Rapport de stage
Programmation Enyo en OCaml

Vincent Botbol

8 août 2012

Table des matières

Chapitre 1

Documentation Enyo

1.1 Présentation

Enyo est un framework Javascript orienté-objet. Il privilégie un style de développement basé sur une encapsulation arborescente de composants.

Dans sa première version, Enyo fut employé pour le développement d'application des tablettes HP TouchPad fonctionnant sous webOS. Après un succès mitigé, HP décida de transformer webOS en projet *open source* devenant ainsi : Open WebOS.

Enyo passe ainsi en version 2.0. Profitant de sa conception en Javascript, l'engouement actuel de ce dernier permet aux développeurs de déployer leurs applications sur les plateformes le supportant, c'est-à-dire, tout appareil se voulant grand public (Android, iOS, WindowsPhone, etc.)

On peut attribuer à Enyo des avantages tels que sa prise en main aisée grâce au style de programmation proche d'une vision du DOM et le poids relativement faible de sa bibliothèque.

1.2 Débuter en Enyo

Il est important de distinguer deux phases dans le développement d'une application Enyo.

- La première, dite de création, consiste à définir les composants, leurs positions dans la hiérarchie de l'application ainsi que le modèle.
- La seconde est celle du déploiement. Permettant à la bibliothèque Enyo de générer, à partir du squelette décrit par la phase de création, l'objet destiné à être instancier puis afficher.

La définition des composants de l'application consiste, vulgairement, à déclarer des objets Javascript possédant des propriétés qui auront un sens au moment d'étendre l'application.

Par exemple, définissons un simple morceau de texte contenant la chaine "Bonjour monde !" :

```
1  var monApplication = {kind: "Control",
2                        name: "MonApp",
3                        content: "Bonjour monde!"}
```

Ici, nous avons simplement créé un objet Javascript muni de propriétés suffisantes pour renseigner Enyo sur le type d'objet à créer.
Explicitons ces propriétés :

- *kind* définit le type d'objet Enyo que nous souhaitons créer.
- *name* nomme l'objet que nous sommes en train de créer. C'est par la valeur de cette propriété que nous pourrons accéder à un composant ou encore l'instancier. [1]
- *content* attribut un texte au composant. Ceci n'a de sens que parce que l'on manipule un objet Enyo de type "Control" ou un de ses descendants qui possède donc cette propriété.

Désormais défini, il faut passer le composant à la méthode *kind* d'Enyo pour finaliser l'application et la rendre instanciable.

```
1  enyo.kind(monApplication);
```

A la suite de cet appel, on pourra constater qu'une nouvelle classe *MonApp* est présente dans l'environnement. Il ne reste donc plus qu'à l'instancier pour pouvoir ensuite l'afficher.

1. Il est important de noter que si placée dans un composant en racine de l'application, la propriété *name* déterminera le nom de la classe générée. Autrement, le nom d'un sous-composant servira de clé à une table d'association définie sur le composant racine de l'application servant à récupérer les divers sous-composants

```
1   var app = new MonApp();
2   app.renderInto(document.body);
```

En s'intéressant de plus près à l'objet "app", on peut constater qu'il possède un nombre important de méthodes et de propriétés que nous n'avons pas forcément définies à la création, telle que la méthode *renderInto* que nous venons d'utiliser.

Ces méthodes ont été ajoutées par induction lors de la construction de l'application par la méthode *enyo.kind*.

Si le champ *kind* spécifie un "Control", il est donc nécessaire à Enyo de lui rajouter toutes les propriétés et méthodes que possède un Control Enyo qui n'ont pas été définies par l'utilisateur.

1.3 Ajout de sous-composants

Enyo suivant une logique d'arborescence de composants. Lorsque l'on crée son application, il est nécessaire de pouvoir lui ajouter des sous-composants. On réalise cela en définissant la propriété *components* avec en valeur le tableau des composants que nous souhaitons.

Imaginons ici que l'on souhaite créer une application consistant d'un champ de saisie et d'un bouton de validation :

```
1  enyo.kind({kind:"Control",
2             name:"MonApp",
3             components: [
4                          {kind:"Input"},
5                          {kind:"Button", content:"ok"}
6                         ]
7            });
```

La fonction kind va récursivement créer tous les sous-composants de la hiérarchie ainsi que leur méthodes associées.

Comme précédemment, pour l'afficher, on instancie la nouvelle classe ainsi générée et on lui applique une méthode d'affichage :

```
1   var app = new MonCadre();
2   app.renderInto(document.body);
```

Si l'on observe le code html après ces appels, on s'aperçoit qu'enyo a injecté dans le DOM une balise html de type dans laquelle sera placée, respectivement, deux balises : un <input type="text" ..> et un <button>.

Notons que nous n'avons pas défini la propriété *name* des sous-composants. Rappelons que le nom d'un composant sert uniquement à l'accession de celui-ci par les autres composants de la hiérarchie ou encore pour son instanciation. Cependant, lors du kind, seule la racine de l'arbre sera étendue à l'environnement.

Il faut aussi noter que si la propriété name n'est pas définie, Enyo s'en charge en lui donnant un label généré ("input0" et "button0" ici). Il n'est évidemment pas recommandé d'utiliser ces derniers pour y accéder.

1.4 Événements

Reprenons l'exemple précédent et supposons désormais que l'on souhaite, lorsque l'on clique sur le bouton, afficher un message d'alerte.

```
enyo.kind({kind:"Control",
           name:"MonCadre",
           components: [
             {kind:"Input", value:""},
             {kind:"Button", content:"ok",
              handlers:{ontap:"monCallback"},
              monCallback:function(sender, event){
                alert("ok");
                return true;
              }
             }
           ]
          });
```

Au bouton, nous avons rajouter une propriété *handlers*. Cet objet sert à définir tous les événements que l'objet va traiter. Cela peut-être des événements du dom (onkeyup, onload, ...) comme des événements spécifiques à la plateforme sur laquelle l'application est déployée (l'inclinaison d'une tablette par exemple). Ici, nous utilisons l'événement "ontap", alias d'"onclick".

A chaque propriété-évenement est attaché une chaine contenant le nom de la fonction a appeler.

Nous définissons donc la fonction click qui prend en premier argument l'emetteur de l'événement et l'événement envoyé contenant ses propriétés.

La valeur de retour de cette fonction spécifie si l'évenement doit s'arrêter ou remonter ("bubble") aux composants parents. On retourne vrai pour l'empêcher de remonter et faux pour le laisser "bullé" sur ses ancêtres.

On notera qu'en général, les fonctions de traitement ne pas censées devoir "buller" sont définies à la racine de l'arbre qui possède un accès à tous ses sous-composants grâce à sa table de hash "$".

Il existe quelques spécificités pour la manipulation d'évenements :

- Si la fonction de traitement n'est pas trouvée, l'événement est remonté dans l'arbre de composants jusqu'à trouver une fonction dont le nom correspond.
- Il n'est nécessaire de déclarer le bloc *handlers* que lorsque la fonction de traitement se situe dans le même objet que celui-ci. Par exemple, cette définition est parfaitement valable :

```
    enyo.kind({kind:"Control",
               components:[{kind:"Bouton",
                            ontap:"traiteClique"}
                           ],
               traiteClique:function(sender, event){
                 alert("Click recu"); return true
```

```
7                   }
8       });
```

– Il est possible de propager un événement manuellement en utilisant la fonction *send(nomEvenement, evenement)* de l'objet *enyo.Signals*. Cet événement est remis à tous les objets de type *kind :"Signals"*. Il suffit alors à l'objet-récepteur de posséder un sous-composant de ce type et d'implémenter la fonction de traitement lié au handler du sous-composant. (voir exemple)

1.5 Propriétés published

Les propriétés placées dans l'objet "published" du composant sont en général utilisées pour la logique de l'application. Certains "published" sont déjà présents dans les objets Enyo. Par exemple, la propriété "content" de l'objet "Control" est elle-même publiée.

Enyo traite ces propriétés comme étant des variables publiques et génère automatiquement : getter, setter ainsi qu'une méthode valueChanged appelée à chaque appel du set avec une nouvelle valeur.

En résumé, un composant déclaré ainsi :

```
1   {kind:"app", published:{maVariable:0}}
```

est équivalent à :

```
1   {kind:"app", maVariable:0,
2     getMaVariable:function(){..},
3     setMaVariable:function(){..},
4     maVariableChanged:function(){}}
```

Cependant il reste possible d'initialiser les valeurs de propriétés déjà publiées hors de l'objet *published* de la même façon dont nous procédions avec le champ *content* jusqu'à présent. Cela aura uniquement pour effet de masquer la première initialisation que le constructeur Enyo aura effectué.

Note : la propriété *monObjet.published.maVariable* ne sert qu'à indiquer au constructeur d'objet Enyo qu'il doit générer la variable et sa barrière d'abstraction. Des effets de bords sur cette variable après instanciation n'auront aucun effet.

1.6 Copie, partage et réutilisation de composants

Il est possible de réutiliser des composants déjà créés. Il faut cependant faire attention à ne pas lui attribuer de nom. Dans le cas échéant, le constructeur considèrera que l'objet existe déjà et tous les traitements visant une des occurences de ce nom seront appliqués au premier objet défini.

```
var unBouton = {kind:"Button", name:"bouton1",
                handlers:{ontap:"clique"},
                content:"un bouton",
                clique:function(){
                  this.setContent("Reception du click")
                }
};

enyo.kind({name:"App", components:[unBouton,unBouton]});
```

Dans cet exemple, au moment de l'instanciation un message d'avertissement concernant une collision de nom est émis mais les deux boutons s'afficheront correctement. Cependant, on constate bien que les clicks émis sur l'un ou l'autre des boutons n'auront pour effet de changer que le premier des deux boutons.

Typiquement, si l'on souhaite réutiliser un composant déjà implémenté, il est recommandé de procéder ainsi :

```
enyo.kind({kind:"Button", name:"MonBouton",
           handlers:{ontap:"clique"},
           content:"un bouton",
           clique:function(){
             this.setContent("Reception du click")
           }
});

enyo.kind({name:"App",
           components:[
             {kind:"MonBouton", name:"bouton1"},
             {kind:"MonBouton", name:"bouton2"}]
});
```

Une nouvelle classe *MonBouton* est alors étendu à l'environnement qui pourra être "Kindé" comme un objet Enyo standard dans une définition de composant.

Chapitre 2

Composants Enyo

2.1 Hiérarchie des objets

2.2 Bibliothèques additionnelles

2.2.1 Onyx

2.2.2 Canvas

2.2.3 Layout

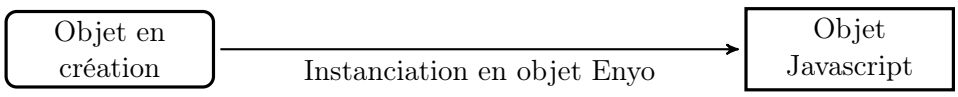
Chapitre 3

Interfaçage d'Enyo en OCaml

Pour une adaptation en OCaml d'Enyo, il est nécessaire d'établir une communication entre les deux langages. Cela présente une première difficulté étant donné le peu de points communs existants. Nous pouvons néanmoins résoudre le problème en utilisant Js_of_ocaml, un compileur de byte-code OCaml vers Javascript développé par l'équipe d'Ocsigen. Nous obtenons ainsi un moyen fiable d'assurer une traduction, de déclarer des objets Javascript ou encore d'attacher des méthodes à ces derniers, le tout en restant dans le monde OCaml. La transposition assurée, nous pouvons nous concentrer sur la modélisation d'Enyo.

Le point critique de l'interfaçage se situe dans l'utilisation des traits dynamiques de Javascript, par exemple, utiliser la méthode *setContent(value)* sur un *kind :"Control"* qui ne possèdera cette méthode qu'après instanciation est rédibitoire pour une transcription directe en OCaml.

Après considérations, j'ai choisi de m'arrêter sur un modèle reposant sur une différenciation d'un objet en construction de celui d'un objet instancié. On représente ainsi un objet *Enyo* en OCaml grâce aux propriétés le définissant.



L'extension de l'objet en cours de création avec propriétés et méthodes personnalisées est abandonné. Cet aspect d'Enyo est nécessaire pour la partie logique de l'application, ce dont nous pouvons nous abstraire puisque nous souhaitons un modèle de calcul reposant sur un programme OCaml classique. A partir de là, les objets de l'interface de programmation (API) d'Enyo et leurs méthodes sont suffisants si l'on considère utiliser Enyo comme une interface graphique multi-plateforme pour OCaml. Il est toutefois nécessaire de prendre en compte les évenements.

Dans une première partie (3.1), nous présenterons la structure choisie.

Ensuite, la définition des propriétés et leur implémentation (3.2)
Dans un troisième temps, ... (3.3)

3.1 Représentation des objets

3.2 Gestion des propriétés

3.3 Gestion des évenements

Chapitre 4

Langage de Description d'Interface (IDL)

Chapitre 5

Applications diverses