d'un code structuré, d'un outil de chargement de script facile à utiliser et modifier : désactivation ou changement provisoire de librairie pour tests par exemple mais aussi chargement conditionnel de script en fonction du contexte, ... Je ne vous ai dévoilé qu'une infime partie de YepNope qui en dépit de sa taille est très puissant. Lisez la documentation, vous verrez ...

Maintenant que notre projet est "propre", nous allons dans le chapitre suivant en profiter pour sécuriser un peu plus notre application.

## 11 Securisation

*Sommaire*

- *Un (tout) petit écran d'administration*
- *Une "vraie" sécurisation des routes côté serveur*

*Profitons-en pour terminer (presque) notre application et la sécuriser. Je dis "presque", car il y aura toujours des fonctionnalités à ajouter, des contrôles de saisie à faire, des codes à optimiser (vous verrez dans ce qui va suivre que ma façon d'accéder au DOM n'est pas forcément la plus élégante à défaut d'être la plus lisible), la gestion des commentaires, refaire l'authentification ...*

Je souhaite cependant pouvoir modifier et ajouter des messages (post). C'est un exercice pour démontrer l'utilité de l'organisation du code en modules pour mieux s'y retrouver.

### 11.1 Il nous faut un "écran d'administration"

Je retourne donc dans le code html de ma page `index.html`. Ajoutons ceci juste après le "grand titre" de notre blog :

*Template de l'écran d'administration :*

```
<div class="span9">
    <div class="hero-unit">
        <h1>Backbone rocks !!!</h1>
    </div>

    <!-- Admin --->
    <script type="text/template" id="admin_template">
        <hr>
        <h3>Administration du Blog</h3>
        <hr>
        <select id="post_choice">{{#posts}}
            <option value="{{id}}">{{title}}</option>
        {{/posts}}</select>
        <a href="#" id="btn_update" class="btn btn-primary">Modifier Post</a>
        <a href="#" id="btn_create" class="btn btn-primary">Ré-initialiser / Ajouter Post</a>
        <hr>
        id : <span name="id"></span><br>
```

```

<input name="author" type="text" placeholder="author"/><br>
<input name="title" type="text" placeholder="title"/><br>
<textarea name="message" placeholder="message"></textarea><br>
<a href="#" id="btn_send" class="btn btn-primary">Sauvegarder</a>
<hr>
</script>

<div id="admin">

</div>
<!-- End of Admin -->

```

J'aurais donc un écran qui va ressembler à ceci :

Pour gérer cet écran je vais avoir besoin d'un objet de type `Backbone.View`, que nous allons créer dans le répertoire `/views` sous le nom d'`AdminView.js`. Avant de "coder" `AdminView`, nous allons le déclarer et l'instancier dans `main.js` :

`main.js` :

```
yepnope({
  load: {
    jquery:           'libs/vendors/jquery-1.7.2.js',
    underscore:       'libs/vendors/underscore.js',
    backbone:         'libs/vendors/backbone.js',
    mustache:         'libs/vendors/mustache.js',
  }
});
```

```

//NameSpace
blog : 'Blog.js',

//Models
posts : 'models/post.js',

//Controllers
sidebarview : 'views/SidebarView.js',
postslistviews : 'views/PostsListView.js',
mainview : 'views/MainView.js',
loginview : 'views/LoginView.js',
postview : 'views/PostView.js',

//--- ADMINVIEW ---
adminview : 'views/AdminView.js',

//Routes
routes : 'routes.js'
},

callback : {
  "routes" : function () {
    console.log("routes loaded ...");
  }
},
complete : function () {
  $(function () {
    console.log("Launching application ...");

    window.blogPosts = new Blog.Collections.Posts();

    window.mainView = new Blog.Views.MainView({collection : blogPosts});

    /*===== Admin =====*/
    window.adminView = new Blog.Views.AdminView({collection : blogPosts});
    /*===== Fin Admin =====*/

    /*===== Authentification =====*/
    window.loginView = new Blog.Views.LoginView({adminView : adminView});
    /*===== Fin authentification =====*/

    window.postView = new Blog.Views.PostView();

    window.router = new Blog.Router.RoutesManager({collection:blogPosts});
    //Backbone.history.start({pushState: true});
    Backbone.history.start();
  });
}
}

```

```
});
```

**Remarque :** Je passe la vue adminView à la vue loginView pour que cette dernière puisse déclencher le rendue de la première si nécessaire.

### 11.1.1 Création d'AdminView

Alors il n'y a pas grand chose à expliquer (c'est dans le code)

*AdminView.js :*

```
var Blog = (function (blog) {

    blog.Views.AdminView = Backbone.View.extend({
        el : $("#admin"),
        initialize : function () {
            this.template = $("#admin_template").html();
            //je prévois de trier ma collection
            this.collection.comparator = function (model) {
                return -(new Date(model.get("date")).getTime());
            }
        },
        render : function () {
            var renderedContent = Mustache.to_html(this.template,
                {posts : this.collection.toJSON() } );
            this.$el.html(renderedContent);
        },
        events : {
            "click #btn_update" : "onClickBtnUpdate",
            "click #btn_create" : "onClickBtnCreate",
            "click #btn_send" : "sendPost"
        },
        onClickBtnUpdate : function () {
            var selectedId = $("#post_choice").val()
            ,   post = this.collection.get(selectedId);

            //Je récupère les informations du post et les affiche
            $("#admin > [name='id']").html(post.get("id"));
            $("#admin > [name='author']").val(post.get("author"));
            $("#admin > [name='title']").val(post.get("title"));
            $("#admin > [name='message']").val(post.get("message"));

        },
        onClickBtnCreate : function () {
            //je ré-initialise les zones de saisie
            $("#admin > [name='id']").html("");
            $("#admin > [name='author']").val("");
            $("#admin > [name='title']").val("");
            $("#admin > [name='message']").val("");
        }
    });
});
```

```

},
sendPost : function () { //Sauvegarde
    var that = this //pour conserver le contexte
    , id = $("#admin > [name='id']").html()
    , post;

    if(id==="") { //si l'id est vide c'est une création
        post = new Blog.Models.Post();
    } else { //l'id n'est pas vide c'est une mise à jour
        post = new Blog.Models.Post({
            id:$("#admin > [name='id']").html()
        });
    }

    post.save({
        author : $("#admin > [name='author']").val(),
        title : $("#admin > [name='title']").val(),
        message : $("#admin > [name='message']").val(),
        date : new Date()
    },{
        success:function () {
            //Si la transaction côté serveur a fonctionné

            //je recharge ma collection
            that.collection.fetch({
                success:function() {
                    //mise à jour de la vue admin
                    that.render();
                    //La vue principale se re-mettra à jour
                    //automatiquement, car elle est "abonnée"
                    //aux changement de la collection
                }
            });
        },
        error : function () {}
    });
}
});

return blog;
}(Blog));

```

**Remarque :** je n'abonne pas la vue aux changement de la collection pour pouvoir maîtriser le moment où je déclenche la méthode render() de celle-ci.

### 11.1.2 Modification de LoginView

Nous allons modifier LoginView.js pour prendre en compte l'ajout de notre fonctionnalité d'administration. Tout d'abord modifications le template associé :

*Template du formulaire de login :*

```

<!-- /*===== Formulaire d'authentification =====*/ -->
<script type="text/template" id="blog_login_form_template">
  <h3>Login :</h3>
  <input name="email" type="text" placeholder="email"/><br>
  <input name="password" type="password" placeholder="password"/><br>
  <a href="#" class="btn btn-primary">Login</a>
  <a href="#" class="btn btn-inverse">Logoff</a><br>
  <b>{{message}} {{firstName}} {{lastName}} </b>
  <br><a id="adminbtn" href="#">{{adminLinkLabel}}</a>

</script>
<form class="container" id="blog_login_form">

</form>
<!-- /*===== Fin du formulaire=====*/ -->

```

Puis modifions le code de LoginView.js :

*LoginView.js* :

```

var Blog = (function (blog) {

  blog.Views.LoginView = Backbone.View.extend({
    el : $("#blog_login_form"),

    initialize : function (args) {
      var that = this;

      this.adminView = args.adminView;

      this.template = $("#blog_login_form_template").html();

      //on vérifie si pas déjà authentifié
      $.ajax({type:"GET", url:"/alreadyauthenticated",
        error:function(err){ console.log(err); },
        success:function(dataFromServer) {

          if(dataFromServer.firstName) {

            that.render("Bienvenue",dataFromServer);

          } else {
            that.render("????",{firstName:"John", lastName:"Doe"});
          }
        }
      });
    },
    render : function (message, user) {

      var renderedContent = Mustache.to_html(this.template, {
        message : message,

```

```

        firstName : user ? user.firstName : '',
        lastName : user ? user.lastName : '',
        adminLinkLabel : user ? user.isAdmin ? "Administration" : "" : ""
    });
    this.$el.html(renderedContent);
},
events : {
    "click .btn-primary" : "onClickBtnLogin",
    "click .btn-inverse" : "onClickBtnLogoff",
    "click #adminbtn" : "displayAdminPanel"
},
displayAdminPanel : function(){
    //this.adminView.render();
},
onClickBtnLogin : function (domEvent) {
//code non affiché pour des raisons de mise en page
//ne pas modifier
},
onClickBtnLogoff : function() {
//code non affiché pour des raisons de mise en page
//ne pas modifier
}

});
return blog;
}(Blog));

```

Maintenant que nous avons fait nos modifications côté client, allons sécuriser les actions côté serveur.

## 11.2 Sécurisation côté serveur

Vous pouvez tester les modifications dès maintenant, cela va fonctionner, mais il est de bon ton de sécuriser côté serveur les actions de création, modification, suppression, etc. ...

**Remarque :** La gestion des session avec express.js ainsi que la sécurisation des routes peut être prise en compte de manière plus « professionnel ». Gardez à l'esprit que les exemples donnés le sont à titre « didactique » et que certains points sont perfectibles.

Nous allons retourner dans le code de `app.js` (vous vous souvenez : notre application express.js) et créer une fonction qui permet de vérifier si l'utilisateur est connecté et est administrateur et cette fonction servira à sécuriser les routes :

### 11.2.1 Sécurisation des routes

Notre fonction vérifie si l'utilisateur connecté est un administrateur, si c'est le cas, elle passe la main à la fonction “callback” que j'ai appelée `next` sinon elle génère une erreur.

Fonction `haveToBeAdmin` :

```
haveToBeAdmin = function (req, res, next) {
    console.log("NEEDS ADMIN RIGHTS");
    if(findUserBySession(req.sessionID)){
        if (findUserBySession(req.sessionID).isAdmin==true) {
            next();
        } else {
            throw "You have to be administrator";
        }
    } else {
        throw "You have to be connected";
    }
}
```

**Remarque** : Si la condition est vérifiée, la fonction `next()` est appelée, elle correspond au traitement de la route sécurisée.

Et sécurisons ensuite nos routes de la manière suivante :

*Ajout* :

```
app.post('/blogposts', [haveToBeAdmin], function(req, res, next){
    console.log("POST CREATE ", req.body);

    var d = new Date(), model = req.body;
    model.saveDate = (d.valueOf());

    posts.save(null,model, function (err, key){
        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        } else {
            model.id = key;
            res.json(model);
        }
    });
});
```

**Remarque “au passage”** : vous remarquez le passage du paramètre `[haveToBeAdmin]`, le contenu de `app.post` ne sera exécuté que si `haveToBeAdmin` “déclenche” `next`.

De la même façon pour la mise à jour et la suppression :

*Mise à jour* :

```
app.put('/blogposts/:id', [haveToBeAdmin],function(req, res, next){
    console.log("PUT UPDATE", req.body, req.params.id);
```

```

var d = new Date(), model = req.body;
model.saveDate = (d.valueOf());

posts.save(req.params.id, model, function (err, key){
    if(err) {
        console.log("Erreur : ",err);
        res.json(err);
    } else {
        res.json(model);
    }
});

});
}
);

```

*Suppression :*

```

app.delete('/blogposts/:id', [haveToBeAdmin], function(req, res, next){
    console.log("DELETE : /delete/" +req.params.id);

    posts.remove(req.params.id, function(err){
        if(err) {
            console.log("Erreur : ",err);
            res.json(err);
        } else {
            //petit correctif de contournement (bug ds nStore) :
            //ré-ouvrir la base lorsque la suppression a été faite
            posts = nStore.new("blog.db", function() {
                res.json(req.params.id);
                //Le modèle est vide si on ne trouve rien
            });
        }
    });
});

```

### 11.2.2 Eh bien testons, maintenant

- Relancez l'application côté serveur.
- Ouvrez le navigateur et connectez vous au blog sans vous authentifier.
- Ouvrez la console du navigateur et saisissez ceci : adminView.render().

Cela va faire apparaître l'écran d'administration alors que vous n'êtes pas administrateur (vous faites une tentativve de hacking sur vous même) !

Sélectionnez un Post dans la liste, et cliquez sur “modifer Post” pour charger le formulaire :

The screenshot shows a Backbone.js application interface. At the top, there's a header with 'Backbone' and a URL 'localhost:3000/#'. Below it is a navigation bar with 'Login' and 'Logoff' buttons, and a user info '??? John Doe'. The main area is titled 'Administration du Blog' with tabs 'Les Routes' and 'Modifier Post' (which is selected). A form contains fields for 'id : 33kclmrg' (with value 'bob'), 'Les Routes' (with value 'Cras mattis consectetur purus sit amet fermentum.'), and a 'Sauvegarder' button.

Below this is a title 'Mais où sont les contrôleur ?' followed by a developer tools console window. The console shows the following log entries:

```

routes loaded ...
Launching application ...
fetching collection > child
> adminView.render()
< undefined
>

```

On the right side of the console, file paths are listed:

```

main.js:29
main.js:34
routes.js:16

```

At the bottom of the developer tools window, there are tabs for Elements, Resources, Network, Sources, Timeline, Profiles, Audits, Console, and CoffeeConsole, along with a search bar and filter buttons for All, Errors, Warnings, and Logs.

Saissez quelques modifications, cliquez sur le bouton “Sauvegarder” . Vous obtiendrez un message d’erreur dans la console :

The developer tools console window shows the same log entries as before, but now includes an error message:

```

routes loaded ...
Launching application ...
fetching collection > child
> adminView.render()
< undefined
✖ ▶ PUT http://localhost:3000/blogpost/33kclmrg 500 (Internal Server Error)
jquery-1.7.2.js:8240
>

```

The error message 'PUT http://localhost:3000/blogpost/33kclmrg 500 (Internal Server Error)' is highlighted in red. At the bottom of the developer tools window, there are tabs for Elements, Resources, Network, Sources, Timeline, Profiles, Audits, Console, and CoffeeConsole, along with a search bar and filter buttons for All, Errors, Warnings, and Logs.

A vous de gérer les message côté client (avec le callback “error” lors du `save()`) Et côté serveur cela vous affiche que vous devez être connecté :

```
1. Default (node)
Default (node) Default (bash)

k33g-orgs-MacBook-Air:blogall k33g_org$ node app.js
Express app started on port 3000
GET (ALL) : /blogposts
NEEDS ADMIN RIGHTS
You have to be connected

```

Donc notre blog est bien “sécurisé”. Maintenant rafraîchissez votre page et connectez vous en tant qu’administrateur. Vous obtiendrez un lien “Administration” au niveau du message de bienvenue qui vous permettra de déclencher l'affichage de l'écran d'administration :

The screenshot shows a web browser window titled "Backbone". The address bar displays "localhost:3000/#". The main content area has a dark header with "Mon Blog". Below it, there's a sidebar with "Les 3 derniers :" and a list of links: "Mais où sont les contrôleurs ?", "Les Routes", and "Backbone ???". There's also a "Login :" section with "email" and "password" input fields, and "Login" and "Logoff" buttons. The main content area features a large, bold text "Backbone rocks !!!". At the top of this area, there are buttons for "Modifier Post" and "Ré-initialiser / Ajouter Post". Below this, there's a list of posts with fields for "id" (set to "1kr8obus") and "content" (containing "bobby", "Mais où sont les contrôleurs ?", and "Nulla vitae elit libero, a pharetra augue."). A "Sauvegarder" button is at the bottom of this list.

Faites quelques modifications, et sauvegardez les modifications :

Vous pouvez ensuite vérifier qu'elles ont bien été prises en compte :

Voilà. Notre application est “terminée”. Je sais il faut le dire vite ;) mais vous avez une base, que vous pouvez faire évoluer (il faudrait ajouter les commentaires par exemple) ou améliorer, ou mieux, cassez tout et faites votre propre application avec votre “propre style”.

Par contre, l'ouvrage n'est pas fini, j'ai encore 2,3 petites choses dont j'aimerais vous parler.

## 12 Backbone.sync()

//TODO: je n'ai encore rien "gratté" à ce sujet (viendra probablement en dernier)

## 13 Backbone <3 Coffeescript

*Sommaire*

- Coffeescript
- On s'outille
- Traduction

*Depuis longtemps, les développeurs "serveur" ont de nombreux a priori vis-à-vis de Javascript : un modèle objet "particulier" difficile à comprendre après des années de programmation orientée classes, un système d'héritage par prototype générant beaucoup d'effets de bord s'il n'est pas maîtrisé (sans compter la maintenabilité du code) et justement pas de classes en javascript, ce qui rend difficile l'organisation du code (toujours d'un point de vue approche "classique").*

L'arrivée de CoffeeScript tend aujourd'hui à gommer ces problématiques et tout particulièrement par l'introduction d'un système de classes qui prend en charge (pour/à la place du développeur) toutes les problématiques liées au modèle objet Javascript, garantissant ainsi la réduction de l'apparition de bugs dus à la méconnaissance de javascript. Gardez cependant une chose à l'esprit : Coffeescript, cela reste du Javascript, mais avec une manière différente de l'écrire, plus simple, plus efficace et (à mon avis) avec moins d'erreurs. Coffeescript, vous aidera aussi à comprendre et mieux écrire le Javascript.

### 13.1 Coffeescript qu'est-ce que c'est ?

Coffeescript est un langage de script qui ressemble beaucoup au Python. Il a été créé par Jeremy Ashkenas, vous savez, le brillant développeur aussi à l'origine de frameworks connus tels Backbone.js et Underscore.js. Coffeescript, c'est aussi un "Transpiler" Javascript. C'est-à-dire, qu'au lieu de compiler pour obtenir un binaire, on "compile" le code Coffeescript en Javascript directement exécutable dans un navigateur (ou côté serveur avec Node.js). Classiquement, le transpiler Coffeescript s'exécute sous Node.js, mais vous pouvez très bien l'utiliser en mode "run-time" et insérer du code Coffeescript directement (inline) dans vos pages HTML. C'est moins performant, mais cela peut être utile pour debugger. L'objet de ce chapitre n'est pas de vous apprendre Coffeescript, mais de vous le présenter rapidement et vous montrer à quoi pourrait ressembler le code de la partie cliente de notre blog traduite en Coffeescript. Vous trouverez les informations complémentaire ici : <http://coffeescript.org/>.

### 13.2 Un code plus lisible ?

Donc, selon moi (et d'autres), Coffeescript permet de simplifier le javascript, générer du javascript « propre » et apporte des évolutions de langage qui vont simplifier et rendre le code plus lisible. Nous allons tenter de voir ça par le biais de quelques exemples.

### 13.2.1 Les fonctions

Le terme `function` disparaît au profit d'une « flèche » ! `return` disparaît complètement !!! (par défaut c'est la dernière ligne qui fait office de return) et les parenthèses ne sont pas partout obligatoires (ce qui facilite l'écriture de DSL) ... Et plus de point-virgule.

*Fonction d'addition en javascript :*

```
function addition (a,b) {
    return a+b
}

//Utilisation :

var c = addition(45, 12)
```

Deviendra :

*Fonction d'addition en Coffeescript :*

```
addition = (a,b)->
    a+b

#Utilisation :

c = addition 45, 12
```

### 13.2.2 Interopérabilité

Coffeescript reste compatible avec les librairies existantes (pas besoin de ré-écrire jQuery ... ouf !)

*Attendre le chargement de la page pour lancer les commandes :*

```
$(function() {
    some();
    init();
    calls();
}) ;
```

*La même chose en Coffeescript :*

```
$ ->
    some()
    init()
    calls()
```

### 13.2.3 Interpolation et Chaînes de caractères

Imaginons un objet bob :

*En Javascript :*

```
var bob = {
  firstName : "Bob",
  lastName : "Morane",
  sayHello : function() { return "Hello !"}
}
```

Nous voulons afficher un message à partir des propriétés de Bob, aujourd’hui nous faisons comme ceci :

*En Javascript :*

```
console.log(
  "Firstname : " + bob.firstName +
  " Lastname : " + bob.lastName +
  "Hello : " +
  bob.sayHello()
) ;
```

Coffeescript introduit des nouveaux concepts à propos des chaînes de caractères et particulièrement `#{}variable` qui permet d’insérer la valeur de variable dans une chaîne. Nous aurons donc ceci :

*Version Coffeescript :*

```
bob =
  firstName : "Bob"
  lastName : "Morane"
  sayHello : ->
    "Hello !"

console.log "
  Firstname : #{bob.firstName}
  Lastname : #{bob.lastName}
  Hello : #{bob.sayHello()}
"
```

**Remarquez au passage**, la possibilité d’écrire vos chaînes de caractères sur plusieurs lignes (dans ce cas les saut de lignes ne sont pas pris en compte, pour les prendre en compte utilisez des triples « doubles-quotes » : “””)

### 13.2.4 Jouez un peu avec les tableaux

Nous avons le tableau suivant (en Coffeescript) :

*Copains :*

```
copains = [
  { nom : "Bob", age : 30 }
  { nom : "Sam", age : 50 }
  { nom : "John", age : 20 }
]
```

Nous voudrions obtenir tous les copains de moins de 50 ans :

*Jeunes copains :*

```
Result = (copain for copain in copains when copain.age < 50)
```

### 13.3 Et enfin (et surtout ?) les classes

Maintenant, grâce à Coffeescript, nous pouvons écrire des classes. Il faut aussi savoir ceci : le mot clé `this` devient `@`, les « champs » (propriétés) de la classes sont déclarés dans le constructeur. Mais un exemple est souvent plus parlant que trop d'explications :

#### 13.3.1 Notre 1ère classe

*Classe “Human” :*

```
class Human
  constructor : (firstName, lastName) ->
    #fields : @ = this
    @firstName = firstName
    @lastName = lastName

    #method
    hello : ->
      console.log "Hello #{@firstName} #{@lastName}"

  #Utilisation
  Bob = new Human "Bob", "Morane"

  Bob.hello()
```

#### 13.3.2 Propriétés & valeurs par défaut

Vous pouvez aller encore plus vite et déclarer vos propriétés et leur valeur par défaut directement en paramètres du constructeur.

*Classe “Human”, autre version, même résultats :*

```
class Human
  constructor : (@firstName = "John", @lastName = "Doe") ->
    hello : ->
      console.log "Hello #{@firstName} #{@lastName}"
```

### 13.3.3 Comme en Java : Composition, Association, Encapsulation

La notion de classe apportée par Coffeescript (même s'il existe en javascript, comme nous avons pu le voir par exemple avec le modèle objet de Backbone) permet facilement de mettre en œuvre les différents concepts habituels en programmation « orientée classes » et plus facilement encore décliner les « design patterns ».

*Composition de classes :*

```
class Dog
  constructor:(@name)->

class Human
  constructor:(@first, @last, @dog)->

Bob = new Human "Bob", "Morane", new Dog "Wolf"
console.log "Le chien de Bob s'appelle #{Bob.dog.name}"
```

*Encapsulation de classes :*

```
class Human
  class Hand
    constructor:(@whichOne)->

    constructor:(@first, @last)->
      @leftHand = new Hand "left"
      @rightHand = new Hand "right"

  console.log JSON.stringify new Human "Bob", "Morane"
```

### 13.3.4 Comme en Java : les membres statiques

La définition/déclaration de membres statiques se fait pour les propriétés en dehors du constructeur cette fois-ci et pré-fixées par @, les méthodes statiques « commencent » elles aussi par @ :

*Toujours et encore notre classe « Human » :*

```
class Human
  @counter : 0 #static property

  constructor : (@firstName, @lastName) ->
    Human.counter += 1

  #static method
  @howMany : ->
    Human.counter

Bob = new Human "Bob", "Morane"
console.log "Human.counter #{Human.howMany()}"
```

### 13.3.5 Mais aussi (comme en Java) : l'héritage !

Coffeescript introduit le mot clé `extends` pour permettre à une classe d'hériter d'une autre classe, et cela s'utilise de façon très intuitive :

*Un humain, et un super héros qui hérite d'humain ... :*

```
class Human
  constructor : (@firstName, @lastName) ->

  hello : ->
    "Hello #{@firstName} #{@lastName}"

class Superhero extends Human
  constructor : (@firstName, @lastName, @name) ->

  hello : ->
    super + " known as #{@name}"

SuperMan = new Superhero "Clark", "Kent", "SuperMan"
console.log SuperMan.hello()
```

**Notez au passage** l'apparition du mot-clé `super` qui permet « d'appeler » la méthode du « parent ».

Une première conclusion s'impose : Maintenant vous pouvez faire du javascript « orienté classes » tout en conservant les possibilités actuelles de javascript.

## 13.4 J'aime / Je n'aime pas

Coffeescript fait beaucoup de bruit dans la communauté javascript, mais aussi du côté des développeurs java, .Net, etc. ... Il a été assez bien adopté par les développeurs Ruby car il a une syntaxe assez proche.

Ce qui plaît beaucoup, plus particulièrement aux développeurs « non javascript », c'est le concept de classes, mais les développeurs javascript « puristes » ne voient pas l'intérêt d'utiliser des classes, les concepts objet de javascript apparaissant largement suffisants.

Personnellement, si l'aspect classe doit contribuer à l'adoption de javascript, laissons les classes ! Il y a encore trop de développeurs rebutés par javascript pour de fausses raisons. Ce n'est probablement pas pour rien si le concept de classe apparaît dans la future version de javascript ES6.

Ce qui plaît beaucoup moins, et là les développeurs javascripts et java se rejoignent, c'est l'absence de points virgule, d'accolades, et la notion d'indentation « significative » comme en python, rendant le code difficilement lisible s'il est trop long.

D'autres vous diront que la simplification du code grâce à Coffeescript contribue à sa lisibilité. Comme quoi, les goûts et les couleurs ...

Le plus gros défaut de Coffeescript, reste la difficulté à débugger du code coffeescript, l'impossibilité de l'utiliser directement dans la console du navigateur. Cependant le projet **Source Map** devrait régler ce type de problème à l'avenir (et pour d'autres transpilers aussi) : <http://www.coffeescriptlove.com/2012/04/source-maps-for-coffeescript.html>.

On pourrait ensuite se demander, mais que va devenir Coffeescript dans le futur avec l'apparition de la nouvelle version de javascript ? Je vous répondrais, qu'avant que cette version soit déployée sur tous les postes de travail (donc que tout le monde dispose d'un navigateur de dernière génération) il va se passer plusieurs années et que Coffeescript génère du code javascript qui fonctionne sur la majorité des navigateurs existants.

Brendan Heich (le papa de javascript) a officiellement donné sa bénédiction « publique » à Coffeescript, nul doute que Jeremy Ashkenas ne fasse évoluer Coffeescript en fonction des spécifications javascript pour garder une compatibilité ascendante et descendante. Pour rappel, en début de page du site de Coffeescript, il est écrit : « **The golden rule of CoffeeScript is: “It’s just JavaScript”** ».

Pour ma part, je pense que c'est un excellent outil d'apprentissage (il génère du code javascript « propre ») mais sur un projet de réalisation professionnel, il est à réserver aux « gurus » qui maîtrise déjà javascript.

Attention, un challenger de poids vient de naître dans le monde des transpilers javascript : **TypeScript**, porté par Microsoft, développé entre autre par le papa de l'ancêtre du Turbo Pascal mais aussi du C# et qui tend à gommer une grande partie des défauts reprochés à Coffeescript. Cela fera l'objet d'un prochain chapitre, mais avant cela, passons à la ré-écriture de notre Blog en Coffeescript.

## 13.5 Ré-écriture de notre blog ?! S'outiller

Avant toute chose, sauvegardez votre arborescence applicative à un autre emplacement pour pouvoir la réutiliser.

### 13.5.1 Installation de Coffeescript

L'installation de Coffeescript est très simple, elle s'effectue en mode commande avec npm (installé en même temps que Node.js) :

```
npm install -g coffee-script
```

**Remarque :** pour les utilisateurs sous OSX, vous devez normalement utiliser la commande `sudo npm install -g coffee-script`

A partir de maintenant, vous pouvez compiler/transpiler du code Coffeescript en javascript avec la commande suivante :

```
coffee -compile Human.coffee
```

qui produira un fichier javascript `Human.js`.

Pour plus d'informations : <http://jashkenas.github.com/coffee-script/#usage>.

### 13.5.2 “Industrialisation” de la transpilation

Nous avons donc décidé de re-écrire la partie javascript cliente de notre blog. Pour cela dans le répertoire de notre application, créez un répertoire `public.coffee` avec un sous-répertoire `models` et un sous-répertoire `views`. Ce répertoire et ces deux sous-répertoires contiendront les fichiers source coffeescript (que vous pouvez d'ores et déjà créer “vides” dans les répertoires correspondants) :

```
public.coffee\
    |-models\
        |-post.coffee
    |-views\
        |-AdminView.coffee
        |-LoginView.coffee
        |-MainView.coffee
        |-PostsListView.coffee
        |-PostView.coffee
        |-SidebarView.coffee
    |
    |-Blog.coffee
    |-main.coffee
    |-routes.coffee
```

Le “but du jeu” étant de transpiler tous les fichiers `.coffee` en fichier javascript `.js` dans le répertoire `public` (et ses sous-répertoires). Faire ceci “à la main” en prenant les fichiers un par un peut devenir très rapidement fastidieux. Mais heureusement, l'installation de Coffeescript inclut un système simple de “build” qui s'appelle **Cake**, qui est capable d'effectuer des tâches simples décrites dans un fichier `Cakefile`. Plus d'information ici : <http://jashkenas.github.com/coffee-script/#cake>.

Donc à la racine de votre application, créez un fichier `Cakefile` avec le contenu suivant :

*Cakefile* :

```
fs = require 'fs'
{print} = require 'util'
{spawn} = require 'child_process'

build = (callback) ->
    coffee = spawn 'coffee', ['-c', '-o', 'public', 'public.coffee']
    coffee.stderr.on 'data', (data) ->
        process.stderr.write data.toString()
    coffee.stdout.on 'data', (data) ->
        print data.toString()
    coffee.on 'exit', (code) ->
        callback?() if code is 0

task 'build', 'Build public/ from public.coffee/' , ->
    build()
```

En fait nous expliquons à Cake que nous souhaitons transpiler en javascript tous les fichiers du répertoire source `public.coffee` vers le répertoire cible `public`. Et pour cela, il suffira, dans un terminal (ou une console) de lancer la commande : `cake build` (à la racine de notre application, là où est situé le fichier `Cakefile`).

## 13.6 Ré-écriture / Traductions

Alors, l'objectif n'est pas d'apprendre Coffeescript (cela pourrait donner lieu à un ouvrage entier), mais de montrer comment il est possible d'écrire notre blog "autrement", donc je vous livre ici la traduction en Coffeescript (pas forcément dans les règles de l'art) qui fonctionnera et assortie de quelques remarques.

### 13.6.1 Blog.coffee

```
window.Blog =
  Models : {}
  Collections : {}
  Views : {}
  Router : {}
```

### 13.6.2 main.coffee

```
yepnope
load:
  jquery: "libs/vendors/jquery-1.7.2.js"
  underscore: "libs/vendors/underscore.js"
  backbone: "libs/vendors/backbone.js"
  mustache: "libs/vendors/mustache.js"

#NameSpace
blog: "Blog.js"

#Models
posts: "models/post.js"

#Controllers
sidebarview: "views/SidebarView.js"
postslistviews: "views/PostsListView.js"
mainview: "views/MainView.js"
loginview: "views/LoginView.js"
postview: "views/PostView.js"
adminview: "views/AdminView.js"

#Routes
routes: "routes.js"

callback:
  routes: ->
    console.log "routes loaded ..."

complete: ->
  $ ->
    console.log "Launching application ..."
    window.blogPosts = new Blog.Collections.Posts()
    window.mainView = new Blog.Views.MainView(collection: blogPosts)
```

```

===== Admin =====
window.adminView = new Blog.Views.AdminView(collection: blogPosts)

===== Authentification =====
window.loginView = new Blog.Views.LoginView(adminView: adminView)

window.postView = new Blog.Views.PostView()
window.router = new Blog.Router.RoutesManager(collection: blogPosts)

Backbone.history.start()

```

### 13.6.3 post.coffee

```

class Blog.Models.Post extends Backbone.Model
  urlRoot: "/blogposts"

class Blog.Collections.Posts extends Backbone.Collection

  model: Blog.Models.Post

  all: ->
    @url = "/blogposts"
    @

  query: (query) ->
    @url = "/blogposts/query/" + query
    @

```

### 13.6.4 AdminView.coffee

```

class Blog.Views.AdminView extends Backbone.View
  el: $("#admin")
  initialize: ->
    @template = $("#admin_template").html()

    #je prévois de trier ma collection
    @collection.comparator = (model) ->
      -(new Date(model.get("date")).getTime())

  render: ->
    renderedContent = Mustache.to_html(@template,
      posts: @collection.toJSON()
    )
    @$el.html renderedContent

  events:
    "click #btn_update": "onClickBtnUpdate"

```

```

"click #btn_create": "onClickBtnCreate"
"click #btn_send": "sendPost"

onClickBtnUpdate: ->
  selectedId = $("#post_choice").val()
  post = @collection.get(selectedId)

#Je récupère les informations du post et les affiche
$("#admin > [name='id']").html post.get("id")
$("#admin > [name='author']").val post.get("author")
$("#admin > [name='title']").val post.get("title")
$("#admin > [name='message']").val post.get("message")

onClickBtnCreate: ->

#je ré-initialise les zones de saisie
$("#admin > [name='id']").html ""
$("#admin > [name='author']").val ""
$("#admin > [name='title']").val ""
$("#admin > [name='message']").val ""

sendPost: -> #Sauvegarde
#that = this #pour conserver le contexte
id = $("#admin > [name='id']").html()
post = undefined
if id is "" #si l'id est vide c'est une création
  post = new Blog.Models.Post()
else #l'id n'est pas vide c'est une mise à jour
  post = new Blog.Models.Post(id: $("#admin > [name='id']").html())
post.save
  author: $("#admin > [name='author']").val()
  title: $("#admin > [name='title']").val()
  message: $("#admin > [name='message']").val()
  date: new Date()
,
  success: ->

#Si la transaction côté serveur a fonctionné

#je recharge ma collection
@collection.fetch success: =>

#mise à jour de la vue admin
@render()

#La vue principale se re-mettra à jour
#automatiquement, car elle est "abonnée"
#aux changement de la collection

```

```

error: ->

//TODO : parler de fat arrow =>

```

### 13.6.5 LoginView.coffee

```

class Blog.Views.LoginView extends Backbone.View
  el: $("#blog_login_form")
  initialize: (args) ->
    that = this
    @adminView = args.adminView
    @template = $("#blog_login_form_template").html()

    #on vérifie si pas déjà authentifié
    $.ajax
      type: "GET"
      url: "/alreadyauthenticated"
      error: (err) ->
        console.log err

      success: (dataFromServer) ->
        if dataFromServer.firstName
          that.render "Bienvenue", dataFromServer
        else
          that.render "????",
            firstName: "John"
            lastName: "Doe"

  render: (message, user) ->
    renderedContent = Mustache.to_html(@template,
      message: message
      firstName: (if user then user.firstName else ""))
    lastName: (if user then user.lastName else "")

    #adminLink : user.isAdmin ? '#/admin' : "",
    adminLinkLabel:
      (if user then (if user.isAdmin then "Administration" else "") else "")
    )
    @$el.html renderedContent

  events:
    "click .btn-primary": "onClickBtnLogin"
    "click .btn-inverse": "onClickBtnLogoff"
    "click #adminbtn": "displayAdminPanel"

  displayAdminPanel: ->
    @adminView.render()

```

```

onClickBtnLogin: (domEvent) ->
  fields = $("#blog_login_form :input")
  #that = this
  $.ajax
    type: "POST"
    url: "/authenticate"
    data:
      email: fields[0].value
      password: fields[1].value

    dataType: "json"
    error: (err) ->
      console.log err

    success: (dataFromServer) =>
      if dataFromServer.infos
        @render dataFromServer.infos
      else
        if dataFromServer.error
          @render dataFromServer.error
        else
          @render "Bienvenue", dataFromServer

onClickBtnLogoff: =>
  #that = this
  $.ajax
    type: "GET"
    url: "/logoff"
    error: (err) ->
      console.log err

    success: (dataFromServer) ->
      console.log dataFromServer
      @render "???", {
        firstName: "John"
        lastName: "Doe"
      }

```

### 13.6.6 PostsListView.coffee

```

class Blog.Views.PostsListView extends Backbone.View
  el: $("#posts_list")
  initialize: ->
    @template = $("#posts_list_template").html()

  render: ->
    renderedContent = Mustache.to_html(@template,
      posts: @collection.toJSON()
    )
    @$el.html renderedContent

```

### 13.6.7 PostView.coffee

```
class Blog.Views.PostView extends Backbone.View
  el: $("#posts_list")
  initialize: ->
    @template = $("#post_details_template").html()

  render: (post) ->
    renderedContent = Mustache.to_html(@template,
      post: post.toJSON()
    )
    @$el.html renderedContent
```

### 13.6.8 SidebarView.coffee

```
class Blog.Views.SidebarView extends Backbone.View
  el: $("#blog_sidebar")
  initialize: ->
    @template = $("#blog_sidebar_template").html()

  render: ->
    renderedContent = Mustache.to_html(@template,
      posts: @collection.toJSON()
    )
    @$el.html renderedContent
```

### 13.6.9 MainView.coffee

```
class Blog.Views.MainView extends Backbone.View
  initialize: ->
    @collection.comparator = (model) ->
      -(new Date(model.get("date")).getTime())

    _.bindAll this, "render"
    @collection.bind "reset", @render
    @collection.bind "change", @render
    @collection.bind "add", @render
    @collection.bind "remove", @render
    @sidebarView = new Blog.Views.SidebarView()
    @postsListView = new Blog.Views.PostsListView(collection: @collection)

  render: ->

    @sidebarView.collection = new Blog.Collections.Posts(@collection.first(3))
    @sidebarView.render()
    @postsListView.render()
```

### 13.6.10 routes.coffee

```

class Blog.Router.RoutesManager extends Backbone.Router
  initialize: (args) ->
    @collection = args.collection

  routes:
    "post/:id_post": "displayPost"
    hello: "hello"
    "*path": "root"

  root: ->
    @collection.all().fetch success: (result) ->

      #ça marche !!!
      console.log "fetching collection", result

  hello: ->
    $(".hero-unit > h1").html "Hello World !!!"

  displayPost: (id_post) ->
    tmp = new Blog.Models.Post(id: id_post)
    tmp.fetch success: (result) ->
      postView.render result

```

Ouf, c'est fini.

### 13.6.11 Transpilons

Si vous lancez la commande : `cake build` vous obtiendrez de nouveau fichiers javascript dans le répertoire public. Relancez votre application, vous noterez qu'elle fonctionne comme avant.

### 13.6.12 Conclusion(s)

Les conclusions sont très personnelles. J'aime beaucoup le principe de “classe” et je le trouve très utile pour organiser son code, tout particulièrement en ce qui concerne les modèles et les collections. Au passage, vous avez du remarquer que le modèle objet de Backbone se marie parfaitement avec celui de Coffeescript (on peut hériter directement d'un Backbone.Model par exemple). C'est normal, les 2 outils sont codés par le même auteur.

Ce que j'aime aussi, c'est les possibilités des chaînes de caractères.

Par contre, je trouve que pour des portions de code très longue, on perd en visibilité (cf. `AdminView` ou `LoginView`) (mais ça n'engage que moi).

Cependant, je ne peux m'empêcher d'aimer ce “petit” langage et les concepts qu'il apporte.

... A vous de voir ;)

## 14 Autres Frameworks MVC

*Sommaire*

- *CanJS*
- *Spine*
- *Knockout*
- *Les autres ...*

*Il faut aller voir ailleurs !!! Backbone peut ne pas correspondre à vos attentes, à votre façon de coder, ... Je vous engage à tester d'autres frameworks.*

*Il existe aujourd'hui de TRES nombreux frameworks MVC en javascript "côté navigateur". Certains fournissent beaucoup plus de services que Backbone, d'autres vont même plus loin, puisqu'ils vont jusqu'à fournir un framework associé côté serveur (en javascript avec du nodejs). Ils sont plus ou moins complexes, plus ou moins bien documentés.*

L'objectif de ce chapitre n'est pas de faire un cours sur tous les frameworks javascript existants (chaque mérirait un ouvrage), mais de lever un coin de voile sur eux, et montrer que chacun possède ses spécificités. J'en ai sélectionné 2, sur un seul critère de sélection (qui me tient à cœur) : **LA SIMPLICITE** (après la notion de simplicité d'un dev à l'autre, est relative ...), il faut que le framework soit simple à mettre en oeuvre, sans pour autant être obligé de le connaître par coeur avant de pouvoir l'utiliser. Autrement dit, si au bout d'un quart d'heure de lecture, vous avez compris les principes de bases, le framework que vous étudiez est susceptible d'être facilement compréhensible par le reste de votre équipe (par exemple) et donc vous ne devriez pas avoir de frein à l'adoption. Si au bout d'une demi-heure, vous n'êtes pas arrivé à l'installé correctement, vous pouvez commencer à avoir des doutes (c'est une méthode très personnelle, si elle ne garantit pas les fonctionnalités du framework, elle vous assure au moins sa simplicité). La qualité de la documentation peut avoir un impact fort dans vos choix.

### 14.1 CanJS

<http://canjs.us>

**CanJS** n'est pas le plus léger des frameworks, mais son approche est très intéressante et relativement intuitive, tout particulièrement pour des développeurs "backend". D'un point de vue objet, on retrouve dès le départ le concept de méthode statique dans le modèle, et c'est la "pseudo classe" modèle qui se chargera de "ramener" un modèle, mais aussi la liste des modèles (alors que dans Backbone, c'est un objet à part : la collection qui s'en charge). Mais un exemple sera tout de suite beaucoup plus parlant :

**Définition d'un modèle :**

```
var Post = can.Model({
  findAll : 'GET /blogposts',
  findOne : 'GET /blogposts/{id}',
  create : 'POST /blogposts',
  update : 'PUT /blogposts/{id}',
  destroy : 'DELETE /blogposts/{id}'
}, {});
```

Ensuite si je souhaite “récupérer” du serveur un modèle particulier par son id :

```
Blog.Post.findOne({id:"2003mac1"},function(post){console.log(post);})
```

Puis, si je veux tous les modèles :

```
Blog.Post.findAll({}, function(posts){ console.log(posts);});
```

... qui me retournera un tableau de modèles.

Vous remarquerez que l'on définit toutes les routes au niveau du modèle, et non pas une seule url comme dans Backbone.

Question templating, **CanJS** embarque son propre langage de template qui ressemble fortement à celui d'Underscore (mais rien ne vous empêche d'utiliser autre chose). Côté “DOM”, **CanJS** sait travailler avec jQuery, Zepto, Dojo, Mootools et YUI, à vous de faire votre choix et de télécharger la version adaptée.

#### 14.1.1 Mise en oeuvre rapide

Pour notre exemple, j'ai utilisé la version de CanJS adaptée pour jQuery, que vous pouvez télécharger ici : <https://github.com/downloads/jupiterjs/canjs/can.jquery-1.0.7.js> et que vous copierez ensuite dans public/libs/vendors (nous utilisons notre application pour pouvoir se connecter à la partie serveur).

Dans notre répertoire public (à la racine), créez une page index.canjs.html avec le code suivant :

*Index.canjs.html :*

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>CanJS</title>
    <link href="libs/vendors/bootstrap/css/bootstrap.css" rel="stylesheet">
    <style>
        body {
            padding-top: 60px;
            padding-bottom: 40px;
        }
    </style>
    <link href="libs/vendors/bootstrap/css/bootstrap-responsive.css" rel="stylesheet">
</head>

<body>

    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand">Mon Blog avec CanJS</a>
            </div>
        </div>
    </div>
```

```

        </div>
    </div>

    <div class="container-fluid">

        <div class="row-fluid">

            <script type="text/ejs" id="myposts_template">
                <% for( var i = 0; i < this.length; i++ ) { %>
                    <li>
                        <b><%= this[i].title %></b>
                        <br>
                        <p><%= this[i].message %></p>
                        <hr>
                    </li>
                <% } %>
            </script>

            <ul id="myposts"></ul>

        </div>

    </div>

</body>

<script src="libs/vendors/yepnope.1.5.4-min.js"></script>

<script type="text/javascript">
yepnope({
    load: {
        jquery : 'libs/vendors/jquery-1.7.2.js',
        canjs : 'libs/vendors/can.jquery-1.0.7.js'
    },
    complete : function () {
        $(function (){

            window.Blog = {}

            Blog.Post = can.Model({
                findAll : 'GET /blogposts',
                findOne : 'GET /blogposts/{id}',
                create : 'POST /blogposts',
                update : 'PUT /blogposts/{id}',
                destroy : 'DELETE /blogposts/{id}'
            }, {});

            Blog.Post.findAll({}, function(posts){
                $('#myposts').html(can.view( 'myposts_template', posts ))
            })
        })
    }
})
</script>

```

```

        });
    });
}

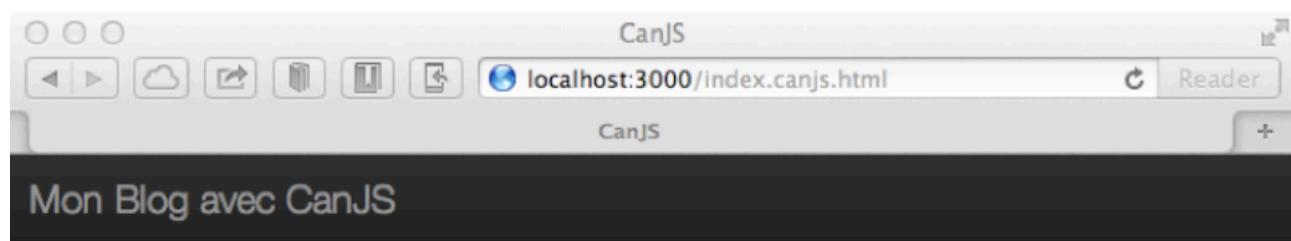
</script>
</html>

```

Nous avons conservé le mécanisme de chargement de yepnope et jQuery, donc une fois le chargement de la page terminé, nous avons défini notre modèle avec les routes nécessaires pour accéder aux services de données, puis nous exécutons un `findAll` pour obtenir la liste des posts sur le serveur et l'affichons dans la page une fois les données récupérées grâce à l'objet `can.view`.

Vous n'avez plus qu'à enregistrer et relancer votre application et ouvrir l'url :

`http://localhost:3000/index.canjs.html` dans votre navigateur, pour obtenir la liste des posts :



- **Premier Message**

Maecenas sed diam eget risus varius blandit sit amet non magna.

- **Les Vues**

  Lorem ipsum dolor sit amet, consectetur adipiscing elit.

- **Les Modèles**

  Donec id elit non mi porta gravida at eget metus.

- **Backbone ???**

  Integer posuere erat a ante venenatis dapibus posuere velit aliquet.

- **Les Routes**

  Cras mattis consectetur purus sit amet fermentum.

### 14.1.2 Conclusion sur CanJS

Je trouve CanJS facile d'accès. Il propose de nombreux plug-ins et apparaît très complet. Pour le moment, le projet semble actif. D'un point de vue code, il n'a pas la concision de Backbone, il y a beaucoup plus de chose, mais il propose aussi beaucoup plus de services "clés en main". Un autre avantage non négligeable est la documentation qui est claire, bien faite et facile d'accès.

**Remarque :** Si vous avez un doute sur la pertinence de votre choix, je vous conseille d'aller faire un tour sur <https://github.com/addyosmani/todomvc>, où Addy Osmani, un googler réputé et spécialiste entre autre de Backbone maintient (avec d'autres développeurs) des exemples de “TODO list” MVC javascript pour chacun des frameworks MVC (ou pas complètement MVC d'ailleurs) du moment. Si le framework que vous étudiez est dans la liste, il y a des chances que ce ne soit pas un mauvais choix, de plus vous disposerez d'un exemple de code (dans les règles de l'art) pour commencer.

## 14.2 Spine

<http://spinejs.com/>

Spine est très très inspiré de Backbone avec les spécificités suivantes (entre autres) :

- il est écrit en Coffeescript (mais vous pouvez bien sûr l'utiliser en pur javascript)
- Spine propose des Contrôleurs ! ... qui ne sont ni plus ni moins l'équivalent des Backbones.Views
- ... et considère que les vues sont représentée par les templates. A ce sujet Spine propose tout un mécanisme évolué mais nous n'utiliserons que Mustache de la même façon qu'avec Backbone.

Spine peut s'installer avec **npm** (node package manager) et ainsi proposer des outils supplémentaires permettant de générer un squelette de projet, de gérer les dépendances, ... Mais pour notre exemple, nous allons le faire “à l'ancienne”, l'objectif étant de ne modifier en rien la stack serveur existante. Vous pouvez donc vous préparer un environnement identique à celui du chapitre sur Coffeescript avec un répertoire **public.coffee** pour votre code coffeescript et n'oubliez pas le fichier de build **Cakefile**.

Un modèle Spine se décrit de la manière suivante :

```
class window.Post extends Spine.Model
  @configure "Post", "title", "message", "author"
  @extend Spine.Model.Ajax
  @url: "/blogposts"
```

la ligne `@configure "Post", "title", "message", "author"` définit le nom du modèle et de ses propriétés, ce qui permettra de faire référence ensuite à celles-ci de la façon suivante : `@title`, `@message`, `@author`, comme ceci par exemple, si nous ajoutons une méthode `toString()` à notre modèle :

*Utilisation des “propriétés” :*

```
class Post extends Spine.Model
  @configure "Post", "title", "message", "author"
  @extend Spine.Model.Ajax
  @url: "/blogposts"

  toString:()->
    "#{@title} : #{@message} / #{@author}"
```

Vous remarquerez que l'on ne définit qu'une seule url. Par contre le système de getter et de setter de Backbone disparaît (à vous de les écrire).

En ce qui concerne le contrôleur, il prend la forme suivante :

*Contrôleur spine :*

```
class Blog extends Spine.Controller

constructor:()->
    super

render:()->
```

#### 14.2.1 Mise en oeuvre rapide

Téléchargez la distribution <http://spinejs.com/pages/download>, dézippez et copiez les fichiers suivants dans `public/libs/vendors` :

- spine.js
- ajax.js
- list.js
- local.js
- manager.js
- relation.js
- route.js

Cette fois-ci, nous allons utiliser 2 fichiers : une page `index.spine.html` qui affichera nos données et un fichier de script coffeescript `main.spine.coffee` qui contiendra notre code applicatif.

A nouveau, dans notre répertoire `public` (à la racine), créez une page `index.spine.html` avec le code suivant :

*Index.spine.html :*

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Spine</title>
    <link href="libs/vendors/bootstrap/css/bootstrap.css" rel="stylesheet">
    <style>
        body {
            padding-top: 60px;
            padding-bottom: 40px;
        }
    </style>
```

```

<link href="libs/vendors/bootstrap/css/bootstrap-responsive.css" rel="stylesheet">
</head>

<body>

    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand">Mon Blog avec Spine</a>
            </div>
        </div>
    </div>

    <div class="container-fluid">

        <div class="row-fluid">

            <script type="text/template" id="myposts_template">
                {{#posts}}
                    <li>
                        <b>{{title}}</b><br>
                        <p>{{message}}</p>
                        <hr>
                    </li>
                {{/posts}}
            </script>

            <ul id="myposts"></ul>

        </div>

    </div>

</body>

<script src="libs/vendors/yepnope.1.5.4-min.js"></script>
<script src="main.spine.js"></script>

</html>

```

et enfin dans le répertoire `public.coffee`, créez un fichier `main.spine.coffee` avec le code ci-dessous :

`main.spine.coffee` :

```

yepnope
load:
  jquery  : "libs/vendors/jquery-1.7.2.js"
  mustache: "libs/vendors/mustache.js"
  spine   : "libs/vendors/spine.js"
  ajax    : "libs/vendors/ajax.js"

```

```

list      : "libs/vendors/list.js"
local    : "libs/vendors/local.js"
manager  : "libs/vendors/manager.js"
relation: "libs/vendors/relation.js"
route    : "libs/vendors/route.js"

complete: ->
$ ->

class Post extends Spine.Model
@configure "Post", "title", "message", "author"
@extend Spine.Model.Ajax
@url: "/blogposts"

toString:()->
"#{@title} : #{@message} / #{@author}"

class Blog extends Spine.Controller

constructor:()->
super
@posts = []
@template = $("#myposts_template").html()
@el = $ "ul"

# on s'abonne au "rafraîchissement" de la liste des posts
# lorsque la liste est rafraîchie, on affiche la liste (méthode render)
Post.bind 'refresh', =>
@posts = Post.all()
@render()

# on charge les posts à partir du serveur
# cela va déclencher l'évènement "refresh"
Post.fetch()

render:()->
@el.html Mustache.to_html(@template,
posts: @posts
)

window.myBlog = new Blog()

```

Maintenant positionnez vous au même endroit que le fichier `Cakefile`, lancez la commande `cake build`, lancez votre application serveur si ce n'est pas déjà fait et connectez vous, comme pour CanJS, vous obtiendrez à nouveau la même liste des posts dans votre navigateur :

The screenshot shows a web browser window with the title bar 'Spine'. The address bar displays 'localhost:3000/index.spine.'. The main content area has a dark header with the text 'Mon Blog avec Spine'. Below the header, there are five horizontal lines, each representing a section with a bullet point and placeholder text.

- Premier Message  
Maecenas sed diam eget risus varius blandit sit amet non magna.
- Les Vues  
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
- Les Modèles  
Donec id elit non mi porta gravida at eget metus.
- Backbone ???  
Integer posuere erat a ante venenatis dapibus posuere velit aliquet.
- Les Routes  
Cras mattis consectetur purus sit amet fermentum.

### 14.2.2 Conclusion sur Spine

Spine est moins simple que Backbone, mais s'en inspire énormément tout en restant facile d'accès (je fais abstraction de Coffeescript en faisant cette remarque). Il est possible d'utiliser Spine directement en javascript, mais le "style" de code ne prend vraiment de l'intérêt que dans un "mode Coffeescript" grâce à la notion de classe. De la même façon que CanJS, Spine apporte plus de services "clés en main" que Backbone, comme par exemple la gestion du localstorage. Spine propose aussi un volet "développement mobile" : <http://spinejs.com/mobile/index>. Si vous n'êtes pas effrayé par Coffeescript, c'est un framework à suivre et à adopter.

Globalement, ce qui m'a intéressé tant dans CanJS que dans Spine, c'est leur simplicité de mise en oeuvre. Il est possible en quelques minutes d'écrire quelques premières lignes opérationnelles, ce qui n'est pas forcément le cas avec d'autres frameworks beaucoup plus (ou trop?) complets.

## 14.3 Knockout

Knockout <http://knockoutjs.com>, l'OVNI de l'équipe, on ne parle plus de MVC mais du pattern "Model-View-View Model", très très orienté IHM, puisqu'il renforce le lien entre les données et l'IHM, c'est

un peu déroutant (mais où est donc encore passé le contrôleur !?), mais cela permet de réaliser des interfaces très réactives en fonction des changements et évolutions des données.

En ce qui concerne **Knockout**, nous n'allons pas repartir de nos exemples précédents, car ce qui caractérise **Knockout**, c'est son système de “**binding**” qui “frise” le “**magique**” : vos actions sur les modèles sont directement impactées sur les vues (le DOM HTML) et inversement vos actions sur le DOM (saisie dans une zone de texte par exemple) sont directement reflétées sur les modèles. Le concept de modèle en tant que tel (comme dans Backbone par exemple) n'existe pas, donc à vous de les écrire, mais **Knockout** apporte les outils nécessaires pour “faire” des “**propriétés observables**” et des “**tableaux observables**” qui seront ensuite “liés” aux vues qui ne sont ni plus ni moins que votre code html.

Donc vous l'aurez compris, **Knockout** met en oeuvre le pattern **Observer** :

[http://en.wikipedia.org/wiki/Observer\\_pattern](http://en.wikipedia.org/wiki/Observer_pattern)

Ou encore mieux, la version “javascript” par Addy Osmani :

<http://addyosmani.com/resources/essentialjsdesignpatterns/book/#observerpatternjavascript>.

#### 14.3.1 Mise en oeuvre très très rapide ... Mais suffisante

Récupérez donc la librairie Knockout.js par ici :

<http://cloud.github.com/downloads/SteveSanderson/knockout/knockout-2.2.0.js>

Puis copiez la dans un répertoire, dans lequel vous allez créer une page `index.html` avec la structure suivante :

```
<!DOCTYPE html>
<html>
<head>
    <script src="knockout-2.2.0.js"></script>
</head>
<body>

    <h1>Tuto KnockOut</h1>
    <!-- ICI VIENDRONT LES VUES -->
</body>
<script>
    /* ICI VIENDRA NOTRE CODE */
</script>
</html>
```

#### 14.3.2 Un modèle selon Knockout

Avec **Knockout**, pour définir qu'une propriété est “observable”, on utilise la méthode :

`ko.observable(nom_de_la_propriété)`

Définissons donc notre 1er modèle :

```

var Message = function (from, subject, body) {
    this.from = ko.observable(from);
    this.subject = ko.observable(subject);
    this.body = ko.observable(body);

    //je m'abonne aux modification de from
    this.from.subscribe(function(value){console.log("value : ", value);});
}

```

Toute propriété “observable” du modèle `Message` va ensuite s’utiliser de la manière suivante (pour instancier, faites un `myMessage = new Message("Bob", "Important", "Hello World !")`):

- changer la valeur de `from` : `myMessage.from("Bob Morane")`
- obtenir la valeur de `from` : `myMessage.from()`
- s’abonner aux changements de valeur de `from` :
 

```
this.from.subscribe(function(value){ //do something ... })
```

Simple, non ?

#### 14.3.3 Collections ?

La notion de collection de Backbone n’existe plus, nous travaillons simplement avec des tableaux (`Array`). Je souhaite donc un tableau “observable” de messages, je vais procéder comme ceci (en utilisant `ko.observableArray`):

```

var Messages = ko.observableArray([
    new Message("John Doe", "First Message", "Hello World"),
    new Message("Bob Morane", "hello", "hello world"),
    new Message("Sam Le Pirate", "Salut", "Salut à Tous")
]);

```

Donc encore plus simple que pour les modèles !!! (*donc j’ai un tableau observable de modèles observables ...*)

#### 14.3.4 Attachons notre modèle à des vues !

Imaginons, que j’ai une portion de mon code html qui est dédiée à afficher un message bien précis :

```

<div id="messageView">
    <span data-bind="text: from"></span>
    <span data-bind="text: subject"></span>
    <span data-bind="text: body"></span>
</div>

```

Le code javascript pour attacher le message à `messageView` sera le suivant :

```
msg = new Message("bob", "hello", "hello world");

ko.applyBindings(msg, document.querySelector("#messageView"));
```

Le lien sera fait avec les attributs html `data-bind`. Toute modification de notre modèle sera reflétée instantanément dans le code html (mais nous y reviendrons plus tard dans la démonstration).

De la même façon, si je souhaite faire un formulaire de saisie lié à mon message :

```
<form>
  <input data-bind="value: from"/> <input data-bind="value: subject"/>
</form>
```

Notez cette fois `data-bind="value: from"` au lieu de `data-bind="text: from"`, `value` exprime le “binding à double sens” (là aussi nous y reviendrons plus tard).

Et le code javascript pour attacher le message au formulaire sera le suivant :

```
ko.applyBindings(msg, document.querySelector("form"));
```

Au fait, l'utilisation de jQuery n'est pas obligatoire (mais tellement pratique), avec jQuery, à la place de `document.querySelector("selector")` nous aurions donc `$(“selector”)`.

**Pas d'objet vue !** La vue est entièrement représentée par les bouts de code HTML.

#### 14.3.5 Attachons notre liste de messages à une vue

En ce qui concerne notre tableau “observable”, cela reste aussi simple :

```
<div id="messagesList">
  <table>
    <thead>
      <tr><th>From</th><th>Subject</th><th>Body</th></tr>
    </thead>
    <tbody data-bind="foreach: messages">
      <tr>
        <td data-bind="text: from"></td>
        <td data-bind="text: subject"></td>
        <td data-bind="text: body"></td>
      </tr>
    </tbody>
  </table>
</div>
```

Pour exprimer à **Knockout** que l'on veut afficher une liste d'éléments, on utilise l'attribut `data-bind="foreach: messages"`. Et en javascript, nous devrons écrire ceci :

```
ko.applyBindings({messages:Messages}, document.querySelector("#messagesList"));
```

Donc, toujours aussi simple !!!

### 14.3.6 Code définitif de notre page index.html

Avant de “lancer” notre page, vérifions que le code ressemble à ceci :

```

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <script src="knockout-2.2.0.js"></script>
</head>
<body>

    <h1>Tuto KnockOut</h1>

    <div id="messageView">
        <span data-bind="text: from"></span>
        <span data-bind="text: subject"></span>
        <span data-bind="text: body"></span>
    </div>
    <hr>
    <div id="messagesList">
        <table>
            <thead>
                <tr><th>From</th><th>Subject</th><th>Body</th></tr>
            </thead>
            <tbody data-bind="foreach: messages">
                <tr>
                    <td data-bind="text: from"></td>
                    <td data-bind="text: subject"></td>
                    <td data-bind="text: body"></td>
                </tr>
            </tbody>
        </table>
    </div>
    <hr>
    <form>
        <input data-bind="value: from"/><input data-bind="value: subject"/>
    </form>

</body>
<script>

var Message = function (from, subject, body) {
    this.from = ko.observable(from);
    this.subject = ko.observable(subject);
    this.body = ko.observable(body);

    this.from.subscribe(function(value){console.log("value : ", value);});
}

```

```

var Messages = ko.observableArray([
    new Message("John Doe", "First Message", "Hello World"),
    new Message("Bob Morane", "hello", "hello world"),
    new Message("Sam Le Pirate", "Salut", "Salut à Tous")
]);

msg = new Message("bob", "hello", "hello world");

ko.applyBindings(msg, document.querySelector("#messageView"));

ko.applyBindings(msg, document.querySelector("form"));

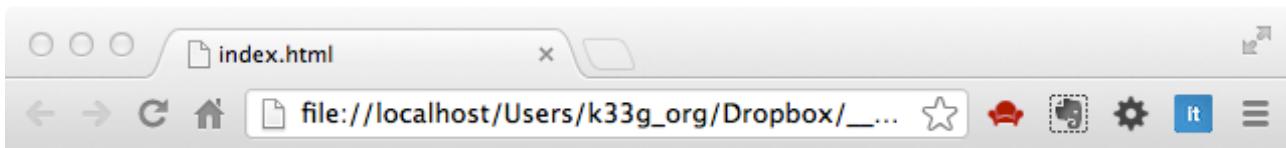
ko.applyBindings({messages:Messages}, document.querySelector("#messagesList"));

</script>
</html>

```

#### 14.3.7 Démonstration !!!

Ouvrez votre page dans votre navigateur préféré (Chrome donc ;)) et ouvrez aussi la console du navigateur. Vous pouvez donc noter que le lien entre les données et la page s'effectue bien :



# Tuto KnockOut

bob hello hello world

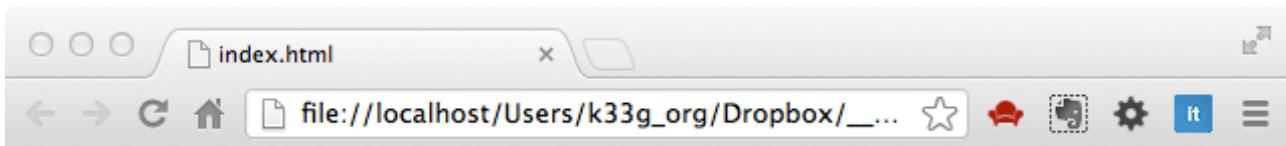
---

From	Subject	Body
John Doe	First Message	Hello World
Bob Morane	hello	hello world
Sam Le Pirate	Salut	Salut à Tous

---



Ensuite dans la console, tapez ceci : `msg.from("BOBBY").subject("Hello You !!!")` et validez. Vous vous apercevez que les modifications sont prises en compte automatiquement aussi bien dans la première ligne d'affichage du message que dans le formulaire de saisie. Notez au passage la possibilité "jqueryesque" (chaînée) de mise à jour des propriétés :

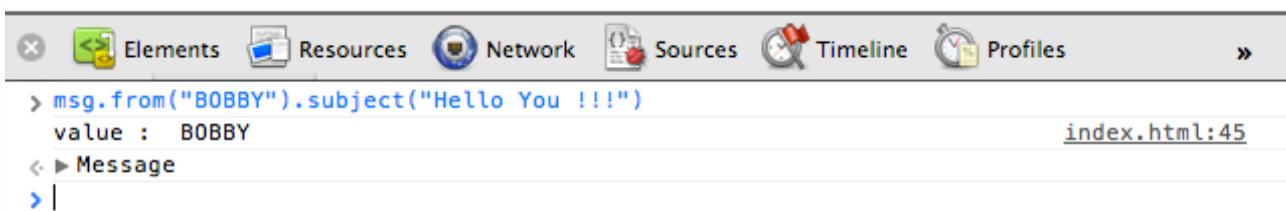


# Tuto KnockOut

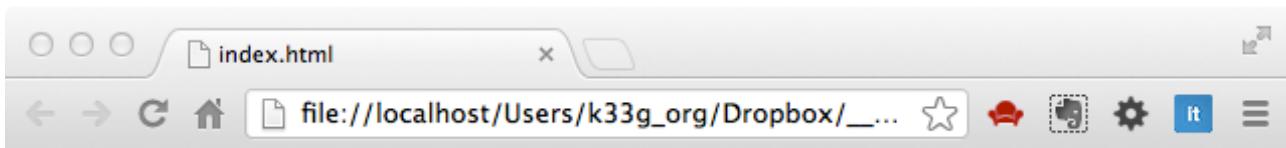
BOBBY Hello You !!! hello world

From	Subject	Body
John Doe	First Message	Hello World
Bob Morane	hello	hello world
Sam Le Pirate	Salut	Salut à Tous

BOBBY                   Hello You !!!



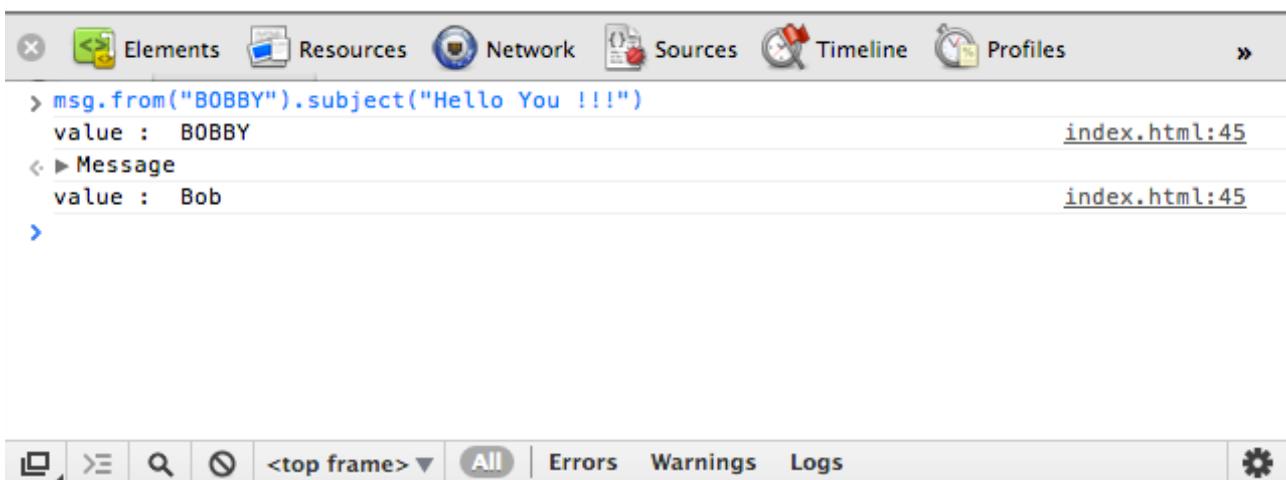
Maintenant, si vous saisissez Bob dans la zone de saisie, vous pouvez vérifier que les changements sont bien répercutés dans les 2 sens :



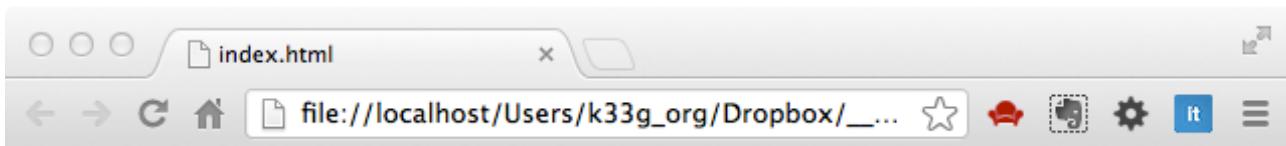
# Tuto KnockOut

Bob Hello You !!! hello world

From	Subject	Body
John Doe	First Message	Hello World
Bob Morane	hello	hello world
Sam Le Pirate	Salut	Salut à Tous



Ensuite, ajoutons notre message `msg` à la liste des messages `Messages` en saisissant ceci dans la console : `Messages.push(msg)`. Vous voyez que la liste des messages se met directement à jour :



# Tuto KnockOut

Bob Hello You !!! hello world

From	Subject	Body
John Doe	First Message	Hello World
Bob Morane	hello	hello world
Sam Le Pirate	Salut	Salut à Tous
Bob	Hello You !!!	hello world

The screenshot shows the Chrome DevTools Sources tab. It displays a message log with the following content:

```

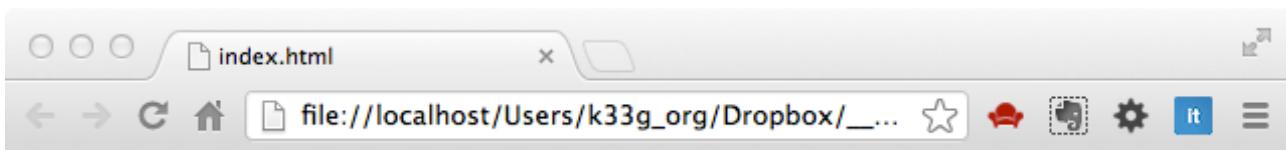
> msg.from("BOBBY").subject("Hello You !!!")
  value : BOBBY
< ▶ Message
  value : Bob
> Messages.push(msg)
  4
>

```

The log entries are timestamped with "index.html:45".



Et enfin, si vous modifiez les données dans les zones de saisie, vous pouvez voir que tout se met à jour même la liste des messages :



## Tuto KnockOut

@K33G\_ORG Hello You !!! hello world

From	Subject	Body
John Doe	First Message	Hello World
Bob Morane	hello	hello world
Sam Le Pirate	Salut	Salut à Tous
@K33G_ORG Hello You !!! hello world		

The screenshot shows the Chrome DevTools Sources tab with a stack trace. The code being executed is:

```

> msg.from("BOBBY").subject("Hello You !!!")
  value : BOBBY
< ▶ Message
  value : Bob
> Messages.push(msg)
  4
  value : @K33G_ORG
>

```

The stack trace indicates the code is from "index.html:45" for each line. Below the sources tab, there is a toolbar with icons for refresh, search, and other developer tools features, followed by tabs for All, Errors, Warnings, and Logs.

### 14.3.8 Conclusion

“Je ne sais pas vous”, mais je trouve **Knockout** particulièrement intéressant :). Si vous souhaitez aller plus loin, faites un tour dans la documentation et tout particulièrement ceci :

- Pour les interaction avec le serveur (ajax & jQuery) :  
<http://knockoutjs.com/documentation/json-data.html>
- Mapping des données :  
<http://knockoutjs.com/documentation/plugins-mapping.html>

Et si vous rêvez de combiner Backbone et Knockout, à voir : **Knockback** :

<http://kmalakoff.github.com/knockback/>

qui fera certainement l'objet d'un chapitre de cet “ouvrage”.

## 14.4 Encore d'autres frameworks MVC (javascript)

ils sont nombreux, mais les “ténors” connus, reconnus et utilisés sont les suivants :

- **Ember.js** <http://emberjs.com> : le concurrent direct “officiel” de Backbone, très très complet, puissant, mais avec une dépendance forte avec le moteur de template Handelbar (un Mustache survitaminé), à suivre de très près. Il est cependant dommage que la gestion des modèles ne soit pas encore en mode stable (c'est un module à part), mais le modèle objet d'Ember.js est évolué, plus strict que celui de Backbone, donc à mon sens plus contraignant, mais cela peut en rassurer d'autres, car il propose un cadre “plus normatif” et permet donc de “forcer” à respecter les normes de développement sur un projet.
- **AngularJS** <http://angularjs.org> : le framework pour Webapp de Google, mais pas complètement MVC (les modèles de sont pas traités) qui met essentiellement l'accent sur le binding des données avec le DOM
- **JavaScriptMVC** <http://javascriptmvc.com> : intéressant (peut-être un des plus anciens), car il essaye de coller le plus possible à ce que des développeurs java connaissent de MVC, mais la mécanique de mise en oeuvre est un peu lourde. Cependant le code est très lisible.

Vous trouverez aussi des frameworks javascript MVC “Client et Serveur”, tels :

- **Batman** <http://batmanjs.org/> : plutôt destiné aux développeurs Coffeescript. Il fournit son propre serveur mais peut être utilisé avec d'autres technologies serveur notamment le Ruby ou même “seul” dans le navigateur. Je trouve la documentation insuffisante, vous êtes parfois obligés d'aller dans le code du framework pour en comprendre le fonctionnement.
- **Matador** <http://obvious.github.com/matador> : celui-ci est essentiellement un framework MVC côté serveur en javascript s'appuyant sur express, il servirait donc plutôt à, par exemple, re-écrire la partie serveur de notre blog. Mais il est intéressant de voir de quelle manière il utilise des frameworks javascript initialement développés pour le navigateur, côté serveur, tel le modèle objet Klass (<https://github.com/ded/klass>).
- **Chaplin** <https://github.com/chaplinjs/chaplin> : qui est une architecture complète autour de Backbone, une fois de plus “programmable” en coffeescript.
- et beaucoup d'autres ...

## 14.5 Conclusion

L'éco-système des frameworks MVC javascript est vaste et il est difficile de faire un choix. L'important est de vous faire votre propre avis, d'utiliser les outils avec vous êtes le plus à l'aise, mais tout en étant sûr que vous avez choisi un framework largement utilisé par d'autres, avec une communauté active. Un dernier petit conseil : le plus souvent les frameworks javascript affiche leur repository sur GitHub (“the place to be”), alors tout le monde peut y être (j'y suis), mais c'est un 1er critère, ensuite un des gros avantages de Github, c'est que vous pouvez voir facilement l'activité autour du framework :

- Qui sont les contributeurs (et à quels autres projets il participent) ?
- Depuis combien de temps le code n'a pas été mis à jour ?

- Combien de personnes sont abonnées au projet ?
- Combien de personnes ont “forké” le projet (dupliqué pour modifications et propositions de modifications, on parle de “pull request”) ?
- Dans le bug tracker, quelle est la qualité des réponses ?
- ...

Par exemple pour Backbone, plus de 10 000 “abonnés”, plus de 1 700 “forks”, c'est peu de dire que mon “jouet préféré” suscite de l'intérêt. Vous pouvez même comparer par rapport à d'autres frameworks java connus pour vous faire une idée. Ayez donc ce réflexe pour tout framework que vous souhaitez utiliser, cela peut vous éviter quelques soucis.

## 15 Backbone et Typescript

//TODO:

Où nous parlerons de Typescript et de sa cohabitation avec Backbone

## 16 Ressources Backbone

//TODO : une liste à faire vivre des ressources intéressantes concernant Backbone

## 17 RestHub Backbone Stack

//TODO : ou comment organiser son code de manière professionnelle