**JAVASCRIPT**

**Modèle MVC**

I. ***EVENTS AND OBSERVING***

Les événements sont le cœur de l'application JavaScript. Ils déclenchent tout et constituent le premier point de contact entre l'utilisateur et l'application.

◦ ***Listening to events***

Les événements tournent autour d'une fonction appelée  : addEventListener.

addEventListener (

type, // event object => type d'événement à enregistrer

listener, // objet qui recevra la notification lorsqu'un événement du type spécifié se produit. C'est soit un objet implémentant EventListener, soit une fonction javascript

useCapture // type event in boolean (true => capturing  : les événements se propagent vers le bas  ; false => bubbling  : les événements se propagent vers le haut)

)

◦ ***Event ordering***

Netscape et Microsoft avaient des idées différentes sur l'ordre des événements parcourus sur un objet et ses ancêtres  :

1. Event capturing (Netscape), qui déclenche les gestionnaires d'événement à partir des ancêtres jusqu'à l'élément en question.

2. Event bubbling (Microsoft), qui déclenche les gestionnaires d'événement depuis l'élément courant pour se propager jusqu'aux ancêtres.

◦ ***Canceling events***

On peut stopper la propagation en appelant la méthode stopPropagation() sur l'objet Event.

On peut aussi annuler certaines actions par défaut déclenchées par les navigateurs comme le chargement d'une page grâce à la méthode preventDefault().

◦ ***Event libraries***

Il existe de nombreuses incohérences lors de la gestion d'événements par les navigateurs.

On conseille d'utiliser des librairies comme jQuery.

Grâce à cette librairie, nous pouvons par exemple créer des «  custom events  » via sa méthode trigger() et la lier à différents éléments DOM via sa méthode bind().

Exemple  :

var exports = this;

(function($){

exports.Events = {

bind: function(ev, callback) {

var evs = ev.split(" ");

var calls = this.\_callbacks || (this.\_callbacks = {});

for (var i=0; i < evs.length; i++)

(this.\_callbacks[evs[i]] || (this.\_callbacks[evs[i]] = [])).push(callback);

return this;

},

trigger: function() {

var args = $.makeArray(arguments);

var ev = args.shift();

var list, calls, i, l;

if (!(calls = this.\_callbacks)) return this;

if (!(list = this.\_callbacks[ev])) return this;

for (i = 0, l = list.length; i < l; i++)

if (list[i].apply(this, args) === false)

return false;

return this;

}

};

})(jQuery);

Events.bind("hail", function(type){

alert("Long live the " + type)

});

Events.trigger("hail", "Users");

◦ ***Delegating events***

On peut ajouter un écouteur sur un élément parent, vérifiant les événements sur les enfants, plutôt que d'attribuer un écouteur sur chaque élément (parent et enfants compris) grâce à la méthode delegate() de jQuery.

~~$(“ul li“).click(function(){/\*...\*/})  ;~~ car on ajoute un écouteur à chaque élément li

$(“ul“).delegate(“li“, “click“, /\*...\*/)  ; on ajoute qu'un seul écouteur valable pour tous les éléments li

II. ***MODELS AND DATA***

**AVANT : SERVER-SIDE**

**APRES : CLIENT-SIDE**

1. Requête de la page

2. Accès au serveur

3. Récupération des données issues d'une BDD

1. Pas de requête/réponse

2. Pas d'accès au serveur

3. Phase de **transition**  : les données sont récupérées à distance et sont stockées temporairement côté client, dans la mémoire.

4. Avantages  :

1. L'accès à la donnée est côté client est quasi instantanée.

2. Impression d'instantanéité pour l'utilisateur.

◦ ***MVC and Namespacing***

Toute la logique associée à la manipulation et au comportement des données doit être organisé dans le Model.

**Données**

**M**

**Model**

**Logique**

**V**

**View**

**Comportement**

**C**

**Controller**

**Manipulation**

Les Model doivent être séparées des View et des Controller. Pour cela, on utilise le namespacing. Cela évite à notre code de s'entremêler et de créer des conflits avec les fonctions et les callbacks.

◦ ***Building an ORM***

ORM  : Object-Relationnal Map. C'est une sorte de traceur, un «  mapper  » en anglais.

1. Il est normalement très utilisé dans d'autres langages que le javascript.

2. Il est très pratique pour gérer les données.

3. Il est très pratique, aussi, pour gérer les Model en javascript.

On peut par exemple lier un ORM avec un serveur distant ou avec un élément HTML. Tout changement sur les instances des Model se reflètera à leur objet lié.

C'est un objet utilisé comme une couche enveloppant des données. L'avantage de cette extra couche est que l'on peut augmenter la donnée de base avec des fonctionnalités poussées, avancées et personnelles.

◦ ***Prototypal Inheritance***

On va utiliser Object.create() pour construire un ORM plutôt que son constructeur combiné avec le mot-clé «  new  ». Cette méthode nous permet d'utiliser l'héritage basée sur le prototype et non la classe.

Object.create(prototypal object){

// return a new object inhereted from protypal object)

}

Ainsi, on crée un objet Model qui va nous servir à instancier des nouveaux objets héritant de ce dernier.

Object.create() a été ajouté à ECMAScript 5, alors s'il n'est pas implémenté dans les navigateurs comme IE, on peut créer et ajouter une nouvelle fonction à Object comme ceci (technique employée à Douglas Crockford)  :

if(typeof Object.create !== “  function  “)

Object.create = function(o) {

function F(){}

F.prototype = o  ;

return new F()  ;

}  ;

var Model = {

inherited: function(){},

created: function(){},

prototype: {

init: function(){}

},

// Returns a new object, inheriting from the Model object

create: function(){

var object = Object.create(this);

object.parent = this;

object.protoype = object.fn = Object.create(this.prototype);

object.created();

this.inherited(object);

return object;

},

// Returns a new object, inhereted from Model.prototype, an instance of the Model object

init: function(){

var instance = Object.create(this.prototype);

instance.parent = this;

instance.init.apply(instance, arguments);

return instance;

}

};

◦ ***Adding ORM properties***

On crée des méthodes permettant aux objets créés à partir de la classe Model d'hériter de ses propriétés.

◦ ***Persisting records***

Pour garder les enregistrements des données, il nous faut garder une référence des instances créées pour pouvoir y accéder plus tard.

On crée un objet Model.records = {}  ;

◦ ***Adding ID support***

Tout enregistrement doit avoir un ID unique, comme dans une BDD. Il faut en générer un automatiquement grâce au générateur GUID.

◦ ***Addressing references***

Allouer des références.

Pour ne pas modifier la classe, il faut cloner les instances lorsque l'on utilise des méthodes et/ou des propriétés de classe, sinon on fait référence à la classe elle-même, et on la modifie alors  !

Il faudra donc dupliquer l'objet à chaque fois que l'on crée ou met à jour un enregistrement.

◦ ***Loading in data***

Concept  :

*Typiquement, un sous-ensemble de données est chargé lorsque l'application démarre et d'autres données sont chargées après interaction.*

*L'idéal serait de tout charger lors du chargement de la page, comme ça pour chaque interaction s'enchaîne immédiatement la donnée. Mais cela est impossible car il y a beaucoup trop de données en général pour être chargées en mémoire du navigateur.*

*Le préchargement de données est alors crucial. Cependant, il ne sert à rien de précharger des données redondantes qui risquent de ne jamais être utilisées.*

*Alors, il faut prédire les données utiles et utilisables par l'utilisateur.*

*Moins il y a de délai, mieux c'est.*

*Quand on récupère une nouvelle donnée, il faut s'assurer que l'UI n'est pas bloqué. Utiliser aussi un indicateur de chargement, mais être certain que l'interface reste utilisable.*

Les données peuvent être en ligne (inline) dans la page initiale. C'est l'idéal si et seulement si une petite somme de données au total car sinon la taille de la page devient trop lourde en cas de données massive.

Les données peuvent être chargées par des requêtes HTTP séparées à travers AJAX ou JSONP. Ces deux techniques permettent aussi de mettre en mémoire-cache la page HTML plutôt que de créer un rendu dynamiquement pour chaque requête.

◦ ***Including data inline***

Il suffit de rendre un objet JSON dans la page, JSON ayant la même structure qu'un objet JavaScript.

◦ ***Loading data with AJAX***

On utilise soit la classe XMLHttpRequest soit la méthode de jQuery $.ajax()  ;

Parmi les différents paramètres de la méthode ajax, le “type“  prend en valeur par défaut GET. Il peut avoir comme valeur POST. Pour ses autres valeurs, DELETE, HEAD, OPTIONS, PUT, elles ne sont pas toutes supportées par les différents navigateurs et il faut activer ces verbes sur IIS.

La limitation de la technologie AJAX est le «  same origin policy  »  : il s'agit d'une restriction des requêtes aux mêmes domaine, sous-domaine et port que ceux de l'adresse de la page en cours. Pour rendre plus souple cette limitation, il existe un standard nommé CORS  : cross-origin resource sharing.

Il faut aussi prévoir la sécurité concernant le «  cross-domain request  ».

◦ ***Populating ORM***

Pour peupler l'ORM, il faut récupérer les données du serveur (loading) et mettre ensuite à jour les enregistrements du Model.

Cette méthode est exécutée dans la fonction de callback de l'appel AJAX.

◦ ***Storing data locally***

Avant, pour enregistrer des données localement, on avait recours aux cookies. Cependant, le stockage des données était limité en quantité et il y avait un envoi des données au serveur pour chaque requête, donc un fort coût en terme de performance.

Maintenant, HTML5 a introduit un support gérant le stockage de données en local  : localStorage (le stockage dure même si le navigateur est fermé) et sessionStorage (le stockage ne dure que lors de la session ouverte du navigateur).

Les avantages sont multiples. La donnée est stockée côté client et plus côté serveur. Il n'y a jamais d'envoi des données au serveur. Le stockage des données en quantité est très importante (5Mo minimum ?).

Les objets localStorage et sessionStorage ont une API issue du WebStorage.

La data est stockée en type “String“. Donc, il faut prévoir une conversion pour les types Object et Number vers un type String avant de les enregistrer. On parle de sérialisation JSON et désérialisation quand on les récupère.

◦ ***Adding local storage to our ORM***

Les enregistrements peuvent durer, notamment entre différents rafraîchissements de page, en ajoutant le support local storage à notre ORM.

Pour utiliser l'objet localStorage, nous devons sérialiser nos enregistrements dans une chaîne JSON. Seulement, il faut limiter la liste des propriétés à sérialiser car sinon on se retrouve avec les propriétés cachées (prototype, parent...) et on devient verbeux.

Pour cela, on va créer une propriété “attributes“ au Model. Or, comme chaque Model a des attributs différents, et que de plus on ne peut pas partager la référence identique, on ne va pas sélectionner la propriété “attributes“ sur le Model directement, mais une fois que le Model aura été créé, comme pour la propriété “records“.

Ensuite, on ajoute une méthode toJSON() sur les instances des Model. La méthode va trouver l'objet à sérialiser plutôt que de sérialiser tous les “records“.

Enfin, nous n'avons plus qu'à ajouter le local storage au Model grâce à deux fonctions  : saveLocal() et loadLocal().

◦ ***Submitting new records to the server***

Maintenant que nous avons créé l'objet “attributes“, pour soumettre les nouveaux enregistrements au serveur, il faut juste utiliser le POST pour envoyer ces attributs d'enregistrement au serveur. Si nous suivons les conventions REST, nous voudrons faire un “HTTP POST“ quand nous créons un enregistrement et une requête “PUT“ quand nous voulons mettre à jour l'enregistrement.

III. ***CONTROLLERS AND STATE***

IV. ***VIEWS AND TEMPLATING***