PROGRAMMEZ.COM

PROGRAMMEZ!

Mensuel n°183 - Mars 2015

le magazine du développeur

Devenez un super héros du JavaScript

Android

- *Les (meilleurs) outils pour développer vos apps*
- *La nouvelle interface Material Design*

Choisir son framework JavaScript

JavaScript pour les Jedis

Choisir son école d'informatique

Késako le Machine Learning ?

Faites carrière dans les jeux vidéo !

09-24-13 © VosjaKorman

Geek

Des étudiants révolutionnent la cabine d'essayage

Accessibilité des applications

Il y a urgence !

SDK Azure : les nouveautés

M 04319 - 183 - F: 5,95 € - RD

Printed in EU - Imprimé en UE - BELGIQUE 6,45 € SUISSE 12 FS - LUXEMBOURG 6,45 € - DOM Surf 6,90 € Canada 8,95 $ CAN - TOM 940 XPF - MAROC 50 DH

My JavaScript is beautiful !

Vous allez aimer JavaScript ! Et vous n'allez pas avoir le choix. JavaScript (JS) a longtemps été le mal aimé des langages Web et à juste raison : complexe, lourd, pas toujours performant. Depuis quelques années, de gros efforts ont été réalisés sur les outils et les nombreux frameworks « simplifient » le développement JS.
Fondamentalement, JS reste un langage hardcore et pas toujours facile, surtout, pour les développeurs débutants. Mais la force de JS vient de sa communauté et des nombreux outils. D'autre part, l'arrivée de serveurs JS (Node.JS, Wakanda) a bouleversé la situation, et la possibilité de faire du Node.JS sur des objets connectés va sans doute ouvrir d'autres perspectives... Nous vous proposons un mini-dossier autour de JS : comment choisir un framework JS, JS en mode Jedi, et, enfin, nous terminerons notre voyage dans le merveilleux monde du fonctionnel...

```
{template .Redaction}
  La redaction!
{/template}
```

Bien choisir son Framework JavaScript : AngularJS, Knockout, Ember, Backbone.js

Choisir un Framework JavaScript n'est jamais 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 connus.

Feltz Ludovic, Consultant **SoftFluent**
ludovic.feltz@softfluent.com

Plusieurs critères sont à prendre en compte dans le choix d'un framework :

- Le poids définira le temps de chargement des pages. Critère important pour les applications mobiles,
- La clarté de la documentation permet de réduire le temps d'apprentissage (lors de l'arrivée d'une nouvelle personne dans l'équipe par exemple),
- La facilité d'utilisation simplifie et accélère le développement,
- Une communauté active garantit l'évolution du Framework et une meilleure aide en cas de problèmes.

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. Nous n'aborderons pas la communication avec le serveur et la sauvegarde dans une base de données.

POURQUOI UTILISER UN FRAMEWORK JAVASCRIPT ?

Les Frameworks JavaScript offrent des fonctionnalités prédéfinies permettant aux développeurs de ne pas avoir à manipuler le DOM (Document Object Model) directement depuis leur code. Ils offrent donc un niveau d'abstraction et facilitent le développement d'applications Web. La plupart des Frameworks étudiés offrent la possibilité de développer des applications sur une seule page (Single Page Application) ce qui permet d'offrir une expérience plus fluide à l'utilisateur en évitant de recharger la page à chaque action. Le fonctionnement se rapproche alors d'une application de bureau. Les ressources sont chargées dynamiquement et ajoutées à la page quand c'est nécessaire, généralement en réponse à une action de l'utilisateur Fig.1. L'utilisation d'un Framework offre aussi un aspect structurant à l'application en proposant l'utilisation d'un pattern MVC, MVVM ou MVP. Un développeur arrivant sur le projet n'aura donc pas à se demander pourquoi et comment vous avez conçu telle ou telle partie de l'application.

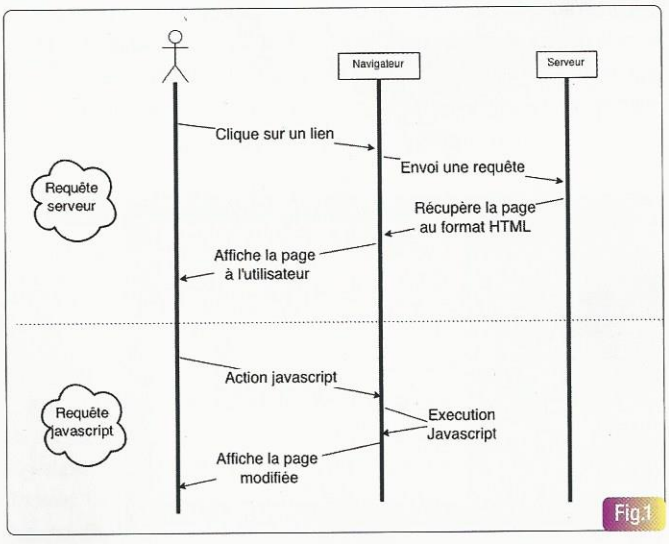

Fig.1

UN PEU D'HISTOIRE POUR COMMENCER

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 17 000 lignes de code. En 3 semaines il est parvenu à réduire ce nombre à seulement 1 000 lignes, ce qui a convaincu Google de sponsoriser ce projet et ainsi créer sa renommée. Son but est de simplifier le développement et les tests de sites Web. Il utilise 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. Il simplifie l'utilisation du JavaScript et utilise 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 SPA (Single Page Application) grâce à son système de route.

Backbone : Créé en 2010 par Jeremy Ashkenas qui a aussi participé 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.

MISE EN PRATIQUE

Dans le but de comparer les différents Frameworks, nous allons créer une application simple en utilisant chacun des Frameworks présentés précédemment Fig.2. Cette application permet de faire du CRUD sur une liste de clients. La colonne Output affiche la valeur du modèle côté client (JavaScript). Pour peupler nos modèles nous allons à chaque fois réutiliser le même jeu de données. Comme nous n'allons pas aborder la connectivité avec une base de données nous allons utiliser de simples tableaux JavaScript :

```
var Genders = [
  {
    id: 1
    key: "Mr",
    value: "Monsieur"
  },
  {
    id: 2,
    key: "Mme",
    value: "Madame"
  }
]


var ClientsList = [
  {
    id: 1,
    gender: Genders[0],
    name: "Jean",
    age: 25,
  },
  {
    id: 2,
    gender: Genders[1],
    name: "Juliette",
    age: 30,
  }
]
```

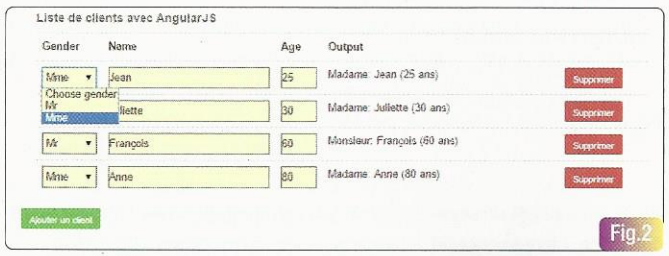

Fig.2

AngularJS

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

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

View : la 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, tels qu'un clic sur un bouton, au Controller.

Controller : il reçoit les informations de l'utilisateur provenant de la vue, les traite et les envoie au modèle. Il renvoie ensuite les informations à la vue pour les afficher. Il agit comme le coordinateur entre le modèle et la vue.

Initialisation

Commençons par créer une application AngularJS très simple. Voici le contenu de la page HTML (index.html) :

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

  <!-- Angular 1.3.0 (aucune dépendance) -->
  <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 le code JavaScript (script/controllers.js) :

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

AngularJS définit ses propres attributs HTML, commençant par « ng- ». Angular permet également de définir des balises comme nous le verrons par la suite. Le premier attribut se trouve directement dans la balise HTML ouvrante (ng-app), il permet de définir la racine de notre application

donnant ainsi la possibilité au développeur d'utiliser toute la page ou seulement une portion de celle-ci.
Le deuxième attribut que l'on voit apparaître est ng-controller. Il fournit le contexte du **controller** et permet de lier notre vue avec notre modèle.

Affichage des données

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

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

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

>> voir code complet (lien à la fin de d'article).

On voit apparaître ici 4 nouvelles balises appartenant à AngularJS. ng-repeat s'apparente à un « **foreach** ». Pour chaque client de notre controller, AngularJS crée une balise <tr> dont le titre sera l'identifiant de ce client.
Les accolades : {{expression}} permettent à AngularJS de savoir quels blocs de codes interpréter, et 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, les données du modèle sont automatiquement mises à jour !
ng-options permet de peupler notre liste déroulante et de définir ce qui sera affiché. Ici on affichera la clé de chaque **gender**
Enfin ng-click permet de spécifier le nom de la fonction qui sera appelée lors d'un clic sur le bouton.

Mise à jour des données

Il ne nous reste maintenant plus qu'à ajouter nos fonctions removeClient() et addClient(client) 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(client) 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. Ce pattern permet de garder une organisation simple d'une application graphique en séparant le code en trois parties:

- **Model:** il décrit les données manipulées par l'application. Il offre des méthodes permettant de récupérer et de modifier ces données.
- **View Model:** est une abstraction de la vue qui sert de médiateur entre le modèle et la vue. Il fournit les fonctions permettant d'ajouter ou supprimer des éléments, etc...
- **View:** affiche les informations du **View Model**, envoie des commandes telles qu'un clic sur un bouton, et est automatiquement mis à jour dès qu'il y a un changement sur le **View Model.**

Initialisation

Avec Knockout il faut lier manuellement le **Model** au **View Model**. Commençons donc par modifier notre modèle, **Genders** n'étant pas modifiable, sa structure ne change pas :

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

Nous modifions maintenant 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 **Views** des changements, pour cela on utilise ko.observable(value) 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 () {...}) qui permet à knockout de savoir que cette propriété dépend d'un ou plusieurs **observables,** et donc qu'il doit la mettre à jour lorsqu'une des dépendances est modifiée. 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),
  ]);
}
```

Notre tableau est un **observableArray** qui permet de suivre l'état de notre collection, ce qui sera utile lorsque nous implémenterons l'ajout et la

suppression de nouveaux clients dans notre collection. Enfin il faut activer Knockout afin d'associer notre **View** à notre **View Model.** Il suffit pour cela 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 **View Model** est correctement construit passons à la vue. On commence par inclure knockout dans le header de notre document :

>> voir code complet (lien à la fin de d'article).

Le début du tableau est identique à AngularJS, on a ensuite la balise tbody avec un attribut data-bind. 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 balises au moment de l'association entre la **view** et le **view model** (ko.applyBindings(view))
Le premier data-bind que l'on retrouve (data-bind="foreach: clients") est simplement une boucle qui va parcourir notre collection de clients. Vient 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 vont prendre les éléments de notre liste déroulante. On affiche ensuite nos différents input 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 **view model.** Dès que la valeur est modifiée le **view model** est automatiquement mis à jour avec les nouvelles valeurs et affiché dans notre sortie formatée. Notre sortie formatée est simplement affichée de la même façon qu'avec les input sauf que l'on spécifie que la valeur n'est pas modifiable : data-bind="text: clientFormatted"

Mise à jour 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 **view 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);
};
```

De la même façon qu'avec AngularJS, lorsque l'on ajoute ou on enlève un **Client** de notre collection, les modifications seront automatiquement reportées dans la vue.
Désormais l'application Knockout fonctionne. Comme on a pu le voir, il y a beaucoup de ressemblances entre AngularJS et Knockout. Bien qu'il y ait de nombreuses similitudes entre AngularJS et Knockout, on note toutefois plusieurs différences dont les principales sont:

- AngularJS définit des balises particulières pour chaque action (ng-*) au sein desquelles on peut mettre du code JavaScript. Tandis que Knockout utilise des balises data-bind à l'intérieur desquelles il définit des mots clefs indiquant l'action à effectuer.
- La seconde grosse différence est qu'avec Knockout il faut spécifier explicitement quels champs du modèle devront être observés afin de notifier la vue. Alors qu'avec AngularJS tout est implicite.

Ember

Ember utilise le pattern MVC comme AngularJS.
Premièrement nous allons avoir besoin d'Ember-data qui est une bibliothèque permettant de 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.** » utiliseront cette bibliothèque.

Initialisation

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. Créons maintenant un **Adapter** qui permet de communiquer avec la source de données. Dans notre cas comme nous n'utilisons pas de base de données, **Ember-data** nous fournit un système de **Fixture** qui permet de charger les données depuis notre tableau JavaScript :

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

Les données sont les mêmes 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: on donne l'ID et non l'objet !
    name: "Jean",
    age: 25,
  },
  {
    id: 1,
    gender: 1,
    name: "Juliette",
    age: 30,
  }
];
```

Déclarons 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 1 à 1
  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'),
});
```

A la différence des autres Frameworks, Ember contrôle le type de chaque champ de la même façon qu'une base de données. On remarque aussi la relation avec **Gender** qui est définie explicitement.
Le champ **clientFormatted** est calculé à partir des autres champs, comme dans les exemples précédents. Il faut aussi vérifier que le champ gender est bien défini pour éviter une erreur.
Il ne reste plus qu'à charger les données depuis nos collections **ClientsList** et **Genders**; on fait cela grâce aux **Fixtures** :

```
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 associe cette route à une URL :

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

Cette route permet de demander à Ember de détecter lorsque l'URL de la page est « / » et d'afficher le Template **clients**, que nous verrons. Un Template est défini avec la syntaxe de scripts **handlebars** de la façon suivante :

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

Affichage des données

A l'intérieur de ce bloc de script, tout ce qui est entre deux accolades ouvrantes 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" optionLabel
Path="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 ressemblances avec les précédents exemples. Le « **foreach** » devient: **{{#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 sélectionnée (dans notre cas le **gender** de notre **client** courant).
Les champs de saisie et d'affichage du texte formaté ressemblent aux précédents exemples.

Mise à jour des 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 'actionName' }}.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 ajoutons 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 sont définies les deux fonctions. Pour ajouter un **Client** on appelle store.createRecord() qui prend en paramètre le nom du modèle et les valeurs par défaut. Pour supprimer un client 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 autres patterns, il décrit la logique métier, les données et les fonctions de manipulation de données.
View : affiche les données du modèle. Dans ce pattern, la vue n'a aucune connaissance du modèle, tout passe par le **Presenter**.
Presenter : gère les évènements de l'interface graphique. A la différence d'un **Controller,** le **Presenter** est totalement découplé de la vue et communique avec elle à travers une interface. Il n'y a pas de liaison entre la vue et le modèle, tout passe par le **Presenter**.

Initialisation

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

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

Voilà le seul code HTML dont nous avons besoin ! Regardons maintenant le code JavaScript. On commence toujours avec 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,
  }
];
```

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)";
  }
});
```

Le modèle prend donc une valeur par défaut qui sera utilisée lors de la création d'un nouveau **Client** et une fonction d'affichage formatée.
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 vues 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 élément **div** vide. Dans notre cas nous référençons l'élément HTML défini précédemment :

```
  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 clients.

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

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

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

_.bindAll(this, 'functionName', ...) permet aux fonctions données 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 (dans le **Presenter**), 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>
  }
```

Mise à jour des données

Cette fonction crée une vue client dont nous allons voir la syntaxe plus tard, et l'ajoute à notre tableau. Enfin, on ajoute un évènement click à 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 nouveau **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 client individuellement.

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

On ajoute un attribut tagName qui est le tag de l'élément qui sera créé 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 travaille 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 aurait pu passer par un autre modèle avec sa vue, mais pour garder le code concis nous allons seulement parcourir notre jeu de données 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 correspondants au changement de sélection dans notre liste déroulante, au changement dans les textes boxes du formulaire, ainsi qu'au clic 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 liée 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! Nous avons réalisé le même exemple d'application avec les 4 Frameworks. On a aussi remarqué un certain nombre de ressemblances et de différences.

En résumé

	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
Contributeurs sur GitHub	1078	48	452	242
Projets annexes	400	100	500	200
Questions sur StackOverflow	66 444	12 374	12 846	16 863
Patterns de conception supportés	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 est essentiel de prendre en compte le temps de chargement et d'initialisation d'une librairie. Ce temps aura aussi un impact si un Framework est lourd, car si son utilisation nécessite peu de lignes de code, il pourra tout de même être avantageux.

Pas de vainqueurs

AngularJS se démarque de ses concurrents grâce à sa simplicité d'utilisation, sa documentation très bien faite et sa communauté active sur StackOverflow. De plus sa méthode pour enrichir le code HTML avec ses balises personnalisées le rendent très lisible. Ce sont les raisons qui font qu'AngularJS est rapide à prendre en main. De plus il est développé et maintenu par Google ce qui est gage d'une qualité certaine. On pourrait lui reprocher un manque de modularité. Malgré tout, il convient parfaitement pour la majorité des projets.

Knockout ressemble beaucoup à **AngularJS**, sa documentation et notamment son système de tutorial interactif le rendent très simple à prendre en main. Malheureusement sa syntaxe qui consiste à tout mettre dans un seul attribut HTML (data-bind) fait perdre la coloration syntaxique. Le code de la page Web devient donc difficilement lisible sur de gros projets.
Ember se détache de ses concurrents par son système de route plus avancé qui l'oriente vers des sites SPA (Single Page Application). Son système qui lui permet de spécifier le type des attributs et son système de store simplifie les connexions à des bases de données.
Malheureusement son apprentissage est un peu plus compliqué et sera réservé à des projets de plus grosse envergure.
Backbone Tout le code HTML étant généré par le code JavaScript, il et donc plutôt adapté à des développeurs maitrisant bien ce langage. Toute la construction du code HTML et la gestion des événements doivent être gérées par le développeur, ce qui alourdit fortement le code final. Sa documentation ne propose aucun tutoriel pas à pas comme le font ses concurrents. Ainsi l'apprentissage de ce Framework peut se révéler compliqué pour un développeur débutant. Malgré tout, il offre une plus grande modularité par rapport à ses concurrents.
Ainsi, il n'y a donc pas de réponse catégorique quant au choix d'un Framework JavaScript. Il dépendra avant tout du besoin, du temps et de l'équipe qui participera au projet. Cet article traite seulement de 4 des Frameworks les plus connus, mais il en existe des dizaines offrant des fonctionnalités différentes.

Développer son propre Framework ?

Utiliser un Framework connu permet aux développeurs d'être plus rapidement opérationnels sur un projet, 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 roue ? 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 fonctionnalités. On décide de l'architecture à adopter et on n'est pas restreint à celle imposée par le Framework que l'on utilise.
Malgré ces avantages, réinventer la roue n'est pas toujours la bonne solution; même si les composants développés seront réutilisables au fil des projets, un nouveau membre dans l'équipe de développement aura du mal à 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 avec les différents navigateurs par exemple, est un problème qui devra être pris en compte dès le début du développement. Cette solution est à envisager dans de rares cas seulement où des exigences bien précises sont requises, car elle demande beaucoup de travail supplémentaire, alors que de nombreux Frameworks existent déjà.

Sources :

Codes complets des exemples :
https://github.com/ludo6577/JavascriptFrameworks

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