BACKBONE.JS

Redonnez de la structure à vos applications

Cédric Clavier

Le 20/10/2015

- Présentation de Backbone
- Les Événements
- Le Modèle
- La Collection
- La Vue
- Le Routeur

Et si on parlait un peu de moi avant ?

- Développeur front-end chez SAOOTI
- J’ai fait mes premières armes en Flex
- Depuis quelques années utilisation du HTML5
- À la recherche des meilleures solutions pour du front-end “propre”

Qu’est-ce que Backbone ?

- Une librairie javascript developpée par Jeremy Ashkenas (également créateur du CoffeScript)
- Utilisée par les plus grands acteurs du web
  http://builtwithbackbonejs.com/
- Un outil pour structurer et découpler votre code
- Un outil pour réaliser des applications web SPA (Single-Page-Application)

Les petits copains de Backbone

- Obligatoire
  - Underscore JS : Fournit un ensemble de méthodes utilitaires
  - JQuery / Zepto : Le fameux framework de manipulation du DOM
  - Handlebars.js : Framework de templating
- Que j'utilise
  - RequireJS : Permet de faire un code javascript modulaire (AMD)
  - Q.js : Permet de gérer proprement l'asynchronisme grâce au Promise (A+)

Pourquoi utiliser Backbone ?

- Parce que c'est léger : Backbone+Underscore (38,5Ko) AngularJS(145Ko)
- Parce que c'est simple, le code est lisible et les erreurs sont facilement traçables
- Parce que c'est bien documenté, et qu'en plus de la documentation il existe une version annotée des sources qui permet de vite comprendre le code
- Parce que c'est populaire (23090 étoiles sur GitHub)
- Parce que ça ne fait pas le ménage, la vaisselle, le café et la comptabilité. Ça ne fait que ce pourquoi c'est fait mais ça le fait bien !

Et maintenant on rentre dans le vif du sujet

Les événements

Les événements

- Backbone utilise le pattern Observable (utilisation d'événements) pour assurer la communication entre les modules
- Ils sont intégrés pour la plupart de façon native dans les composants de base
- Mais vous pouvez (devez) ajouter les vôtres

Les événements

Un événement est un objet transmis avec un label

Tous les objets Backbone utilisent les événements avec les méthodes suivantes :

- Pour ajouter un event listener
  .on(label, callback, [context])
- Pour supprimer un event listener
  .off(label, callback, [context])
- Pour envoyer un événement
  .trigger(label, args)*

Les événements

Exemple d'envoi et de réception d'événements

```
var Receiver = Backbone.Model.extend({

    receiver: undefined,

    setSender: function (r) {
        this.receiver = r;
        this.receiver.on('refresh', this._onRefresh, this);
    },

    dispose: function () {
        this.receiver.off('refresh', this._onRefresh, this);
        this.receiver = undefined;
    },

    _onRefresh: function () {
        console.log('Je suis rafraîchi ;-)');
        if (arguments.length) {
            console.log(arguments);
        }
    }

});

var sender   = new Backbone.Model();
var receiver = new Receiver();
receiver.setSender(sender);

sender.trigger('refresh');
sender.trigger('refresh', 'g33k', '1337');

receiver.dispose();
```

```
Je suis rafraîchi ;-)
Je suis rafraîchi ;-)
["g33k", "1337"]
```

Le modèle

Le modèle

- Représente des données
- Interagit avec le back-end afin d’assurer la persistance
- Contient également la logique métier (validation, conversion, calcul, …)

Le modèle

Quelques fonctions utiles :

- Récupérer le contenu du modèle au format JSON
  .toJSON()
- Définir une propriété
  .set('propertyName', propertyValue, [options])
  .set(jsonModel, [options])
- Récupérer une propriété
  .get('propertyName')

Les modèles

Lorsque l'on ajoute ou modifie une propriété, deux événements sont automatiquement envoyés

- Le plus précis est un événement de la forme
  "change:propertyName"
- Le plus générique indique juste un changement
  "change"

Si on affecte un ensemble de propriétés d'un coup (avec un objet JSON) l'événement "*change*" est envoyé une seule fois

Les modèles

Pour interagir avec le serveur il existe trois méthodes :

- Pour récupérer le modèle depuis le serveur
  .fetch([options])
- Pour sauvegarder le modèle sur le serveur (création ou mise à jour)
  .save([attributes], [options])
- Pour supprimer le modèle du serveur
  .destroy([options])

Il existe une propriété *idAttribute* permettant de définir quelle est l'id du modèle

Par défaut tous les transferts se font en JSON

Lors de la suppression d'un modèle un événement *'destroy'* est envoyé

Les modèles

Pour interagir avec le serveur :

- L'implémentation par défaut de Backbone envoie/reçoit du JSON sérialisé avec le content-type "application/json" et suit la spécification suivante

create	*POST*	*/media*
read	*GET*	*/media/id*
update	*PUT*	*/media/id*
delete	*DELETE*	*/media/id*

- Pour utiliser le fonctionnement par défaut il suffit alors de définir une urlRoot au modèle
- Toutes les méthodes interagissant avec le serveur sont évidement asynchrones et renvoient un objet jqXHR

Le modèle

Exemple :

- Création d'un média

```
var Media = Backbone.Model.extend({
    idAttribute    : 'mediaId',
    urlRoot        : '/media/'
});

/*-------------------------------------------------------------------------------------------
 Création d'un nouveau média sur le serveur
-------------------------------------------------------------------------------------------*/
var m = new Media({'name': 'The fear', 'artist': 'Lily Allen'});

m.save().then(function () {
//-> l'url est : 'POST /media/' avec comme contenu {"name":"The fear","artist":"Lily Allen"}
    console.log(m.get('mediaId'));
//-> 1
}, function () {console.error('Error while saving media');});
```

Le modèle

Exemple :

- Récupération d'un média depuis le serveur
- Modification du média
- Suppression du média

```
/*-------------------------------------------------------------------------------------------
 Récupération d'un média sur le serveur, puis modification puis suppression
 -------------------------------------------------------------------------------------------*/
var m = new Media({'mediaId': 1});

m.fetch().then(
//-> l'url est : 'GET /media/1'
        function () {
            console.log(JSON.stringify(m.toJSON()));
//-> {"name":"The fear","artist":"Lily Allen","mediaId":1}
            m.set('name', 'F*** You');
            return m.save();
//-> l'url est : 'PUT /media/1'
        }, function () {console.error('Error while fetching media');}
).then(
        function () {
            return m.destroy();
//-> l'url est : 'DELETE /media/1'
        }, function () {console.error('Error while saving media');}
).then(function () {
            //Model is currenlty destroyed
        },
        function () {console.error('Error while deleting media');}
);
```

Les modèles

On peut si on veut redéfinir les urls pour les interactions avec le serveur

Pour ce faire il faut redéfinir la fonction *sync* du modèle
.sync(method, model, [options])

Le premier paramètre de la méthode indique l'action ("create", "read", "update", ou "delete") et le deuxième le modèle sur lequel agir

Ainsi on peut utiliser n'importe quelle combinaison (http action,url) pour chacune des fonctions ci dessus

Je m'en sert également pour wrapper les retours des "promises" jQuery dans des promises Q

Le modèle

Exemple :

- Redéfinition de l'interaction serveur 'fetch'

```
var Media = Backbone.Model.extend({
    idAttribute: 'mediaId',

    sync: function (method, model) {
        switch (method) {
            case "read"   :
                return Q.promise(function (res, rej) {
                    $.get('/media/get/' + model.id).then(
                            function (data) {
                                model.set(data);
                                res(model);
                            },
                            function () {
                                console.error('Error while fetching media')
                                rej(new Error('MEDIA_FETCHING_ERROR'));
                            }
                    );
                });
                break;
            //[...]
      }
    }
});
```

Le modèle

Exemple :

- Redéfinition de l'interaction serveur 'create'

```
var Media = Backbone.Model.extend({
    idAttribute: 'mediaId',

    sync: function (method, model) {
        switch (method) {
            case "create" :
                return Q.promise(function (res, rej) {
                    $.ajax({
                        'url'        : '/media/create',
                        'type'       : 'post',
                        'dataType'   : 'json',
                        'processData': false,
                        'contentType': 'application/json;charset=utf-8',
                        'data'       : JSON.stringify(data)
                    }).then(function (data) {
                        model.set(data);
                        res(model);
                    }, function () {
                        console.error('Error while creating media');
                        rej(new Error('MEDIA_CREATION_ERROR'));
                    });
                });
                break;
            //[...]
        }
    }
});
```

La collection

La collection

Une collection contient un ensemble d'objets appartenant au même modèle

Pour chaque collection on définit le modèle utilisé

Comme pour un modèle on peut effectuer des interactions serveur via *'sync'*

Lors d'un changement sur un des modèles de la collection, celle-ci envoie un évènement *'change'*

La collection

Les opérations basiques sur une collection :

- Récuperation d'un élément (par l'id)
  .get(id)
- Ajout d'un élément
  .add(models, [options])
- Suppression d'un élément
  .remove(models, [options])
- Nettoyage d'une collection
  .reset([models], [options])

De plus pour les trois dernières opérations ci-dessus un événement éponyme est envoyé

La collection

Une collection peut être triée en utilisant *'comparator'*

Toutes les méthodes fournies par *underscore* peuvent également être utilisées

- map
- reduce
- filter
- find
- …

Attention le retour de ces méthodes sur une collection n'est pas une collection mais un tableau javascript

La collection

Exemple :

- Exemple d'utilisation d'une collection

```
var Medias = Backbone.Collection.extend({
    model     : Medias,
    comparator: 'name',

    getNames: function () {
        return this.pluck('name');
    },

    getIdsAndNames: function () {
        return this.map(function (m) {
            return {
                'mediaId': m.id,
                'name'   : m.get('name')
            }
        });
    }
});
```

La vue

La vue

La vue fait la glu entre le DOM et l'application

On utilise généralement des templates (handlebars, mustache,...) pour faire le rendu (opération consistant à transformer à l'aide d'un template des données javascript en HTML)

Dans la vue on va récupérer les interactions utilisateurs via les événements du DOM

Et on va y répercuter les modifications de l'application via les événements Backbone

La vue

Lors de la création de la vue on va lui fournir un sélecteur DOM qui va correspondre à sa zone d'affichage
el

On va également lui fournir les objets (modèles, collections, …) sur lesquels elle va pouvoir réagir
.initialize([options])

Puis on va faire le rendu
.render()

La vue

Il existe un raccourci pour attacher des DOM event listeners à la vue

Le formatage est le suivant
events : { ‘action .selector’ : ‘methodToCall’}

La méthode de callback sera automatiquement appelée avec le contexte de la vue (le bon)

Si le DOM appartenant à la vue change, les event listeners seront automatiquement réaffectés sur les nouveaux éléments

La vue

Exemple :

- Exemple d'utilisation d'une vue

```
var MyView = Backbone.View.extend({
    media    : undefined,
    template: myTemplate,

    initialize: function (options) {
        this.media = options.media;
        this.media.on('change', this.render, this);
    },

    render: function () {
        this.$el.html(this.template(this.media.toJSON()));
        return this;
    },

    events: {
        'click .edit': 'onEditMedia'
    },

    onEditMedia: function (event) { /*Do something*/ }
});

var view = new MyView({
        el: '.media-edit-zone',
        media: new Media({'mediaId': 1}).fetch()
}).render();
```

Le routeur

Le routeur

Le routeur sert à utiliser des urls partielles (hash) pour assurer une navigation dans l'application

Il n'existe à priori qu'un seul routeur par application

En utilisant des liens de type *hash (eg #/create/media)* on peut faire des affichages différents sans recharger la page depuis le serveur

On peut sûrement faire mieux en utilisant l'API History HTML5 mais je n'ai jamais essayé

Le routeur

Dans un routeur on définit un ensemble de route avec pour chacune une fonction de callback qui va être appelée lorsqu'elle sera dans la barre d'adresse (par navigation ou saisie)

Comme pour la vue, le contexte fourni lors du callback est le contexte du routeur (le bon)

On appelle à la fin de l'initialisation du routeur la méthode *Backbone.History.start* afin de faire fonctionner l'ensemble

Les routes peuvent être statiques (eg *#/media/create*) ou dynamiques (eg *#/media/edit/:id*)

Le routeur

Exemple :

- Exemple d'utilisation d'un routeur

```
var Router = Backbone.Router.extend({
    routes: {
        'media/create'   : 'onCreateMedia',
        'media/edit/:id': 'onEditMedia'
    },

    onCreateMedia: function () { /*Do something*/ },

    onEditMedia: function (id) { /*Do something*/ }
});

var router = new Router();
Backbone.History.start();
```

Et pour finir

Quelques conseils

- Soyez vigilant lors de l'utilisation des event listeners, créez systématiquement une méthode *dispose* dans chacun de vos objet où vous désaffecterez les event listeners que vous avez affecté lors de l'initialisation (ou après)
- Dans les vues essayez de cibler au maximum les éléments du DOM à changer et éviter de recourir systématiquement à la méthode *render* dès qu'il y a un changement (gain de performance significatif)
- N'hésitez pas à décomposer au maximum vos vue pour en avoir beaucoup de petites plutôt que peu de très grosses
- Vous avez le contrôle alors profitez-en, soyez vigilant sur vos actions et leurs impacts, surveillez vos performances