Bien choisir son Frameworks JavaScript: AngularJS, Knockout, Ember, Backbone.

Choisir son Framework front end n'est jamais une chose facile, ce choix impactera le temps de chargement, la vitesse de développement et la maintenabilité du code source de votre site web. Il en existe des dizaines, dans cet article nous allons nous restreindre aux 4 plus fameux afin de vous aider à faire le bon choix.

Auteur : Feltz Ludovic (http://feltz.fr/), Consultant SoftFluent (ludovic.feltz@softfluent.com)

Plusieurs critères sont à prendre en compte dans le choix d'un Framework, son poids par exemple définira le temps de chargement des pages ce qui est un critère important pour les applications mobiles, sa facilité d'utilisation, sa documentation et sa communauté active permettent de réduire le temps de développement. Nous allons commencer par un bref historique puis nous allons plonger dans le vif du sujet avec un exemple simple que nous allons reproduire en utilisant chacun de ces Frameworks.

Cet article n'a pas pour but de décrire le fonctionnement précis de chaque Frameworks mais seulement de les présenter à travers un exemple interactif simple afin d'en comprendre les différences. Nous n'allons pas exemple pas montré comment communiquer avec le serveur et sauvegarder les données dans une base.

Utiliser un frameworks?

Pourquoi utiliser un Frameworks JavaScript à la place (ou en plus) du code serveur (ASP, PHP, …).

Histoire :

AngularJS: Apparu en 2009 sous le nom de GetAngular il est utilisé par Misko Hevery un des ingénieurs qui l'a développé pour recréer une application web qui représentait plus de 17 milles lignes de code. En 3 semaines il est parvenu à réduire ce nombre à seulement 1000 lignes ce qui a convaincu Google de sponsoriser ce projet ce qui a créé sa renommée. Son but est de simplifier le développement et les tests de site web suivant le pattern MVC en ajoutant du vocabulaire au code HTML de votre application.

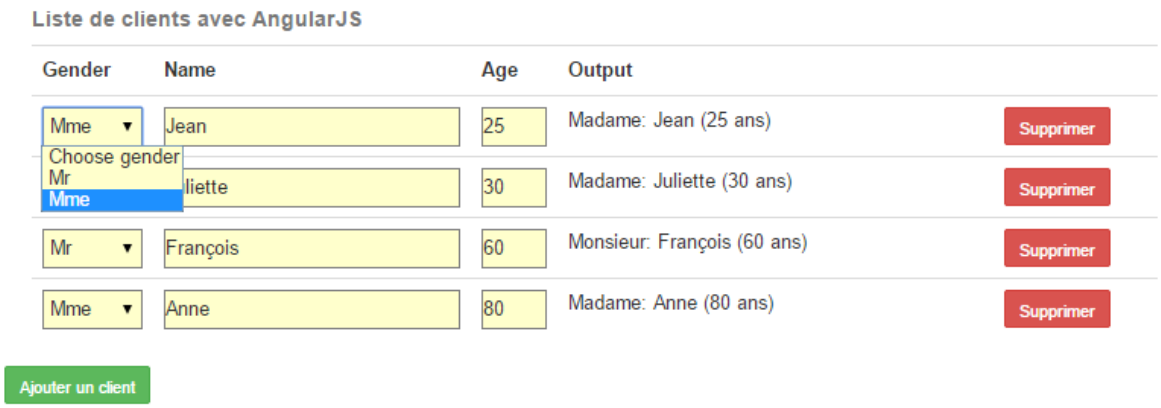
Knockout : Né en 2010 et maintenu comme un projet open source par Steve Sanderson, employé chez Microsoft. Simplifie l'utilisation du JavaScript en appliquant le pattern MVVM.

Ember : Développé en 2007 par SproutIt puis par Apple il est finalement forké en 2007 par Yehuda Katz, un des principaux contributeurs de JQuery et Ruby. Il est utilisé par des grands noms tels que Yahoo, Groupon et ZenDesk. Il est basé sur le pattern MVC. Son avantage est de permettre le développement d'applications d'une seule page grâce à son système de route.

Backbone : Crée en 2010 par Jeremy Ashkenas qui à aussi participer au développement de Coffee Script, il est très léger ce qui lui a permis de se faire un nom parmi les autres Frameworks JavaScript. Il repose principalement sur le pattern MVP (model view presenter).

Exemple similaire

Pour comparer ces différents Frameworks nous allons nous baser sur un exemple d'application simple que nous allons reproduire en utilisant les 4 Frameworks puis nous commenterons le code produit.



Cet exemple nous permet de modifier les informations d'une liste de clients, d'en ajouter et d'en supprimer. Un affichage formaté nous permet de vérifier que lors d'une modification de notre model les modifications sont affichés directement sans rafraichir la page.

Pour peupler nos modèles nous allons à chaque fois réutiliser le même jeu de données :

```
var Genders = [
    {
        id: 0,
        key: "Mr",
        value: "Monsieur"
    },
    {
        id: 1,
        key: "Mme",
        value: "Madame"
    }
]


var ClientsList = [
    {
        id: 0,
        gender: Genders[0],
        name: "Jean",
        age: 25,
    },
    {
        id: 1,
        gender: Genders[1],
        name: "Juliette",
        age: 30,
    }
]
```

AngularJS :

AngularJS utilise le pattern MVC, nous allons rapidement décrire le fonctionnement de ce pattern puis nous allons créer notre application.

Modèle : Le modèle décrit la logique métier de notre application. Il contient les données ainsi que les méthodes de manipulation de celles-ci.

Vue : est la façon d'afficher les données de notre application, c'est le code HTML qui est en charge de formater les données du modèle et d'envoyer les évènements tel qu'un clique sur un bouton au Controller.

Controller : Reçoit les informations de l'utilisateur provenant de la vue les traites et les envoie au modèle. Il renvoi ensuite les informations à la vue pour les afficher. Il agit comme le coordinateur entre le modèle et la vue.

Templating simple

Commençons par créer une application AngularJS très simple. Nous avons donc le contenu de notre page HTML (index.html):

```
<html ng-app="myApp">
<head>
    <title>AngularJS</title>

    <!-- Angular 1.3.0 (no dependency) -->
    <script src="/script/angular.js"></script>

    <!—  Notre controller -->
    <script src="/script/controllers.js"></script>

</head>
<body>
    <div class="container" ng-controller="AppController as controller">
        <!—
             Corps de la vue
       -->
    </div>
</body>
```

Et ce code javascript (script/controllers.js):

```
var app = angular.module('myApp', []);
app.controller("AppController", ['$scope', function ($scope) {
      /*
       *  Corps du controller
       */
}]) ;
```

On remarque que angularJS définit ses propre balise HTML commençant par ng-*. La première se trouve directement dans la balise HTML ouvrante, elle permet de définir la racine de notre application donnant ainsi la possibilité au développeur de dire à angular d'utiliser toute la page ou seulement une portion de celle-ci.

La deuxième balise que l'on voit apparaitre est ng-controller elle va nous fournir le contexte de notre application et va nous permettre de lier notre vue avec notre modèle.

Angular encourage à utiliser le pattern MVC pour structurer le code.

Affichage des données

Ensuite nous allons peupler notre modèle avec le jeu de donnée décrit précédemment et nous allons ajouter une directive afin d'avoir un formatage du texte. Pour cela nous ajoutons simplement les variables clients, genders et une directive à notre Controller de la façon suivante:

```
app.controller("AppController", ['$scope', function ($scope) {
    $scope.clients = ClientsList
    $scope.genders = Genders;
}])
.directive('clientFormatted', function ()
    return {
        template: '{{client.gender.value}}: {{client.name}} ({{client.age}} ans)'
    };
});
```

Enfin nous modifions notre vue afin d'afficher tous les clients dans un tableau, le corps de notre page devient donc:

```
<body>
    <div class="container" ng-controller="AppController as controller">

        <!-- Corps de l'application -->

        <table class="table">
            <caption>Liste de clients avec AngularJS</caption>
            <tr>
                <th>Gender</th>
                <th>Name</th>
                <th>Age</th>
                <th>Output</th>
                <th></th>
            </tr>
            <tr ng-repeat="client in clients" title="{{client.ID}}">
                <td>
                    <select ng-model="client.gender" ng-options="gender.key for gender
in genders">
                        <option value="">Choose gender</option>
                    </select>
                </td>
                <td><input type="text" ng-model="client.name"></td>
                <td><input type="text" ng-model="client.age"></td>
                <td><client-formatted></client-formatted></td>
                <td><button type="button" ng-
click="removeClient(client);">Supprimer</button></td>
            </tr>
        </table>
        <button type="button" ng-click="addClient();">Ajouter un client</button>
    </div>
</body>
```

On voit apparaitre ici 4 nouvelles balises appartenant à AngularJS. `ng-repeat` s'apparente à une « **foreach** ». Pour chaque client de notre controller on crée une balise `<tr>` dont le titre sera l'ID de ce client. Les accollades : **{{expression}}** permette à Angular d'effectuer une liaison avec le modèle.

`ng-model` permet d'effectuer une liaison dans les deux sens (two way binding) avec notre modèle. C'est-à-dire qu'il permet de récupérer la valeur du modèle et que dès lors qu'une valeur est modifiée le modèle est automatiquement mis à jour !

`ng-options` permet de peupler notre liste déroulante et de définir ce qui sera affiché. Ici on affichera la clef de chaque « **gender** »

L’ajout et la suppression des données

Enfin `ng-click` permet de spécifier le nom de la fonction qui sera appelé lors d’un clique sur le bouton. Il ne nous reste plus qu’à ajouter nos fonctions « **removeClient** » et « **addClient** » à notre controller :

```
$scope.addClient = function () {
    $scope.clients.push({ gender: Genders[0], name: "Inconnu", age: 0 });
};

$scope.removeClient = function (client) {
    index = $scope.clients.indexOf(client);
    $scope.clients.splice(index, 1);
};
```

addClient ajoute seulement un nouveau client à notre collection grâce à la fonction JavaScript **push()** et **removeClient** supprime un client avec la fonction **splice**.

Simple n’est-ce pas ? Passons maintenant à Knockout, nous verrons que son comportement est très proche de celui d’AngularJS.

Knockout :

Knockout utilise le pattern MVVM. Une petite explication de ce pattern qui permet de garder une organisation simple d'une application graphique:

- Le « **model** »: indépendant de l'interface graphique c'est lui qui stocke les informations (généralement en base de données). Dans cet article nous n'aborderons pas la persistance des données en base.
- Le « **view model** » : est une représentation des données en code seulement (ici en Javascript), par exemple une liste de donnée avec des fonctions permettant d'ajouter ou supprimer des éléments, etc … Ce sont des données non sauvegardé sans lien avec l'interface graphique puisqu'il n'y a aucune notion d'affichage ou de boutons.
- La « **view** » : affiche les informations du « **view model** », envoi des commandes tel qu'un clique sur un bouton, et est automatiquement mis a jour dès qu'il y a un changement sur le « **view model** »

Avec Knockout il faut lier manuellement le modèle à la vue. Commençons donc par modifier notre modèle, Genders n'étant pas modifiable sa structure ne change pas :

Templating simple

```
var Genders = [
    {
        id: 0,
        key: "Mr",
        value: "Monsieur"
    },
    {
        id: 1,
        key: "Mme",
        value: "Madame"
    }
]
```

Nous modifions donc le modèle de **Client**, on crée donc un constructeur avec la structure suivante :

```
var Client = function (id, gender, name, age) {
    this.id = id;
    this.gender = ko.observable(gender);
    this.name = ko.observable(name);
    this.age = ko.observable(age);

    this.clientFormatted = ko.computed(function () {
        return this.gender().value + ": " + this.name() + " (" + this.age() + " ans)";
    }, this);
}
```

Knockout a besoin de savoir quels champs du **view model** observer afin de notifier la ou les **vue** des changements, pour cela on utilise **ko.observable()** qui prend en paramètre la valeur par défaut.

Comme avec AngularJS on veut avoir une sortie formatée qui affiche les informations de notre Client. Pour cela on utilise **ko.computed(function(){…}).** Permet à knockout de savoir que cette propriétée dépend d'un ou plusieurs **observable** et donc qu'il doit la mettre en jours lorsqu'une des dépendances est modifié.

Passons maintenant à notre « View model ». On crée donc un objet contenant notre liste de clients:

```
var ViewModel = function () {
    var self = this;

    self.clients = ko.observableArray([
        new Client(0, Genders[0], "Jean", 25),
```

```
        new Client(1, Genders[1], "Juliette", 30),
    ]);
}
```

On remarque ici que notre tableau est un « **observableArray** ». permet de suivre l'état de notre collection, ce qui sera utile lorsque nous implémenterons l'ajout et la suppression de nouveau clients dans notre collection.

Enfin il faut activer knockout afin d'associer notre « vue » à notre « vue model », il suffit pour ça d'ajouter la ligne suivante à la fin de la page html contenant notre vue :

```
ko.applyBindings(new ViewModel());
```

Affichage des données

Maintenant que notre « vue modèle » est correctement construit passons à la vue. On commence par inclure knockout dans le header de notre document:

```
<html>
<head>
    <title>Knockout</title>

    <!-- Knockout 3.2.0 -->
    <script src="/script/knockout.js"></script>

    <!—  Notre viewModel  -->
    <script src="/script/viewModel.js"></script>
</head>
```

Ensuite dans le corps de notre vue

```
<body>
    <div class="container">
        <table class="table">
            <caption>Liste de clients</caption>
            <tr>
                <th>Gender</th>
                <th>Name</th>
                <th>Age</th>
                <th>Output</th>
                <th></th>
            </tr>
            <tbody data-bind="foreach: clients">
                <tr>
                    <td>
                        <select data-bind="options:Genders, optionsText:'key',
optionscaption:'Choose gender', value:gender"></select>
                    </td>
                    <td><input type="text" data-bind="value: name"></td>
                    <td><input type="text" data-bind="value: age"></td>
                    <td><span data-bind="text: clientFormatted"></span></td>
                    <td><button type="button" data-bind='click:
$parent.removeClient'>Supprimer</button></td> <!-- parent car on est dans le scope
foreach -->
                </tr>
            </tbody>
            <button type="button" data-bind='click: addClient'>Ajouter un
client</button> <!-- pas parent plus de scope -->
        </table>
    </div>

<script src="/script/viewModel.js"></script>

</body>
</html>
```

Le début du tableau est identique à AngularJS, on a ensuite la balise **tbody** avec un attribut **data-bind** (on aurait pu mettre une simple **div** a la place de **tbody**). L'attribut **data-bind** n'est pas natif en html mais ne provoque pas d'erreur de validation et n'est pas interprété par le navigateur. Knockout va interpréter ces balise au moment de l'association entre la « **vue** » et le « **vue model** » (ko.applyBindings())
Le premier **data-bind** que l'on retrouve (`data-bind="foreach: clients")` est simplement une boucle qui va parcourir notre collection de clients.

Viens ensuite la liste déroulante de **Gender** (`data-bind="options:Genders, optionsText:'key', optionscaption:'Choose gender', value:gender"`). **Options** représente la collection depuis laquelle on va afficher les données, **optionsText** est le champ que l'on souhaite afficher dans la liste, **optionscaption** est la valeur par défaut à afficher et enfin **value** est l'attribut value que va prendre les elements de notre liste déroulante.

On affiche ensuite nos différents **inputs** permettant à l'utilisateur de modifier les informations d'un client. Pour cela on ajoute l'attribut : `data-bind="value: name"` afin de lier notre vue au champ « name » de notre « vue model ». Dès que la valeur est modifiée le « vue model » est automatiquement mise à jour avec la nouvelle valeurs et affiché dans notre sortie formatté.

Notre sortie formatté est simplement affiché de la même façon qu'avec les inputs sauf que l'on spécifie que la valeur n'est pas modifiable : `data-bind="text: clientFormatted"`

L'ajout et la suppression des données

Vous aurez remarqué l'ajout des boutons ajouter et supprimer qui appelle respectivement les fonctions addClient et removeClient grâce au tag `data-bind='click: addClient'.` Nous allons maintenant ajouter ces deux méthodes à notre « vue model »

Dans notre classe **ViewModel** on ajoute donc :

```
self.addClient = function () {
    self.clients.push(new Client(null , Genders[0], "Inconnu", 0 ));
};

self.removeClient = function (client) {
    index = self.clients.indexOf(client);
    self.clients.splice(index, 1);
};
```

Un peu de la même façon qu'avec AngularJS on ajoute et on enlève de notre collection un Client et les modifications seront automatiquement reporté dans notre vue.

Voilà notre application Knockout fonctionne. Comme on a pu le voir il y a beaucoup de ressemblances entre AngularJS et Knockout, les principales différences sont qu'AngularJS définit des balises particulières pour chaque actions (ng-*) alors que Knockout utilise des balise data-binding à l'intérieur desquels il définira les actions à effectuer. La seconde grosse différence est qu'avec Knockout il faut spécifier explicitement quels champs devront être observés afin de notifier la vue alors qu'avec AngularJS tous est implicite.

Ember

Ember utilise le pattern MVC comme AngularJS.

Premièrement nous allons avoir besoin Ember-data qui est une librairie permettant gérer les données et de faciliter les interactions avec le serveur, nous allons en avoir besoin pour créer nos modèles. Tous les appels de méthodes commençant par DS.* utiliserons cette librairie.

Templating simple

Nous commençons par créer une application Ember de la façon suivante :

```
window.App = Ember.Application.create();
```

Cette ligne rendra la variable App disponible dans toute notre application. C'est cette App qui contiendra toute la logique d'Ember.

Nous créons maintenant un **Adapter** qui permet de communiquer avec la source de donnée. Dans notre cas comme nous n'utilisons pas de base de données **Ember-data** nous fournis un système de **Fixture** qui nous permettra de charger les données depuis notre tableau Javascript :

```
App.ApplicationAdapter = DS.FixtureAdapter.extend();
```

Les donnée sont les même que précédemment à la différence que le gender d'un client est maintenant donné par son id et non plus directement par l'objet :

```
var Genders = [
    {
        id: 0,
        key: "Mr",
        value: "Monsieur"
    },
    {
        id: 1,
        key: "Mme",
        value: "Madame"
    }
];


var ClientsList = [
    {
        id: 0,
        gender: 0,  //Note: we give the id !
        name: "Jean",
        age: 25,
    },
    {
        id: 1,
        gender: 1,
        name: "Juliette",
        age: 30,
    }
];
```

Nous pouvons maintenant déclarer le modèle qui accueillera nos données :

```
App.Gender = DS.Model.extend({
    key: DS.attr('string'),
    value: DS.attr('string'),
```

```
})

App.Client = DS.Model.extend({
    gender: DS.belongsTo('gender', { async: false }), //relation one to
    name: DS.attr('string'),
    age: DS.attr('boolean'),

    clientFormatted: function () {
        if (this.get('gender'))
            var gender = this.get('gender').get('value');
        else
            var gender = "Inconnu"

        return gender + ': ' + this.get('name') + ' (' + this.get('age') + ' ans)';
    }.property('gender', 'name', 'age'),
});
```

Nous remarquons qu'à la différence des autres Frameworks Ember contrôle le type de chaque champs de la même façon qu'une base de donnée. On remarque aussi la relation avec gender.

Le champs **clientFormatted** est calculé à partir des autre champs comme dans les exemples précédents, on remarque qu'il faut vérifier que **gender** est bien définit pour éviter une erreur.

Il ne reste plus qu'à charger les données depuis notre collection de Clients et de Genders:

```
App.Gender.FIXTURES = Genders;
App.Client.FIXTURES = ClientsList;
```

Pour rendre nos deux modèles accessibles dans l'application nous définissons la route principale:

```
App.ApplicationRoute = Ember.Route.extend({
    model: function () {
        return Ember.RSVP.hash({
            clients: this.store.find('client'),
            genders: this.store.find('gender')
        })
    },
});
```

Ensuite on map les routes aux URL

```
App.Router.map(function () {
    this.resource('clients', { path: '/' });
});
```

Cette route permettent de dire à Ember de détecter lorsque l'URL de la page est « / » et d'afficher le Template **clients,** Que nous allons voir maintenant.

Un template est définit avec la syntaxe d'handlebars de la façon suivante :

```
<script type="text/x-handlebars" data-template-name="clients">
        <!--
            Corps du template
         -->
</script>
```

Affichage des données

A l'interieur de ce bloque de script tous ce qui est entre deux accolades ouvrante et deux fermantes (**{{…}}**) sera interprété par Ember. Voici donc le code de la page

```
<script type="text/x-handlebars" data-template-name="clients">
   <div class="container">
```

```
        <table class="table">
            <caption>Liste de clients</caption>
            <tr>
                <th>Gender</th>
                <th>Name</th>
                <th>Age</th>
                <th>Output</th>
                <th></th>
            </tr>
            {{#each client in model.clients}}
            <tr>
                <td>
                    {{view "select" content=model.genders optionValuePath="content.id"
optionLabelPath="content.key" prompt="Choose gender"
selectionBinding="client.gender"}}
                </td>
                <td>{{input type="text" value=client.name}}</td>
                <td>{{input type="text" value=client.age}}</td>
                <td>{{client.clientFormatted}}</td>
                <td><button type="button" {{action 'deleteClient'
client}}>Supprimer</button></td>
            </tr>
            {{/each}}
        </table>
        <button type="button" {{action 'addClient' }}>Ajouter un client</button>
    </div>
</script>
```

Beaucoup de ressemblance avec les précédents exemples. Le **foreach** deviens `{{#each client in model.clients}}`.

La liste déroulante `{{view "select" …}}` prend en paramètres **content** qui est le modèle qui va être listé, **optionValuePath** est la valeur que va prendre l'attribut value de chaque **option, optionLabelPath** est la valeur que va prendre chaque option, **prompt** est la valeur par défaut et enfin **selectionBinding** est la valeur qui va être sélectionné (dans notre cas le **gender** de notre **client** courant)

Les champs de saisie et d'affichage du texte formaté ressemblent aux précédents exemples.

Ajout suppression de données

On voit ensuite les deux boutons permettant d'ajouter et de supprimer des clients. Ces boutons appellent la fonction désignée par la balise **action**, on peut donner un paramètre comme avec l'exemple du bouton supprimer qui prend en paramètre le client que l'on souhaite supprimer.

Nous allons maintenant écrire ces fonctions d'ajout et de suppression de clients, pour cela on ajoute un **Controller** à notre code Javascript :

```
App.ClientsController = Ember.ObjectController.extend({
    actions: {
        addClient: function () {
            var todo = this.store.createRecord('client', {
                gender: 0,
                name: "Inconnu",
                age: 0
            });
        },

        deleteClient: function (client) {
            client.deleteRecord();
            client.save();
        }
    }
});
```

Dans ce Controller est définis les deux fonctions. Pour ajouter un Client on appelle store.createRecord() qui prend en paramètre le nom du modèle et les valeurs par défault. Pour supprimer un clients on appelle simplement **deleteRecord()** sur le client donné en paramètre.

Voilà pour Ember, passons maintenant à Backbone.

Backbone

Finissons par Backbone, ce framework utilise le pattern MVP qui est légèrement différent du pattern MVC présenté précédemment.

Modèle : comme pour les autre pattern il décrit la logique métier, les données et les fonctions de manipulation de donnés.

Vue : affiche les données du modèle. Dans ce pattern la vue n'a aucune connaissance du modèle, tout passe par le présenter.

Présenter : gère les évènements de l'interface graphique. A la différence d'un Controller le présenter est totalement découplé de la vue et communique ensemble à travers une interface. Il n'y a pas de liaison entre la vue et le modèle, tout passe par le Presenter.

Templating simple

Nous allons voir que le fonctionnement de ce Frameworks est très différent des deux autres. Toute la logique se trouve dans le code JavaScript (le presenter), c'est pourquoi on se retrouve avec un code HTML très minimaliste :

```
<div id="backBoneView"> <!-- View -->
</div>
```

Voila le seul code HTML dont nous avons besoin ! Regardons maintenant le code Javascript. On commence toujours avec le même jeux de données :

```
var Genders = [
    {
        id: 0,
        key: "Mr",
        value: "Monsieur"
    },
    {
        id: 1,
        key: "Mme",
        value: "Madame"
    }
];


var ClientsList = [
    {
        id: 0,
        gender: Genders[0],
        name: "Jean",
        age: 25,
    },
    {
        id: 1,
        gender: Genders[1],
        name: "Juliette",
        age: 30,
    }
];
```

Créons donc notre modèle Client:

```
var Client = Backbone.Model.extend({
    defaults: {
        id: -1,
        gender: Genders[0],
        name: "Inconnu",
        age: 0,
    },

    clientFormatted: function () {
```

```
        return this.get('gender')['value'] + ": " + this.get('name') + " (" +
this.get('age') + " ans)";
    }
});
```

Notre modèle prend donc une valeur par défaut qui sera utilisé lors de la création d'un nouveau Client et une fonction d'affichage formaté.

On ne va pas créer de modèle pour le Gender afin d'éviter d'alourdir le code pour un modèle qui n'est de toute façon pas modifiable.

Lions maintenant notre modèle à une Collection. Une collection est simplement une liste de modèles :

```
var List = Backbone.Collection.extend({
    model: Client
});
```

Créons ensuite la vue principale qui affichera la liste de nos clients et gérera l'évènement d'ajout d'un nouveau client:

```
var ListView = Backbone.View.extend({
```

Toutes les vue de Backbone contiennent une propriété **el** qui référence le DOM. Si cette propriété n'est pas définie Backbone va en construire une automatiquement avec un element **div** vide. Dans notre cas nous référençons l'élément HTML définit précédement :

```
    el: $('#backBoneView'),
```

Cette vue prend une fonction d'initialisation qui est automatiquement appelée au moment de l'instanciation de notre vue. Cette fonction instancie notre collection de client.

```
    initialize: function () {
        _.bindAll(this, 'render', 'addClient', 'appendClient');

        this.collection = new List(ClientsList);
        this.collection.bind('add', this.appendClient);

        this.counter = -1;
        this.render();
    },
```

bindAll permet aux fonctions donné en paramètre d'avoir accès au contexte « **this** ».

Les lignes suivantes permettent de créer la collection **List** avec le jeu de donnée **ClientList** et de lier l'évènement **add** avec la fonction **appendClient**.

La variable **counter** nous permettra de changer l'ID d'un client au moment de l'ajout.

Affichage des données

Enfin nous appelons la méthode **render** qui permet d'afficher notre liste. Cette fonction va générer le code HTML de notre tableau :

```
    render: function () {
         var self = this;

        $(this.el).append("<table class='table'><caption>Liste de
clients</caption><tr><th>Gender</th><th>Name</th><th>Age</th><th>Output</th><th></th><
/tr></table>");

        _(this.collection.models).each(function (item) {
            self.appendClient(item);
        }, this);
```

```
        $(this.el).append("<button type='button' id='add'>Ajouter un
client</button>");
    },
```

Backbone préconise de garder au maximum la logique dans le code Javascript, c'est pourquoi on ne crée pas le tableau directement dans le code HTML de notre page mais on préfère le générer dans la fonction de rendu.

On parcourt notre collection et pour chaque client on appelle la fonction **appendClient** suivante :

```
    appendClient: function (item) {
        var client = new ClientView({
            model: item
        });

        $('table', this.el).append(client.render().el); //Automatically add <tr></tr>
    }
```

Ajout suppression de données

Cette fonction crée un vue client dont nous allons voir la syntaxe plus tard et l'ajoute à notre tableau. Enfin on ajoute un évènement **add** de notre bouton qui appellera la fonction **addClient**:

```
    events: {
        'click button#add': 'addClient',
    },

    addClient: function () {
        var client = new Client();
        client.set({
            id: this.counter
        });
        this.counter--;
        this.collection.add(client);
    },

});
```

La fonction **addClient** crée un nouveaux **Client**, change son **id** et finalement l'ajoute à notre collection.

Voilà pour notre vue principale qui s'occupe d'afficher la liste de tous les clients. Passons maintenant à la vue d'un seul client. Cette vue est appelée dans la fonction **appendClient** vue précédemment et va nous permettre d'afficher chaque clients individuellement.

```
var ClientView = Backbone.View.extend({
```

On ajoute un attribut **tagName** qui est le tag de l'élément qui sera crée pour chaque client.

```
    tagName: 'tr',
```

La méthode d'initialisation fonctionne de la même façon que la vue générale à la différence qu'ici on ne travail plus sur une collection mais sur un modèle:

```
    initialize: function () {
        _.bindAll(this, 'render', 'updateName', 'updateAge', 'unrender', 'remove');
        this.model.bind('remove', this.unrender);
    },
```

Passons maintenant à la fonction d'affichage de notre vue Client:

```
    render: function () {
        var self = this;
```

On commence par la fonction qui construit la liste de « Genders ». On n'aurait pu passer par un autre modèle avec sa vue mais pour garder le code concis nous allons seulement parcourir notre jeu de donnée et afficher ses valeurs.

```
        var select = "<select id='gender'>";
        Genders.forEach(function (entry) {

            var selected = '';
            if (entry.id == self.model.get('gender')['id'])
                selected = 'selected';

            select += "<option " + selected + " value='" + entry.value + "'>" +
       entry.key + "</option>";

       });
       select += "</select>";
```

Ensuite on affiche le modèle du Client courant :

```
        $(this.el).html("<td>" + select + "</td><td><input type='text' id='name'
value='" + this.model.get('name') + "'></td><td><input type='text' id='age' value='" +
this.model.get('age') + "'></td><td>" + this.model.toString() + "</td><td><button
type='button' class='delete'>Supprimer</button></td>");
        return this;
    },
```

Maintenant que notre affichage est terminé on ajoute les évènements correspondant au changement de sélection dans notre liste déroulante, au changement dans les textes boxes du formulaire ainsi qu'au clique sur le bouton supprimer.

```
    events: {
        'change select#gender': 'genderChange',
        'change  input#name': 'updateName',
        'change input#age': 'updateAge',
        'click button.delete': 'remove'
    },
    genderChange: function(e){
        var index = $(e.currentTarget)[0].selectedIndex;
        this.model.set({ 'gender': Genders[index] });
        this.render();
    },
    updateName: function (e) {
        var val = $(e.currentTarget).val();
        this.model.set({ 'name': val });
        this.render();
    },
    updateAge: function (e) {
        var val = $(e.currentTarget).val();
        this.model.set({ 'age': val });
        this.render();
    },
    remove: function () {
        this.model.destroy();
    },
    unrender: function () {
        $(this.el).remove();
    }
});
```

Lors d'un évènement on récupère la nouvelle valeur, on change la valeur correspondante dans le modèle, puis on appelle la fonction de rendu.

La fonction **unrender** est automatiquement appelée lors de la suppression du modèle. On l'a lier dans la méthode **initialize** précédente.

Finalement il ne reste plus qu'à créer notre vue principale afin que le rendu soit fait :

```
var listView = new ListView();
```

Voila !

Particularités

Les routes AngularJS & Backbone & amber (Knockout plugin externe).

Backbone everything is JS mais plus de code?

Les chiffres

	AngularJS	Knockout	Ember	Backbone
Lignes de code	170 000	10 000	50 000	10 000
Poids avec dépendances (gziped)	56.9KB	19.8KB	129 + 37.1* + 28.7** + 28.6*** = 223.4KB	6.8 + 37.1* + 5.5**** = 49.4KB
Rapidité exécution	60ms	30ms	220ms	45ms
Contributeurs sur Github	1078	45	452	242
Projets annexes	400	100	500	200
Questions sur stackoverflow	66 444	12 374	12 846	16 863
Pattern de conception conseillé	MVC	MVVM	MVC	MVP

*jquery, **ember-data, ***handlebar, ****underscore

Le temps de chargement d'une page web est crucial pour sa réussite. Les utilisateurs ne montrent pas beaucoup de patience quant au temps de chargement d'une page web, c'est pourquoi il essentiel de prendre en compte le temps de chargement et d'initialisation d'une librairie. Malgré le nombre de ligne de code plus important que ses concurrent **AngularJS** est plus légère car elle n'a besoin d'aucunes dépendances. **Backbone** nécessite l'utilisation de **underscore** et de **JQuery** ce qui lui fait prendre du poids malgré son nombre de lignes de codes moins important. **Ember** a besoin de **JQuery** et de **Handlebar** pour fonctionner, d'où son poids très important par rapport à ses concurrents. Enfin **Knockout** ne nécessite aucune autre librairie ce qui en fait la plus légère de ce comparatif. La taille de votre site impactera aussi, si un Framework est lourd mais son utilisation produit peu de ligne de code il pourra tout de même être avantageux.

Développer son propre Framework ?

Utiliser un Framework connu à l'avantage d'offrir au développeur de la rapidité sur un Framework qu'il connait déjà et de faciliter le transfert de la maintenance du code source à un client ou à une équipe tierce. Mais alors pourquoi développer son propre Framework et réinventer la route ? Tout d'abord cela permet d'avoir plus de souplesse, on peut le modifier, l'adapter et le faire évoluer selon les besoins tout en restant plus léger qu'un Framework existant dont on n'utilisera probablement pas toutes les fonctions. On décide de l'architecture à adopter et on n'est pas restreint à celle imposé par le Framework que l'on utilise.

Malgré ces avantages réinventer la roue n'est pas toujours la bonne solution, même si les composants développé seront réutilisables au fils des projets un nouveau venu aura du mal en en comprendre les subtilités et mettra donc du temps à s'adapter, il ne pourra pas forcement s'aider d'internet ou de documentations pour avancer. De plus Il faudra gérer vous-même la maintenance, les évolutions et les adaptations aux nouveautés des langages sur lesquels il repose. La compatibilité entre les navigateur par exemple, est un problème qui est devra être pris en compte dès le début du développement

Cette solution est à envisager dans de rare cas seulement ou des exigences bien précise sont requise car elle demande beaucoup de travail supplémentaire alors que de nombreux Frameworks existent déjà.

Pas de vainqueurs…

Même si **AngularJS** semble légèrement devant ses concurrents il n'en est pas le vainqueur à l'unanimité. En effet chacun se démarque par son approche différente du sujet. **Knockout** est un bon concurrent grâce à sa légèreté et la concision du code produit. **Amber** avec son système de routage peut être très utile. **Backbone** est très orienté JavaScript et donc adapté à des développeurs connaissant bien ce langage.

Il n'y a donc pas de réponse catégorique quant au choix d'un Framework ou d'utiliser son propre Framework, il dépendra avant tout du besoin, du temps et de l'équipe qui participera au projet. Cet article traite seulement la surface des 4 Frameworks les plus connu mais il en existe des dizaine offrant des fonctionnalités différentes.

Aller plus loin

Si vous décidez d'utiliser un Framework voici quelques liens utiles :

Sites officiels :

- AngularJS : https://angularjs.org/
- Knockout : http://knockoutjs.com/
- Ember : http://emberjs.com/
- Backbone : http://backbonejs.org/

Explorer d'autres Frameworks :

Comparaison Frameworks :
http://en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks,
http://codebrief.com/2012/01/the-top-10-javascript-mvc-frameworks-reviewed/ et
http://www.infoq.com/research/top-javascript-mvc-frameworks

Développer son propre Framework :
http://blogs.infinitesquare.com/b/jonathan/archives/mon-framework-mvvm-a-moi

Why backbone: http://backbonetutorials.com/why-would-you-use-backbone/