Part 4: Web client implementation [TOTRANSLATE]

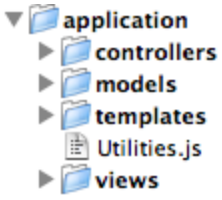
Implémentation d'une ressource côté client Javascript

Dans cette section, nous décrivons d'abord la manière d'ajouter un modèle côté Javascript. Celui-ci aura pour tâche de communiquer avec le serveur selon les principes REST. Nous décrivons ensuite la manière de l'utiliser via un exemple concret pour chaque opération possible (GET, POST, PUT, DELETE) et la façon de générer des vues HTML pour afficher les résultats.

Outils

L'interface Web Cytomine est une application écrite en Javascript. Lorsqu'un utilisateur se connecte à l'adresse du serveur avec un navigateur Web, le serveur lui envoie l'entièreté de l'application, dont les fichiers sources ont été fusionnés et minimisés dans un fichier unique appelé *application.js*. L'application fonctionne de manière autonome et contient le code logique pour communiquer avec l'API Web.

Pour structurer notre application et maximiser la maintenabilité à long terme, nous utilisons une bibliothèque Javascript appelée Backbone.js. Comme son nom l'indique, elle fournit un squelette de base à l'application, et fournit des outils précieux pour isoler le code qui s'occupe du traitement des requêtes HTTP (Controleur/Router), de la gestion des ressources (Model) et de la vue (View).



Nous détaillons ici brièvement certains de ces outils. Nous renvoyons à la doc Backbone.js le lecteur qui souhaite un complément d'informations :

- **Model** : objet qui contient les données d'une ressource (par exemple le nom et prénom pour une ressource *user*). Il connaît également l'URL de la ressource dont il contient les données. Il propose notament trois méthodes pour communiquer avec le serveur, toujours sur les principes REST :
    - *fetch()* : effectue une requête HTTP GET sur l'URL de la ressource. La méthode renvoie un modèle contenant les données de la ressource obtenue.
    - *save()* : effectue soit une requête POST si la ressource n'existe pas sur le serveur, soit une requête PUT si la ressource existe au préalable. Les données du modèle sont automatiquement envoyés via le corps (body) de la requête HTTP. La méthode renvoie soit le nouveau modèle, soit le modèle mis à jour.
    - *destroy()* : effectue une requête DELETE sur l'URL de la ressource. Le résultat est une suppression de la ressource côté serveur. La méthode indique si l'opération est un succès ou non.
- **Collection** : une collection est un ensemble de Model. Elle permet de contenir les modèles d'une même ressource, par exemple l'ensemble des *user*. Elle propose également des méthodes pour parcourir, comparer et trier les modèles.
- **Controller/Router** : un *Router* (l'ancien nom est *Controller*) permet de définir les routes (URLs) de l'application Web et de leur associer des actions. Ainsi, lorsqu'un utilisateur navigue sur une URL présente dans un *Router*, l'action associée est automatiquement exécutée par le *Router*.
- **View** : une vue (*View*) est un objet qui a pour tâche d'effectuer le rendu HTML d'un composant de l'interface. Elle est souvent, mais pas forcément, associée à un modèle ou à une collection. Elle a également la responsabilité de dispatcher les évènements que l'utilisateur déclenche dans la page HTML générée (survol d'un élément, click souris, ...).

Création d'un domaine et d'une collection

L'étape de création d'un modèle consiste à étendre l'objet Model présent dans la librairie Backbone et à définir la méthode *url()*. Cette méthode renverra les URL associées à la ressource dont on écrit le modèle, avec comme préfixe $CYTOMINEURL. Ces URLs sont définies par le serveur (URL mappings), comme expliqué précédemment. On peut également définir une méthode *validate()* pour imposer certaines contraintes. Cette méthode est appelée avant toute opération (*fetch()*, *save()* et *destroy()*). Dans le cas où elle échoue, la requête HTTP n'a pas lieu. Cela permet d'éviter de faire des requêtes inutiles côté serveur si la ressource n'est pas valide (exemple : champs requis inexistant). Notons également qu'il n'est pas nécessaire de définir les attributs liés à un *user* (firstname, lastname, ...) car ils seront dynamiquement définis en fonction de la réponse du serveur. Cela procure évidemment une grande flexibilité et un gain de temps non négligeable lors du développement de l'application.

Dans le cas présent, la méthode *url()* renverra :

- "api/user.json" si la ressource n'existe pas côté serveur.
- "api/user/ID.json si la ressource existe côté serveur. Dans ce cas, ID correspond à l'identifiant de la ressource.

Implémentation d'un modèle

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

    validate: function (attrs) {
        if (attrs.name) {
            if (!_.isString(attrs.name) || attrs.name.length === 0) {
                return "Name must be a string with a length";
            }
        }
    },

    url: function () {
        var base = 'api/user';
        var format = '.json';
        if (this.isNew()) return base + format;
        return base + (base.charAt(base.length - 1) == '/' ? '' : '/') + this.id + format;
    }
});
```

Dès lors que le modèle est définit, il devient très facile de manipuler les ressource *user*. Point important, le code résultant du choix de ces outils s'avère élégant et compréhensible.

Création d'un nouveau user

```
var user = new UserModel({
    "username": "johndoe",
    "firstname": "john",
    "lastname": "doe",
    "password": "doedoe2011"
});
```

Sauvegarde du *user* et implémentation des callbacks en cas de succès ou d'erreur

```
user = user.save({ //Requête POST
    success: function (model, response) {
        //la ressource a été ajoutée
    },
    error: function (model, response) {
        //la ressource n'a pas été ajoutée, une erreur s'est produite
    }
});
```

Obtention du *user* grâce à son identfiant

```
//pour cet exemple, id contient l'identifiant de l'utilisateur que l'on souhaite récupérer
var user = new UserModel({
    id: id
});
//obtention des données relatives à l'utilisateur
user = user.fetch({ //Requête GET
    success: function (model, response) {
        //la ressource a bien été récupéré, on peut la mettre à jour
    },
    error: function (model, response) {
        //code error
    }
});
```

Mise à jour du *user* et implémentation des callbacks en cas de succès ou d'erreur

```
  user.set({
      "firstname": "johnny"
  }); //mise à jour du firstname
  user.save({ //Requête PUT
      success: function (model, response) {
          //la ressource a été modifiée
      },
      error: function (model, response) {
          //la ressource n'a pas été modifiée, une erreur s'est produite
      }
  });
```

Suppression du *user* et implémentation des callbacks en cas de succès ou d'erreur

```
/ /!\Pour cet exemple, la variable id  contient l'identifiant de l'utilisateur précédemment sauvé /!\
var user = new UserModel({
    id: id
});
user = user.destroy({ //Requête DELETE
    success: function (model, response) {
        //la ressource a été supprimée
    },
    error: function (model, response) {
        //la ressource n'a pas été supprimée, une erreur s'est produite
    }
});
```

Il est également intéressant de définir une collection qui sera en mesure de contenir plusieurs instances de notre ressource. Il s'agit d'étendre la classe *Collection*, d'implémenter la méthode *url* et d'indiquer la ressource concernée (*UserModel*).

Implémentation d'une Collection

```
// define our collection
var UserCollection = Backbone.Collection.extend({
   model: UserModel, //on lui indique le modèle concerné

   url: function() {
      return "api/user.json"; //cette url renvoie la list de tous les utilisateurs
   }
});
```

Tout comme les modèles, une collection possède une méthode *fetch()*. Celle-ci effectue une requête HTTP GET sur l'URL associée à la collection. En cas de succès, la collection crée un tableau contenant une instance du modèle (*UserModel* dans ce cas-ci) pour chaque ressource présente dans la base de données.

Récupération de tous les *user*

```
var users = new UserCollection().fetch({
    success: function (collection, response) {
        //les modèles user ont bien été obtenus
    },
    error: function (collection, response) {
        //une erreur s'est produite
    }
});
```

Création de la vue

Pour rappel, une vue (*View*) est un objet qui a pour tâche d'effectuer un rendu HTML dans la page Web et d'intercepter les évènement sur celle-ci.
On passe généralement deux variables importantes à la vue :

- el : l'identifiant de l'élément HTML dans lequel le rendu sera fait
- model : une collection ou un modèle

Dans notre cas, il s'agit alors de définir un nouvel objet qui étend *View* et d'implémenter la méthode *render()*.

Implémentation de la vue

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

    tagName: "li",

    events: {
        "click .refresh": "refresh"
    },

    render: function () {
        var tpl = "<li>{{firstname}} {{lastname}}</li>"; //template html de base pour un user
        this.model.each(fonction (user) { //on suppose que this.model existe car passé dans le constructeur
            //rendu HTML du user. La variable html contientra alors : "<li>John doe</li>"
            //pour le user dont le username est "johndoe"
            var html = _.template(tpl, user.toJSON());
            //Enfin, on affiche le code généré dans la page HTML
            $("#" + this.el).append(html);
        });
    }

    refresh: function () {
        // mise à jour de la liste
    }

});
```

Un cas d'utilisation de la vue est expliqué dans la création du *Router*.

Création du Controller/Router

Comme dit précédemment, un *Router* permet de faire correspondre des urls à une action que l'utilisateur peut effectuer dans l'interface.
Nous prenons le cas d'un lien existant dans une page html dont la cible est "user/list". Lorsque l'utilisateur effectue un click sur ce lien, le routeur concerné détecte l'évènement et déclenche l'action associée. Dans ce cas-ci, l'action sera de lister les users. Un tel contrôleur se définit comme suit :

Implémentation d'un Routeur/Controlleur

```
var UserRouter = Backbone.Router.extend({

    routes: {
        "user/list": "list" // #user/list => list()
    },

    /* Fonction qui liste les users */
    list: function () {
        //Récupération des users grâce à l'objet UserCollection
        var users = new UserCollection();
        users.fetch({
            success: function (collection, response) {
                //les modèles user ont bien été obtenus, on peut donc créer la vue afin de les lister.
                //on instancie la vue en spécifiant l'élément HTML dans lequel le rendu sera fait (userList)
                //ainsi que la collection de users.
                new UserListView({
                    el: "usersList",
                    model: collection
                }).render();
            },
            error: function (collection, response) {
                //une erreur s'est produite
            }
        });
    }
});
```