

Contents

1	Préambule & Remerciements	1
1.1	Avertissement	1
2	Présentation de Backbone & rappels MVC	1
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
3	Tout de suite "les mains dans le cambouis"	6
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
4	1er contact ... avec Backbone	24
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

5 Le modèle objet de Backbone	37
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
6 Il nous faut un serveur !	43
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

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 ;)

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 recharge 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 eclipsé par l'apparente simplicité de mise en œuvre des ActiveX (Flash était alors utilisée 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 promet 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. Ceux 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 appréhender en le codant) qui va ordonner les événements et interaction 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.

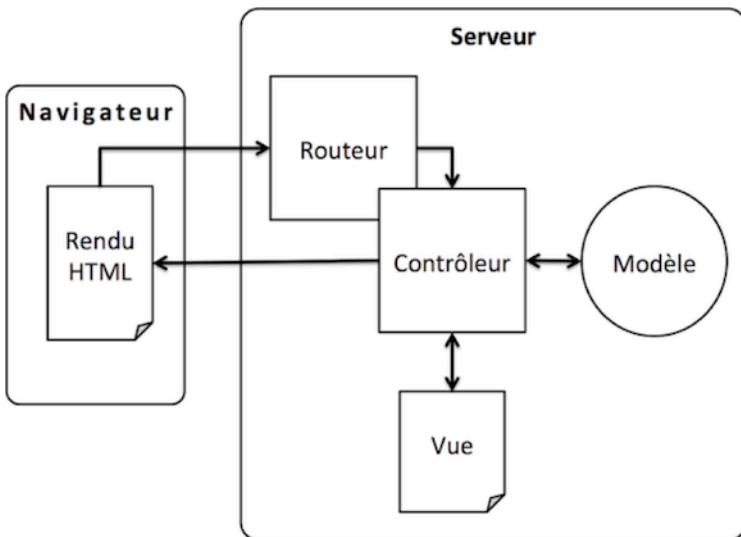


Figure 1-1. MVC Vision “Back-End”

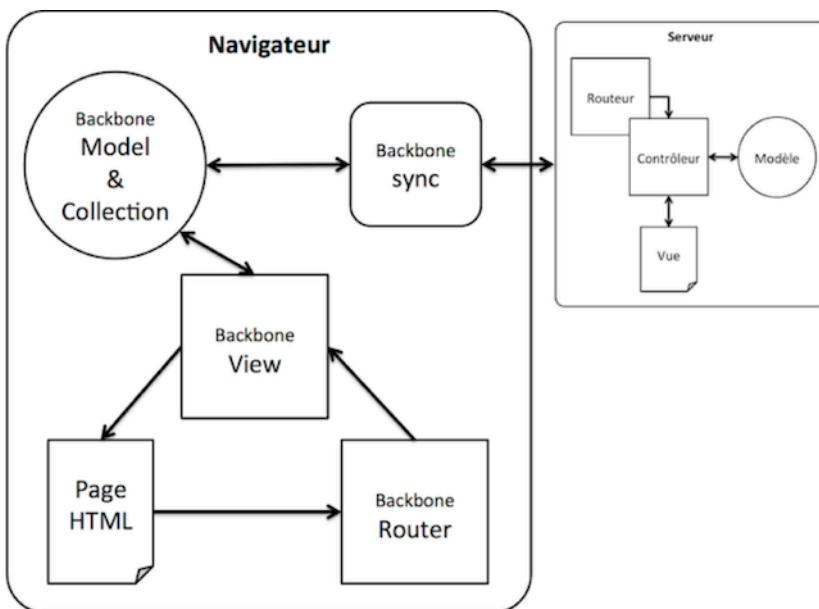


Figure 1-2. MVC Vision “Front-End”

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 IntelliJ, 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.
- Apatana fourni aussi un bon IDE dédié Javascript sur une base Eclipse, mais je trouve qui 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 :

```

<!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>

```

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 */
        }
    </style>

```

```

        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>

        <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>

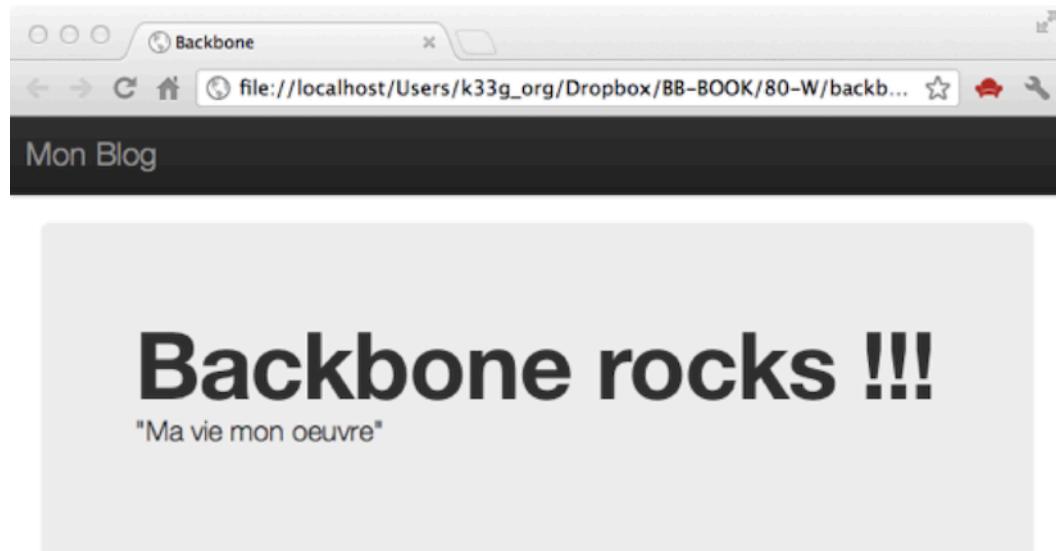
</body>
<!-- === Références aux Frameworks === -->
<script src="libs/vendors/jquery-1.7.2.js"></script>
<script src="libs/vendors/underscore.js"></script>
<script src="libs/vendors/backbone.js"></script>

<!-- === 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

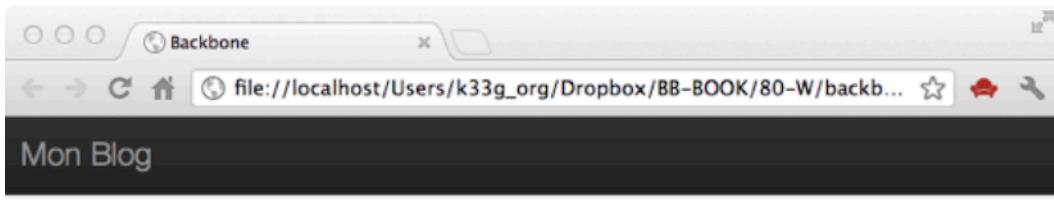
- Backbone et le localStorage
- Backbone.sync : comment ça marche

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 sélectionner “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 :

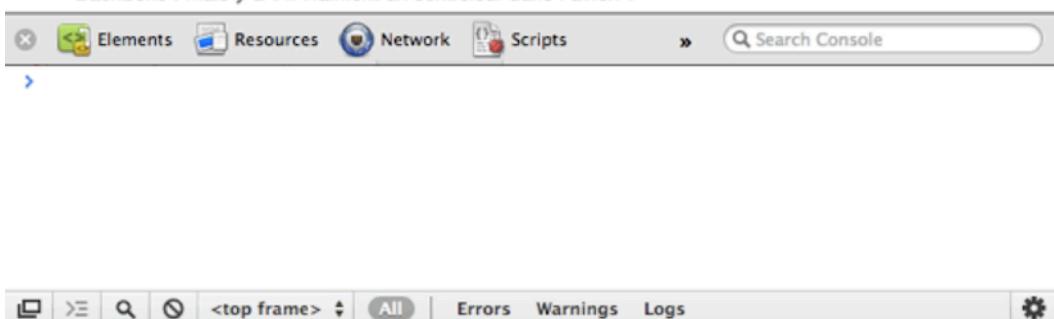


Backbone rocks !!!

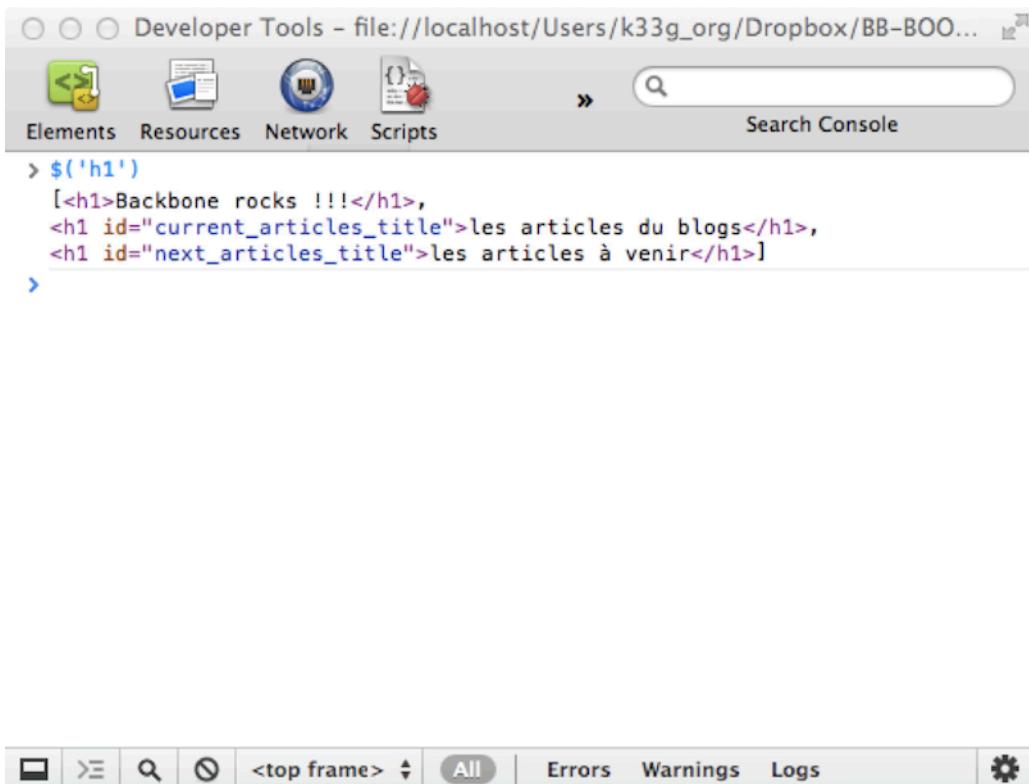
"Ma vie mon oeuvre"

les articles du blogs

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



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 :

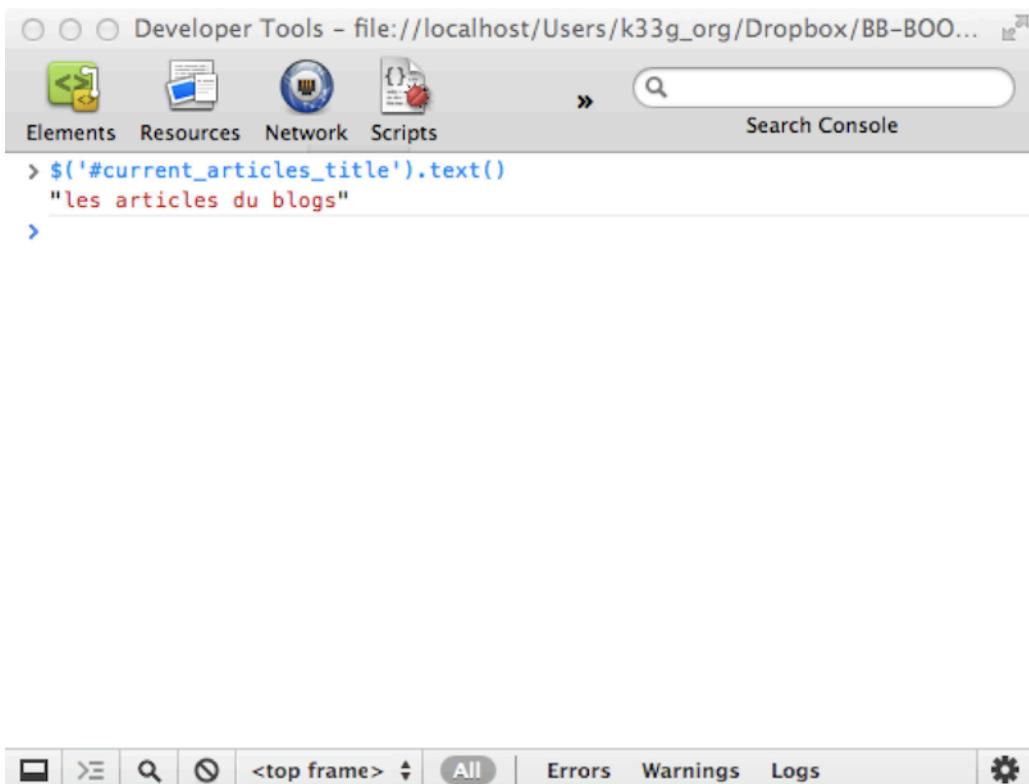


The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The console window displays the following code:

```
> $('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 console, the status bar shows various developer tool icons and tabs: Elements, Resources, Network, Scripts, All, Errors, Warnings, Logs, and a gear icon.

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 :

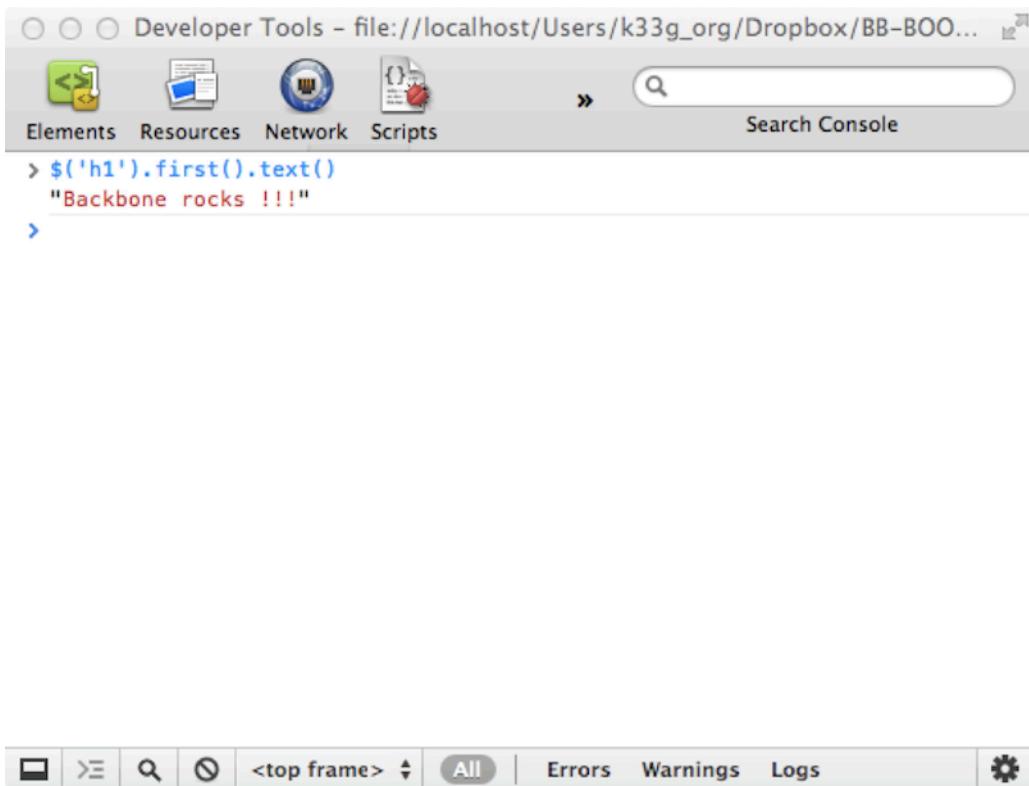


The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The console window displays the following code and its result:

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

Below the console, the status bar shows various developer tool icons and tabs: Elements, Resources, Network, Scripts, All, Errors, Warnings, Logs, and a gear icon.

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"})
```

Developer Tools – file:///localhost/Users/k33g_org/Dropbox/BB-BOO... 

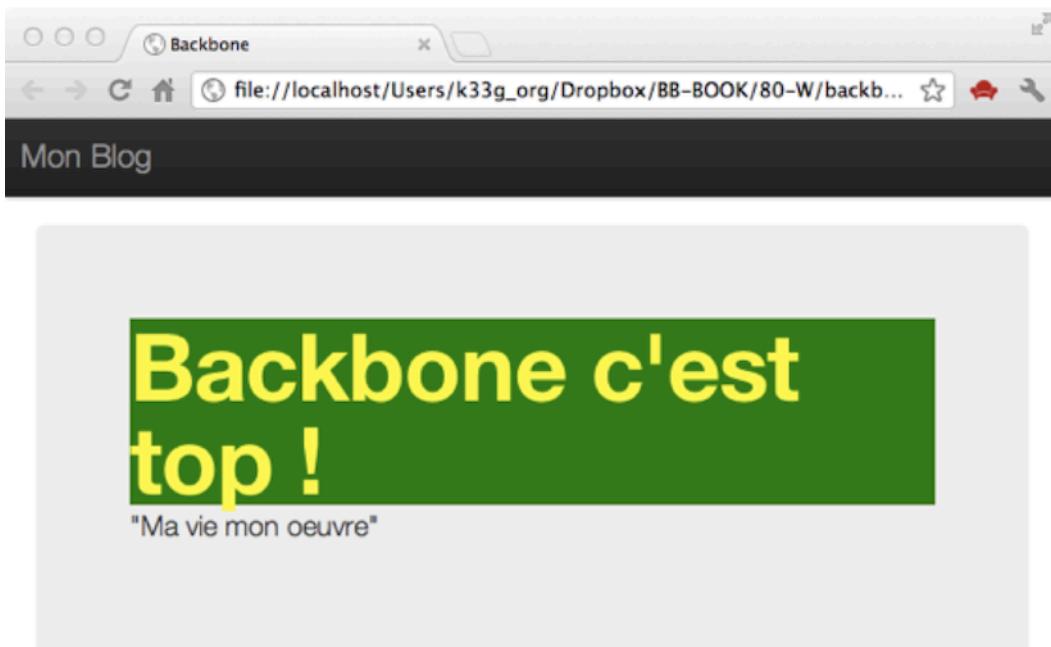
Elements Resources Network Scripts »

```

> $('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>
]
>
```

    <top frame>  | Errors Warnings Logs 



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')`



```

Developer Tools - file:///localhost/Users/k33g_org/Dropbox/BB-BOO...
Elements Resources Network Scripts Search Console

> $('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 :

```

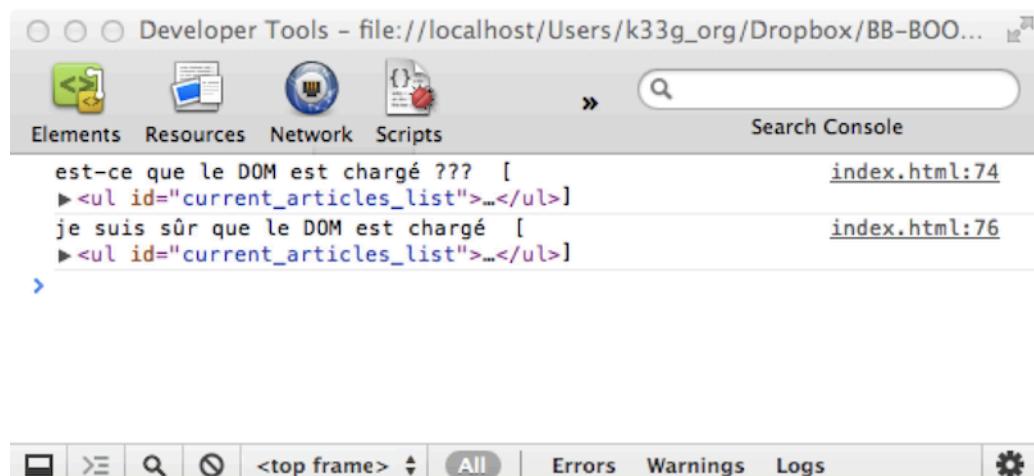
var currArtList = $('#current_articles_list');
currArtList.hide('slow');
currArtList.show('fast');

```

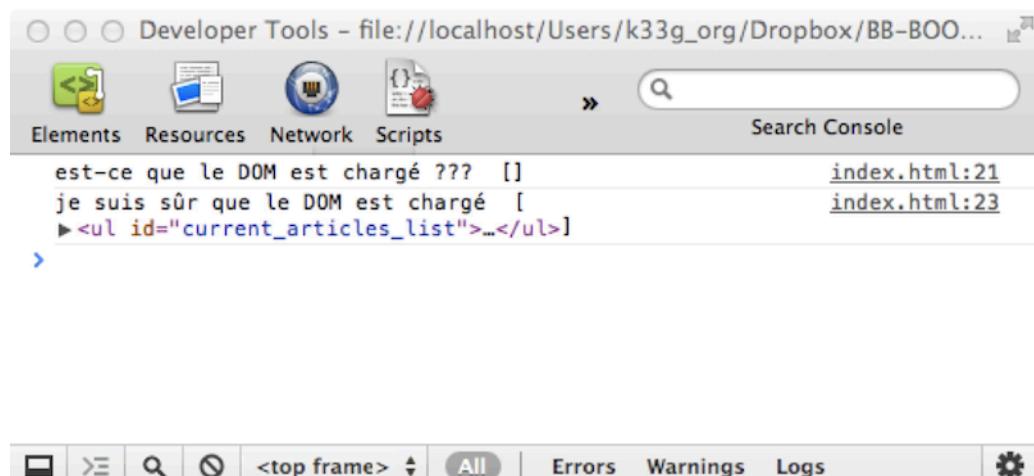
Soyez sûr que les éléments de votre page sont tous chargés : Il est intéressant (indispensable) d'avoir la garantie que son code javascript n'est exécuté qu'une seule fois la page HTML chargée dans son entièreté, surtout si ce code accède à des éléments du DOM. jQuery a une fonction pour ça : `$(document).ready(handler)`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

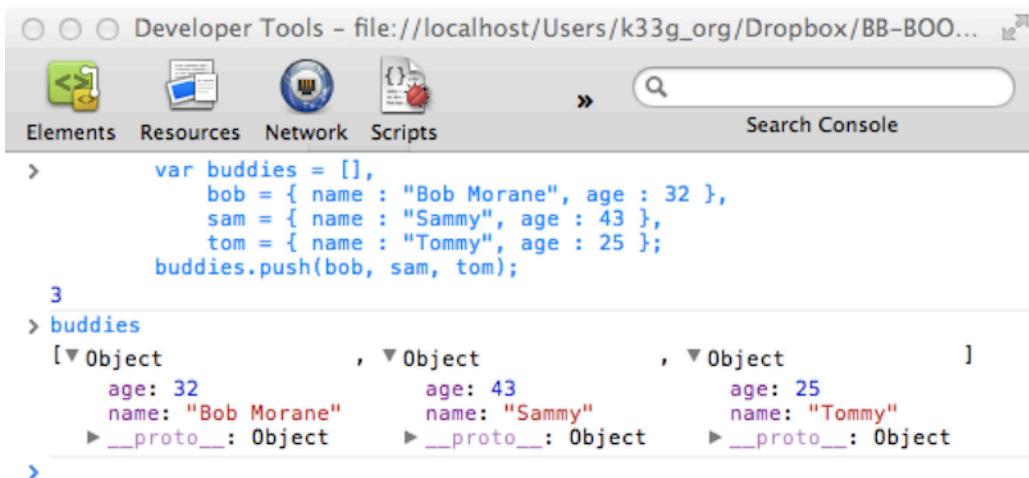
3.5.1 Quelques exemples d'utilisations

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 DevTools.js tab of the Chrome Developer Tools. At the top, there are tabs for Elements, Resources, Network, Scripts, and a search bar labeled "Search Console". Below the tabs, a code block is shown:

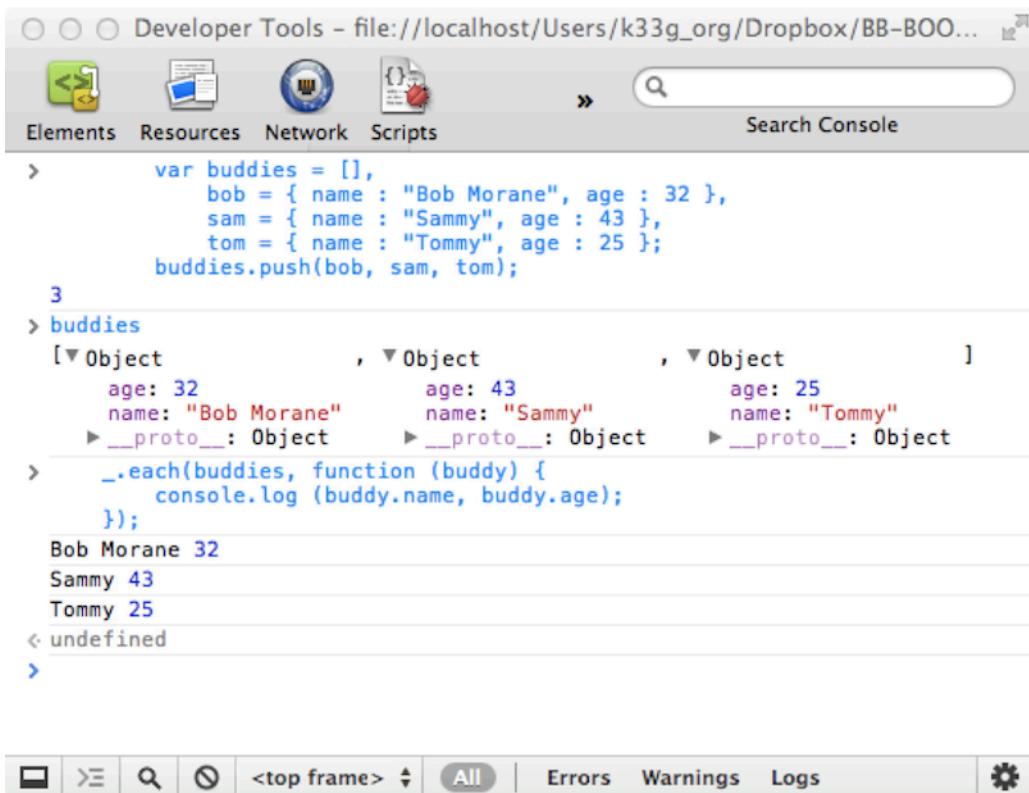
```
> var buddies = [],
    bob = { name : "Bob Morane", age : 32 },
    sam = { name : "Sammy", age : 43 },
    tom = { name : "Tommy", age : 25 };
buddies.push(bob, sam, tom);
3
> buddies
[▼ Object , ▼ Object , ▼ Object ]
  age: 32          age: 43          age: 25
  name: "Bob Morane"  name: "Sammy"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

At the bottom of the DevTools interface, there is a toolbar with icons for back, forward, search, and refresh, 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 :



```

Developer Tools - file:///localhost/Users/k33g_org/Dropbox/BB-BOO...
Elements Resources Network Scripts » Search Console

> 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
>

```

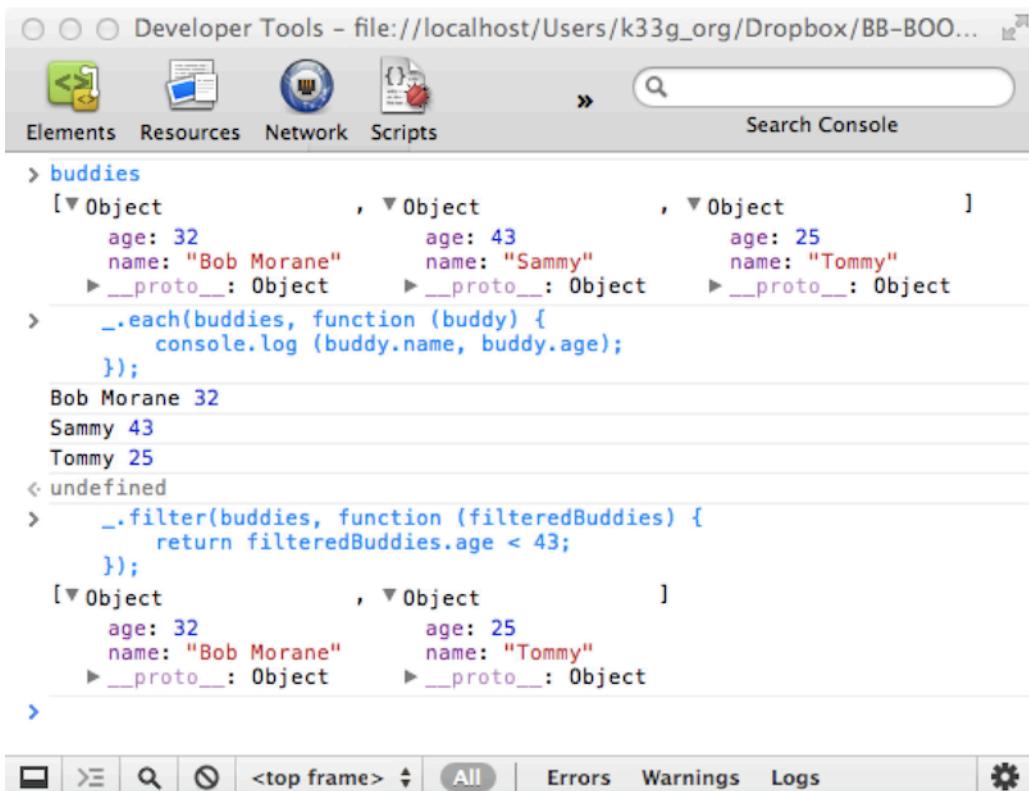
The screenshot shows the Chrome Developer Tools Console tab. At the top, there are tabs for Elements, Resources, Network, and Scripts. Below the tabs is a search bar labeled "Search Console". The main area contains a command-line interface where JavaScript code is being run. The code defines an array "buddies" containing three objects (bob, sam, tom) with properties name and age. It then uses the underscore.js library's "each" method to iterate over the array and log each buddy's name and age to the console. The output shows the names and ages of Bob Morane (32), Sammy (43), and Tommy (25). At the bottom of the console window, there is a toolbar with various icons and a status bar indicating "All" errors, warnings, and logs.

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 (``) 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>";

```

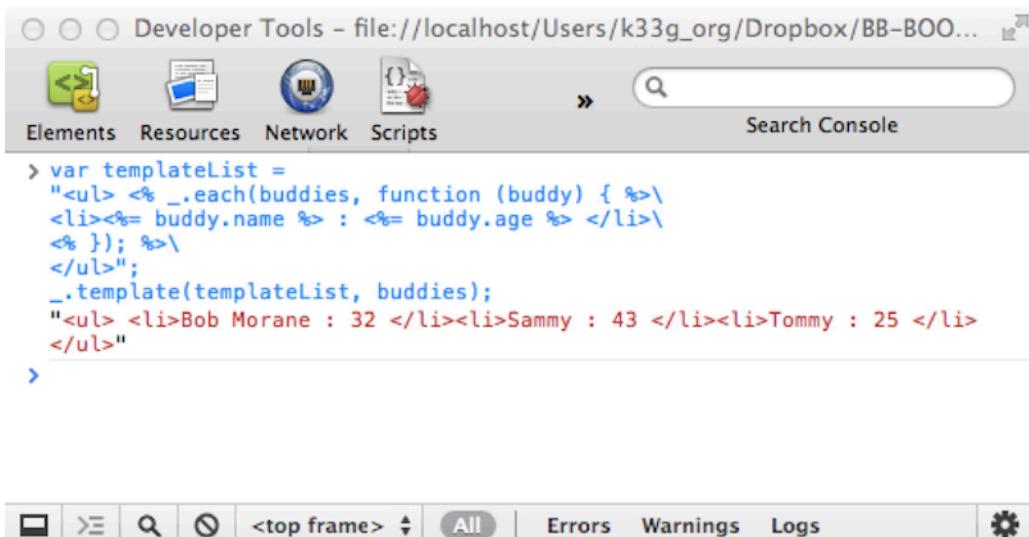
Que nous utiliserons de cette façon (nous passons à la méthode le template et les données):

```

_.template(templateList, buddies);

```

Pour le résultat suivant :



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

The screenshot shows the Network tab of the Chrome Developer Tools. It displays a Backbone template for a list of buddies. The template uses underscore.js's `each` function to iterate over a collection of `buddies`, rendering their `name` and `age` into an `ul` list. The code is shown in blue, while the rendered output is in red.

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 ---*/

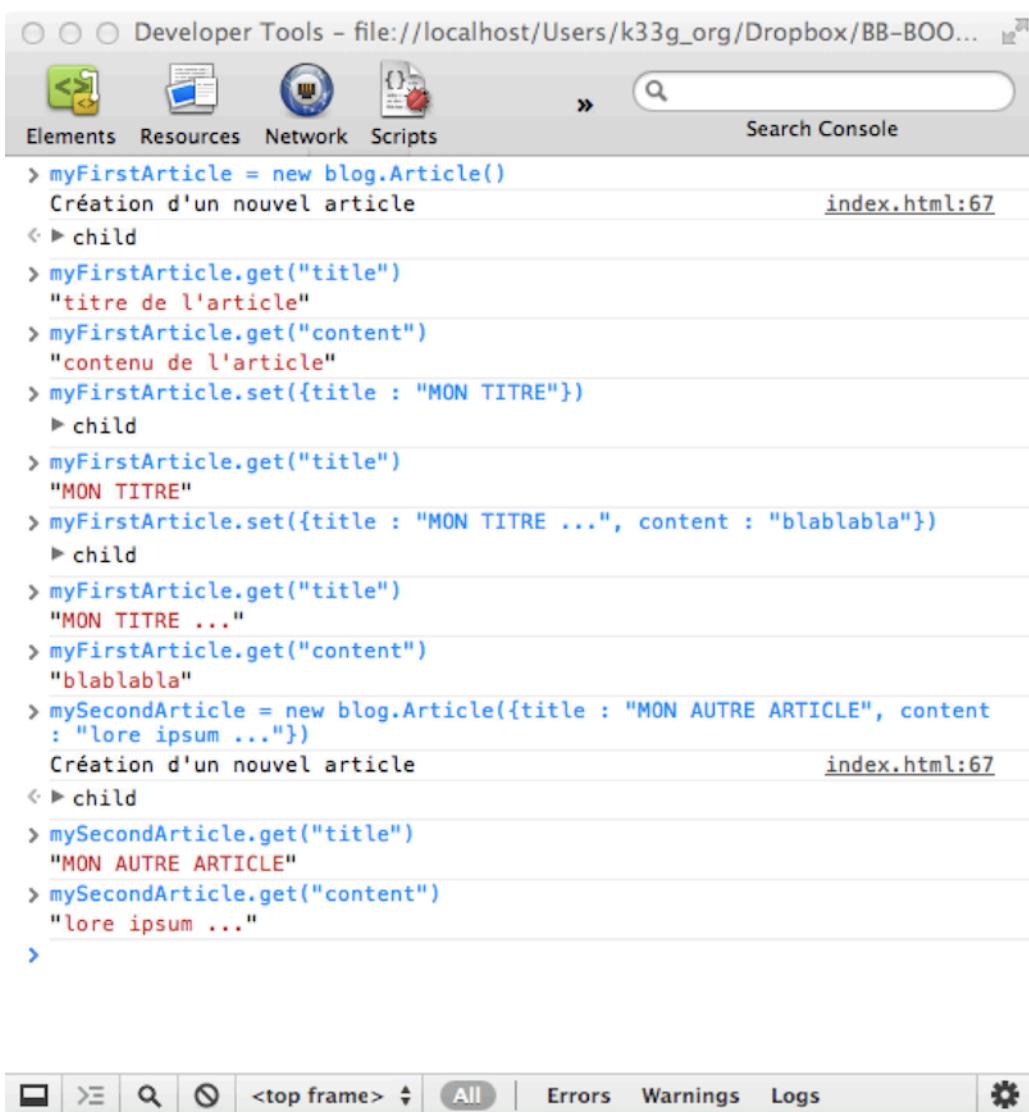
    // une "sorte" de classe Article
    blog.Article = Backbone.Model.extend({
        //les valeurs par défaut d'un article
        defaults : {
            title : "titre de l'article",
            content : "contenu de l'article",
            publicationDate : new Date()
        },
        // s'exécute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

});

</script>
```

Sauvegarder, relancer 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 Chrome Developer Tools Network tab. 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()
```

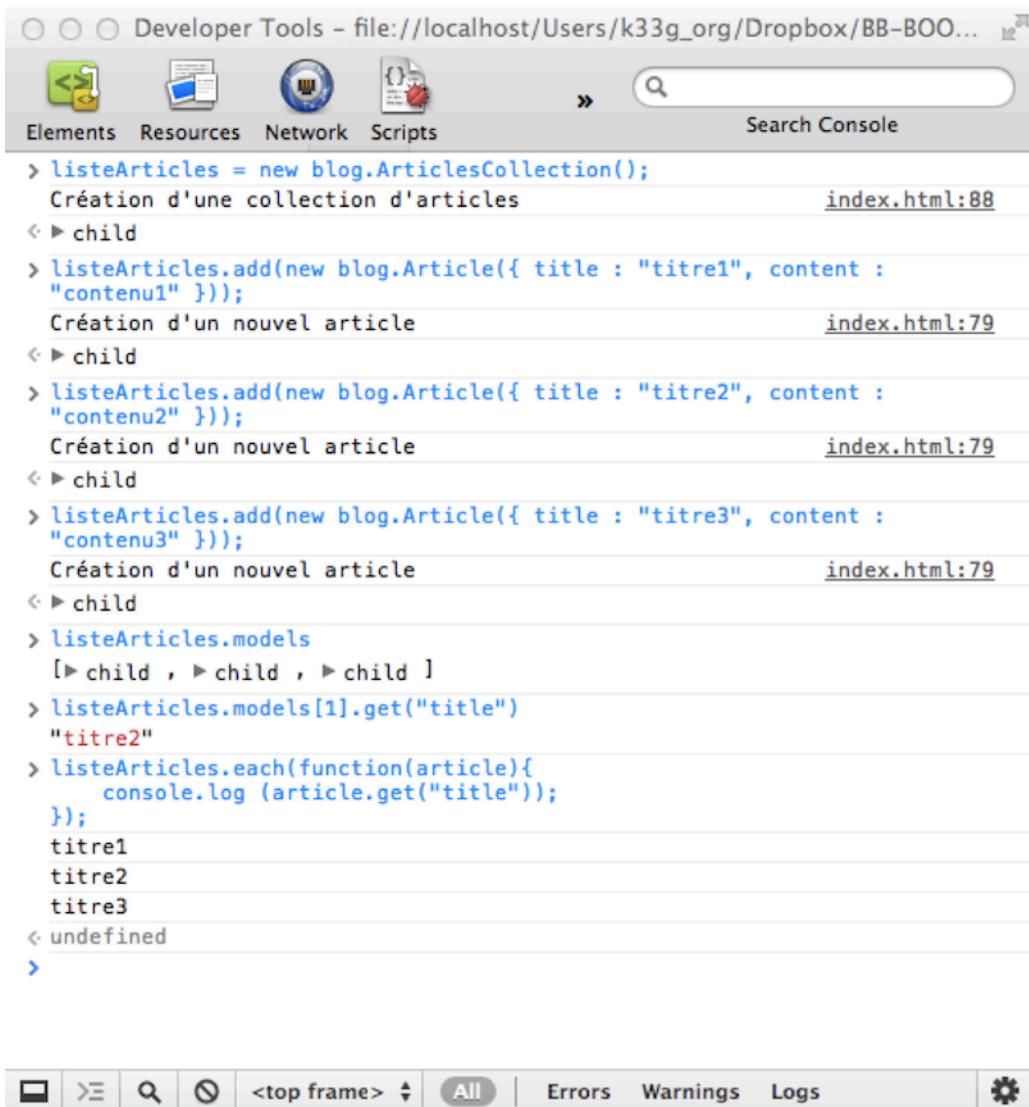
- Ajout d'articles à la collection :

```
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:

- Request 1: `listeArticles = new blog.ArticlesCollection();` - Creation of a new collection of articles at index.html:88.
- Request 2: `listeArticles.add(new blog.Article({ title : "titre1", content : "contenu1" }));` - Creation of a new article titled "titre1" with content "contenu1" at index.html:79.
- Request 3: `listeArticles.add(new blog.Article({ title : "titre2", content : "contenu2" }));` - Creation of a new article titled "titre2" with content "contenu2" at index.html:79.
- Request 4: `listeArticles.add(new blog.Article({ title : "titre3", content : "contenu3" }));` - Creation of a new article titled "titre3" with content "contenu3" at index.html:79.
- Request 5: `listeArticles.models` - Returns an array of three models: [titre1, titre2, titre3].
- Request 6: `listeArticles.models[1].get("title")` - Returns the title of the second model: "titre2".
- Request 7: `listeArticles.each(function(article){ console.log (article.get("title"));});` - Logs the titles of all three models: titre1, titre2, titre3.

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 ?

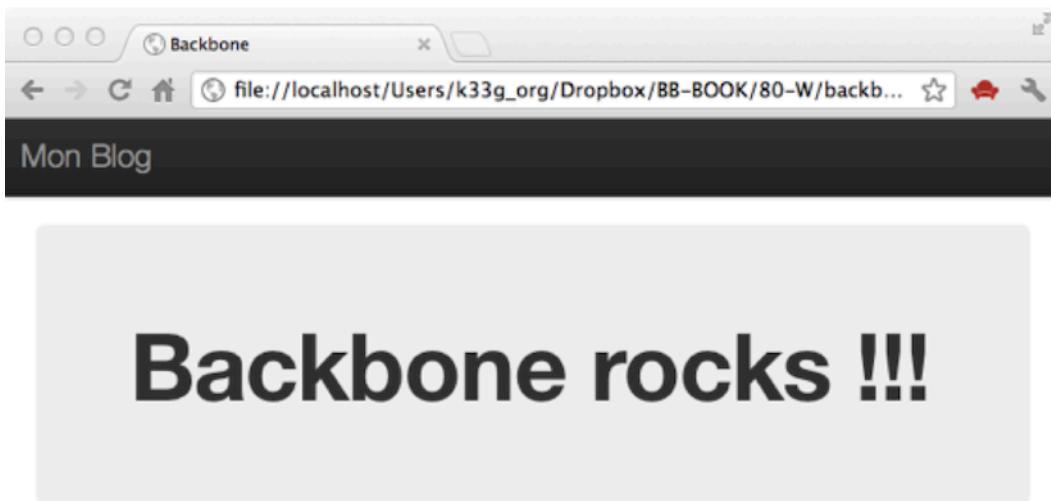
Eh bien, nous avons défini une vue avec :

- Une propriété `el` (pour élément) à laquelle on “attache” le `<div>` dont l'id est “`articles-collection-contai`
- 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 définit 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 :

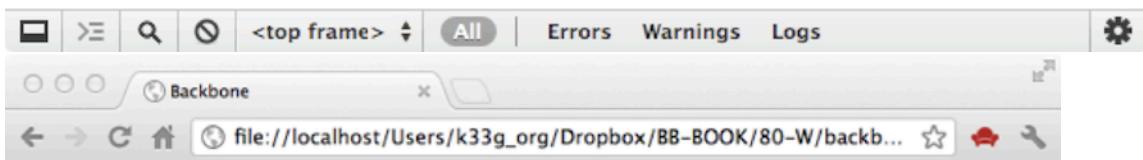
Developer Tools – file:///localhost/Users/k33g_org/Dropbox/BB-BOOK/80-W/...

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.sex);
  },
  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.sex);
  },
  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) :

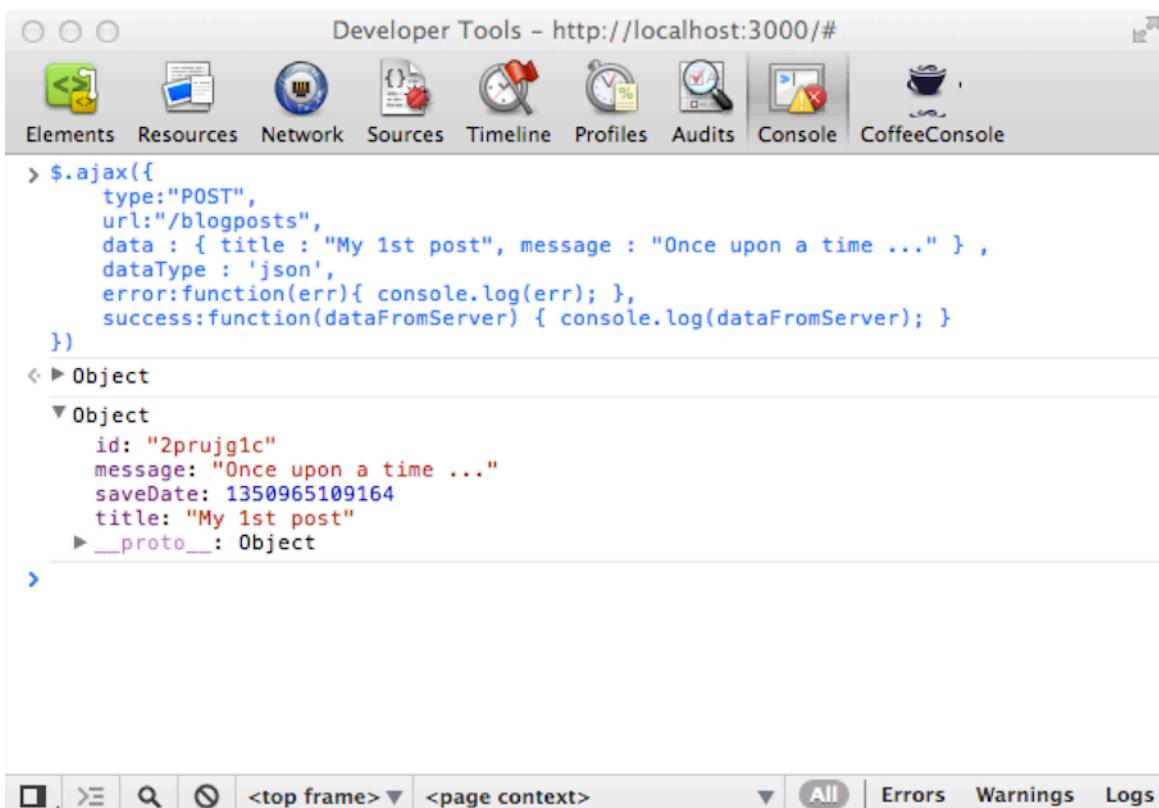
Requête http de type POST :

```

$.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 :

```

1. Default (node)
27 Jun 19:43:36 - [nodemon] starting `node app.js`
Express app started on port 3000
POST CREATE  { title: 'My 1st post', message: 'Once upon a time ...' }

```

6.4.2 Obtenir tous les enregistrements

Dans la console tapez ceci :

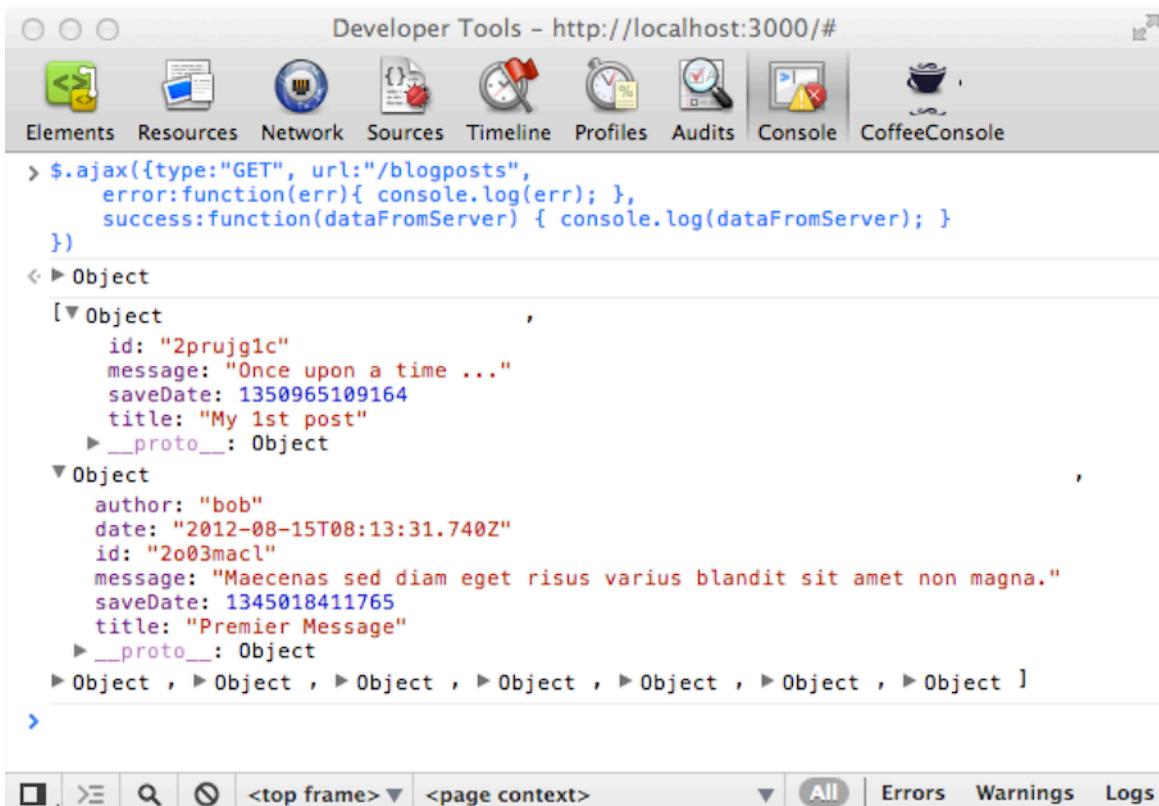
Requête http de type GET :

```

$.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 :



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The title bar says 'Developer Tools - http://localhost:3000/#'. Below the tabs are icons for Elements, Resources, Network, Sources, Timeline, Profiles, Audits, Console, and CoffeeConsole. The console log output is as follows:

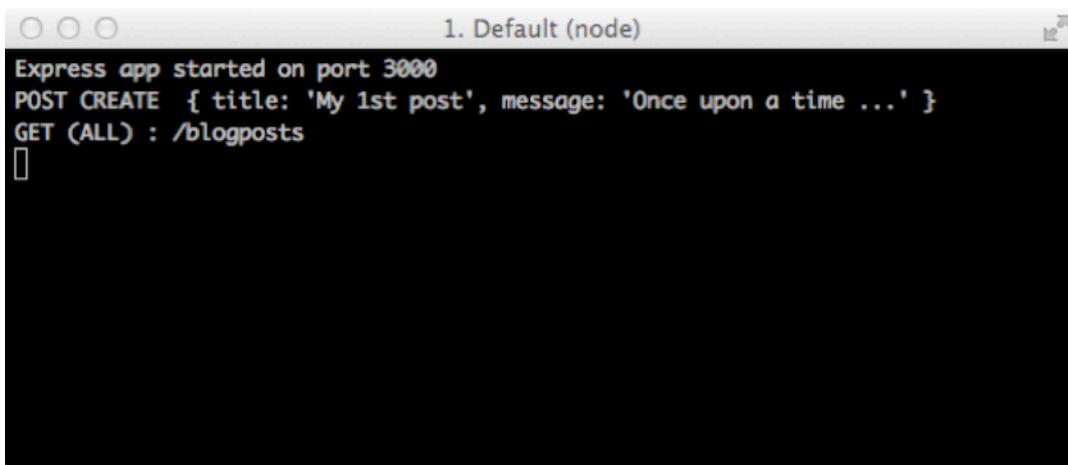
```

> $.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 ]
>

```

At the bottom of the console are buttons for **All**, Errors, Warnings, and Logs.

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



The terminal window shows the following log output:

```

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

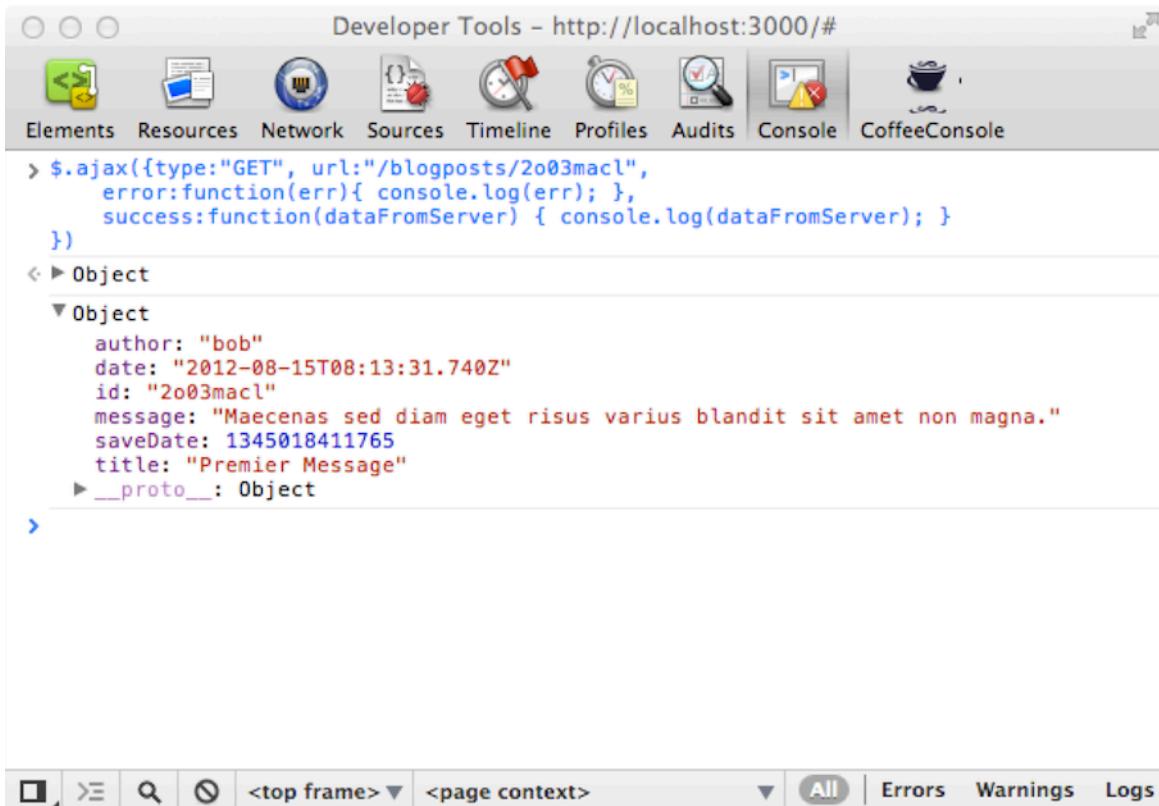
Requête http de type GET :

```

$.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 :

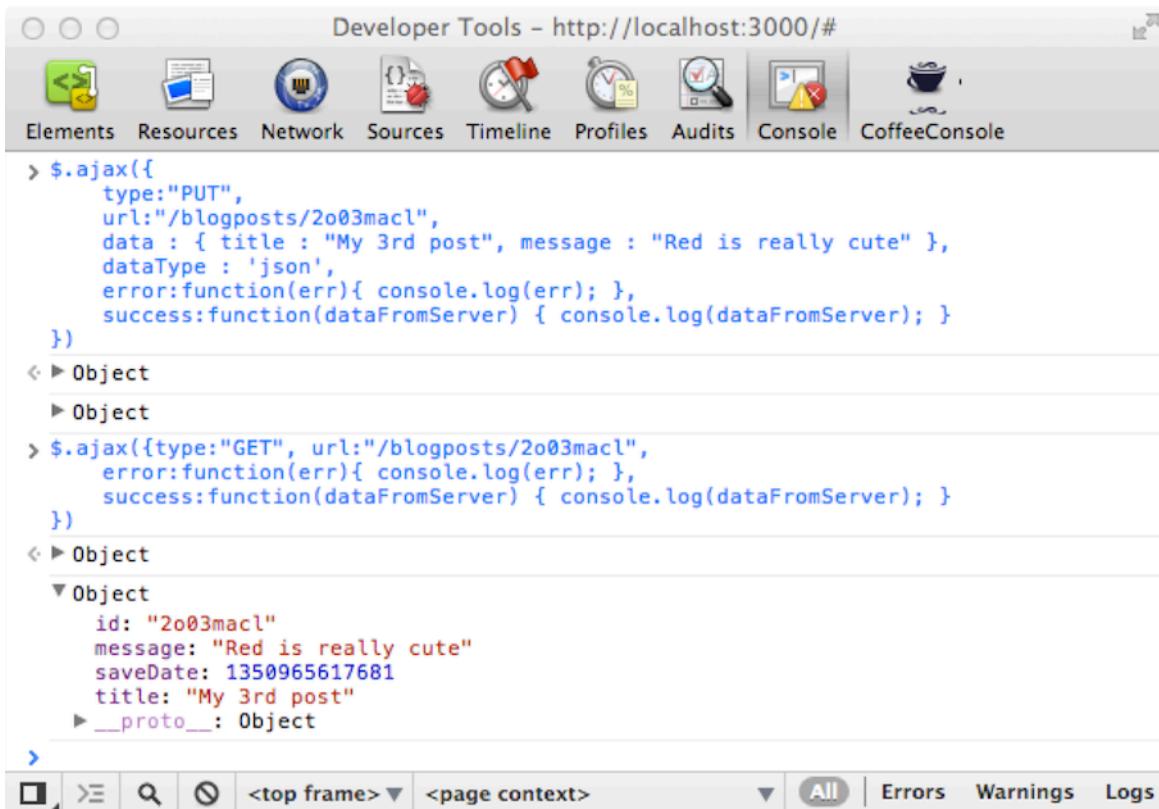
```
$.ajax({
  type:"PUT",
  url:"/blogposts/2o03macl",
  data : { title : "My 3rd post", message : "Red is really cute" },
  dataType : 'json',
  error:function(err){ console.log(err); },
  success:function(dataFromServer) { console.log(dataFromServer); }
})
```

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 Chrome Developer Tools interface with the 'Console' tab selected. At the top, there's a toolbar with icons for Elements, Resources, Network, Sources, Timeline, Profiles, Audits, and CoffeeConsole. Below the toolbar, the console output is displayed:

```

> $.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 filters for All, Errors, Warnings, and Logs.

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”*) :

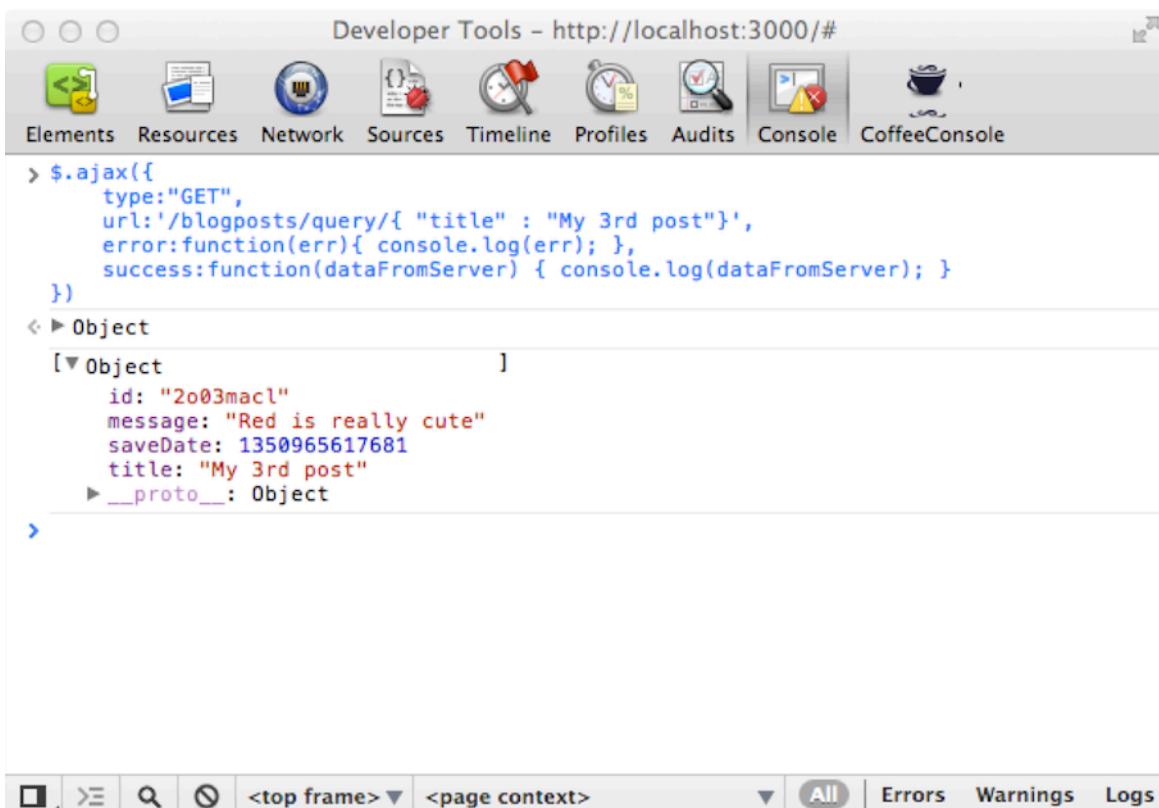
Requête http de type GET :

```

$.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 :

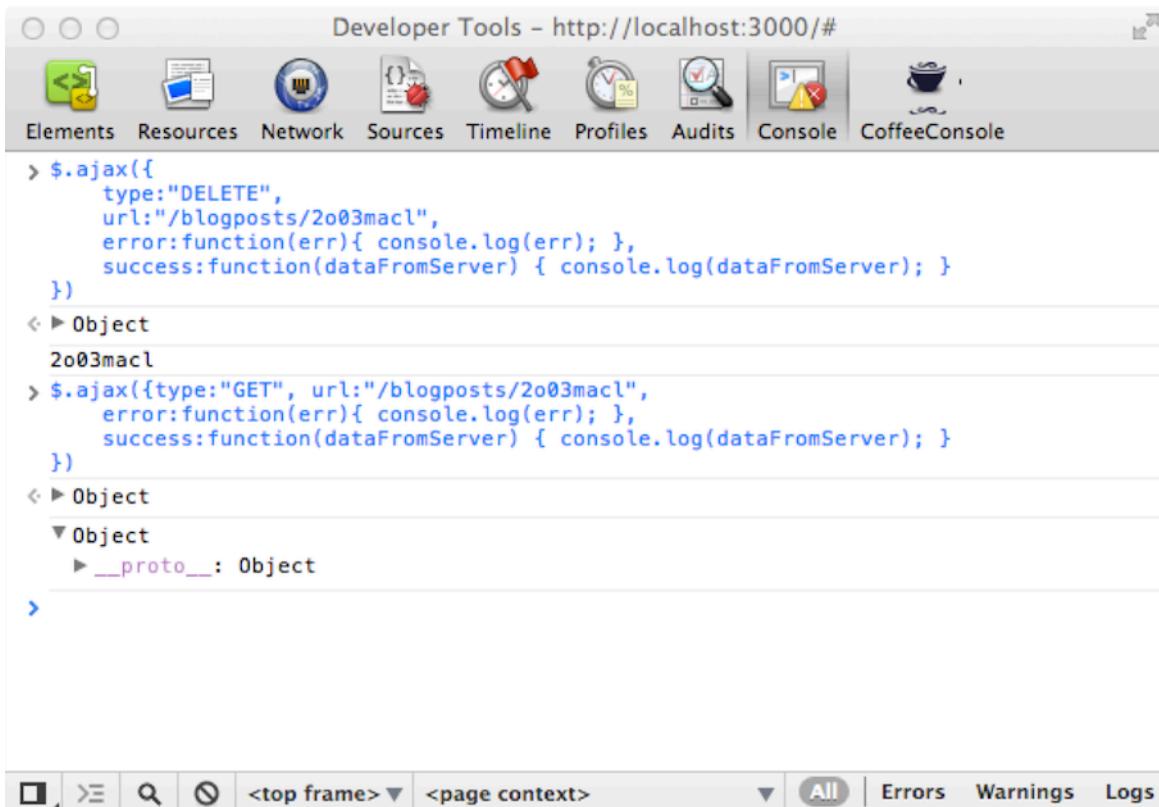
Requête http de type GET :

```

$.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 title bar says 'Developer Tools - http://localhost:3000/#'. Below the title bar is a toolbar with icons for Elements, Resources, Network, Sources, Timeline, Profiles, Audits, Console, and CoffeeConsole. The main area 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
>

```

At the bottom of the developer tools window, there is a toolbar with icons for back, forward, search, and other developer tools controls. To the right of the toolbar, there are tabs for 'All', 'Errors', 'Warnings', and 'Logs', with 'All' being the active tab.

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).