Le modèle évènementiel de VEGAS : Mode d'emploi - Partie 1

par Marc Alcaraz (Mes articles)

Date de publication : 03/03/2008

Dernière mise à jour :

1ère partie d'un tutoriel qui regroupe de façon générale toutes les possibilités du package **vegas.events.*** de la version AS2 de **VEGAS**

I - Généralités

J'ai déjà écrit 2 tutoriels sur le modèle événementiel de VEGAS (Gestion des événements dans le framework VEGAS - Partie **1** et **2**) mais il y a tellement de chose à dire à ce sujet que je vais écrire un nouveau tutoriel qui regroupera de façon générale toutes les possibilités du package vegas.events.* de la version AS2 de VEGAS.

Pour rappel, le modèle événementiel de VEGAS est basé sur le **DOM2/3 du modèle événementiel du W3C**.

Les versions **AS2** et **SSAS** du modèle événementiel de VEGAS sont totalement compatibles. La version **AS3** est compatible mais reste basée sur le système événementiel proposé par Adobe dans le framework AS3.

Vous pouvez avant de lire la suite de ce tutoriel prendre le temps de consulter le contenu de la documentation en ligne de **VEGAS** au niveau du package **vegas.events** : **http://www.ekameleon.net/vegas/docs/**

II - Définitions

II-A - Event

L'interface **vegas.events.Event** est utilisée pour diffuser des informations contextuelles via des événements propagés dans un flux d'événements spécifiques. Un objet qui implémente cette interface doit généralement posséder en premier paramètre de sa classe un **type**_. D'autres informations contextuelles spécifiques peuvent être passées dans l'événement pour être ensuite utilisées par les différents écouteurs qui recevront ces événements. Cette interface permet d'implémenter correctement tout objet de type **EventListener** au niveau de sa fonction **handleEvent(e:Event)**.

II-B - EventListener

L'interface **EventListener** définit la méthode principale utilisée par les écouteurs pour intercepter les événements durant une propagation d'événements. Les utilisateurs implémentent des objets de type **EventListener** et enregistrent ensuite leurs écouteurs sur un objet de type **EventTarget** via la méthode **addEventListener**. Les utilisateurs peuvent également supprimer un **EventListener** avec la méthode **removeListener** quand ils n'ont plus besoin de diffuser des événements.

II-C - EventDispatcher

La classe **EventDispatcher** est le centre névralgique du modèle événementiel au coeur de vos applications. Il est possible via cette classe ou les instances de cette classe d'enregistrer certains événements et de propager un événement avec la méthode **dispatchEvent()**.

II-D - EventTarget

L'interface **EventTarget** est implémentée par tous les objets qui doivent supporter la diffusion des événements via le modèle événementiel (DOM) défini par **VEGAS**. Cette interface définit les méthodes principales que doivent avoir un diffuseur d'événements pour enregistrer, désenregistrer et diffuser des événements.

II-E - FrontController

Le modèle de conception **FrontController** définit un simple diffuseur d'événements (**EventDispatcher**) responsable de toutes les requêtes principales d'une application.

Un **FrontController** centralise toutes les **commandes** de l'application via une représentation modulaire indépendante de toutes les fonctionnalités générales de l'application telles que les mises à jour des vues, les récupérations de données, les interactions avec les modèles de l'application, etc. L'intérêt d'utiliser un **FrontController** réside dans la souplesse d'utilisation des différentes commandes totalement indépendantes les unes des autres et le fait qu'une modification d'une fonctionnalité entraîne une très petite modification de l'application sans influencer sur le reste du code.

II-F - IEventDispatcher

Cette interface hérite de l'interface **EventTarget** et contient toutes les méthodes indispensables de la classe **EventDispatcher**.

III - Introduction

Pour utiliser la classe **EventDispatcher** de **VEGAS**, il faut au minimum créer 3 types d'objets : le diffuseur d'événement (**EventDispatcher**), l'écouteur(**EventListener**) et un événement (**Event**).

Pour expliquer l'utilisation de la classe **EventDispatcher**, je vais commencer par écrire quelques exemples très simples :

III-A - Créer un objet de type EventListener

L'interface **EventListener** doit être implémentée par vos écouteurs avec la présence de la méthode **handleEvent(e:Event)** dans vos objets. Par exemple, vous pouvez rapidement créer une classe qui servira pour tester la propagation des événements via tous les diffuseurs d'événements dans votre application.

```
import vegas.events.Event ;
import vegas.events.EventListener;

/**
 * The DebugHandler class.
 */
class test.events.DebugHandler implements EventListener
{

    /**
     * Creates a new DebugHandler instance.
     */
    public function DebugHandler(name)
    {
        _name = name || "default" ;
    }

    /**
     * Returns tne name of the debugger.
     */
    public function getName():String
    {
        return _name || "" ;
    }

    /**
     * Handles the event.
     */
    public function handleEvent(e:Event)
    {
        trace( "-------- DebugHandler : Event has been triggered." ) ;
        trace( "Event type     : " + e.getType()    ) ;
        trace( "Event target   : " + e.getTarget()  ) ;
        trace( "Event context  : " + e.getContext() ) ;
    }

    /**
     * Returns the String representation of the object.
     */
     public function toString():String
     {
          return "[DebugHandler name:" + _name + "]" ;
     }

     /**
      * The internal private name property of this instance.
      */
     private var _name:String ;
```

```
}
```

Quand l'écouteur que nous venons de définir reçois un événement, nous allons afficher le **type** de l'événement, son contexte et la cible de son diffuseur.

III-B - Enregister un écouteur

Pour enregistrer un écouteur pour qu'il reçoive des événements c'est très facile. Tout d'abord, il faut utiliser une instance de type **EventDispatcher** . Cette instance implémente les méthodes de l'interface **EventTarget** or **IEventDispatcher**, en principe on utilise une instance directe de la classe **EventDispatcher** ou une classe qui hérite de celle-ci.

Nous pouvons maintenant dans nos exemples créer une instance de la classe **DebugHandler** définie dans le code ci-dessus et passer celle-ci dans la méthode **addEventListener** du diffuseur d'événements. La méthode **addEventListener** nécessite au minimum deux paramètres (les autres paramètres sont optionnels).

- Le premier paramètres est le **type** des événements que l'écouteur pourra recevoir.
- Le second paramètre la référence de l'écouteur (**EventListener**).

```
import vegas.events.EventListener ;
import vegas.events.EventDispatcher ;

import test.events.DebugHandler ;

var listener:EventListener = new DebugHandler();

var disp:EventDispatcher = new EventDispatcher() ;
disp.addEventListener( "onLogin", listener ) ;
```

III-C - Diffuser un événement.

Pour diffuser un événement, il faut appeler la méthode **dispatchEvent()** et passer en paramètre le type de l'événement en premier paramètre (obligatoire de la méthode)

```
import vegas.events.BasicEvent ;
import vegas.events.EventListener ;
import vegas.events.EventDispatcher ;

import test.events.DebugHandler ;

var listener:EventListener = new DebugHandler();

var disp:EventDispatcher = new EventDispatcher() ;
disp.addEventListener( "onLogin", listener ) ;

disp.dispatchEvent( new BasicEvent("onLogin", this, "the context") ) ;
```

La méthode **dispatchEvent()** possède une déclaration bien spécifique :

```
dispatchEvent( event , isQueue:Boolean, target, context):Event
```

Le premier paramètre est un instance de type **Event** (de façon général une classe qui hérite de la classe **BasicEvent** ou directement une instance de type **BasicEvent**).

Le premier paramètre peut être aussi une simple chaîne de caractères qui représente le type de l'événement à diffuser. La méthode **dispatchEvent** créera alors automatiquement un instance de type **BasicEvent**.

```
import vegas.events.BasicEvent ;
import vegas.events.Event ;
import vegas.events.EventListener ;
import vegas.events.EventDispatcher ;

import test.events.DebugHandler ;

var listener:EventListener = new DebugHandler();

var disp:EventDispatcher = new EventDispatcher() ;
disp.addEventListener( "onLogin", listener ) ;

var e:Event = disp.dispatchEvent( "onLogin", this ) ;

trace(e) ; // output : [BasicEvent]
```

La méthode **dispatchEvent()** utilise en interne la classe utilitaire vegas.util.factory.EventFactory pour créer un nouvel événement.

Il est possible d'utiliser les options **target** et **context** en paramètre pour compléter la création de l'événement.

Le second paramètre de la méthode **dispatchEvent** définit la possibilité de mettre en file d'attente (queue) votre événement.

La méthode **dispatchEvent()** renvoie une référence de l'événement diffusé par la méthode.

IV - Exemple basique

Je vais essayer de mettre en place dans ce chapitre un exemple basique d'utilisation du modèle événementiel de VEGAS. En essayant d'aller un peu plus loin qu'avec l'exemple précédent.

Imaginons que nous devions mettre en place un système d'authentification dans notre application. Nous pourrions l'implémenter comme ceci :

```
/**
 * The Author class.
 */
class test.events.Author
{

    /**
     * Creates a new Author instance.
     */
    public function Author() {}

    static private var USER:String = "ekameleon" ;

    static private var PASS:String = "mypass" ;

    /**
     * Authentificate the specified user.
     */
    public function authenticate(user:String, pass:String):Boolean
    {

        if (user == Author.USER && pass == Author.PASS )
        {

           _username = user ;
            return true ;

        }

        return false ;
    }


    /**
     * Clear the authentification of the current author.
     */
    public function clearAuthentification():Void
    {
        _username = null;
    }

    /**
     * Returns the name of the user.
     */
    public function getUsername():String
    {
        return _username ;
    }

    /**
     * Returns the String representation of the object.
     */
    public function toString():String
    {
        var txt:String = "[Author" ;
        if (_username)
```

```
            {
                txt += ", name:" + _username ;
            }
            txt += "]" ;
            return txt ;
        }

        private var _username:String ;

    }
```

Cet exemple est un exemple extrêmement simple d'authentification pour illustrer comment nous pourrions implémenter réellement ce type de fonctionnalité via le modèle événementiel de VEGAS.

Maintenant il faut afficher une page d'enregistrement (login) en actionscript ? Il faut également écrire dans un système de log, l'historique des connexions bonnes ou mauvaises à l'application ? Nous pouvons alors implémenter un script comme ceci :

```
import test.events.Author ;

var user:Author = new Author() ;
var result:Boolean = user.authenticate( username, password ) ;
if ( result )
{
    trace("log success....") ;
}
else
{
    trace("log failed...") ;
}
```

Le code ci-dessus fonctionne, mais il va être rapidement plus intéressant d'utiliser une approche plus flexible utilisant la classe **EventDispatcher** et le package **vegas.events** :

```
import test.events.Author ;

var disp:EventDispatcher = new EventDispatcher() ;

var user:Author = new Author() ;
var result:Boolean = user.authenticate( username, password ) ;
if ( result )
{
    disp.dispatchEvent("onLogin", user) ;
}
else
{
    disp.dispatchEvent("onFail", user) ;
}
```

Au lieu de fournir tout de suite l'information sur l'enregistrement de l'utilisateur, nous avons à présent la possibilité d'abonner des écouteurs au système événementiel. Ce procédé permet facilement d'ajouter ou d'enlever des fonctionnalités à notre application sans alourdir inutilement le code après l'authentification de l'utilisateur.

Mettons en place un écouteur avec la classe **AuthorLogger** suivante :

```
import test.events.Author ;

import vegas.events.Event ;
import vegas.events.EventListener ;

class test.events.AuthorLogger implements EventListener
```

```
{

    /**
     * Creates a new AuthorLogger instance.
     */
    public function AuthorLogger()
    {
        //
    }

    /**
     * Handle the event.
     */
    public function handleEvent(e:Event)
    {

        var type:String   = e.getType();
        var author:Author  = Author( e.getTarget() ) ;
        var name:String   = author.getUsername() ;

        trace("AuthorLogger : Event has been triggered");

        trace( "event-type   : " + type ) ;
        trace( "event-target : " + author ) ;
        trace( "author-name  : " + name ) ;

        trace("---") ;

    }
}
```

Pour finir l'exemple, nous allons remanier légèrement le script précédent dans notre application :

```
import vegas.events.BasicEvent ;
import vegas.events.Event ;
import vegas.events.EventDispatcher ;
import vegas.events.EventListener ;

import test.events.Author ;
import test.events.AuthorLogger ;

var log:EventListener    = new AuthorLogger() ;
var disp:EventDispatcher = new EventDispatcher() ;

disp.addEventListener("onLogin" , log) ;
disp.addEventListener("onFail"  , log) ;

var user1:Author = new Author() ;
var result:Boolean = user1.authenticate("ekameleon", "mypass") ;
if ( result )
{
    disp.dispatchEvent( new BasicEvent("onLogin", user1) ) ;
}
else
{
    disp.dispatchEvent( new BasicEvent("onFail", user1) ) ;
}

var user2:Author = new Author() ;
var result:Boolean = user2.authenticate("ekameleon", "otherPass") ;
if ( result )
{
    disp.dispatchEvent( new BasicEvent("onLogin", user2) ) ;
}
else
{
    disp.dispatchEvent( new BasicEvent("onFail", user2) ) ;
```

```
    }
```

Avec cette solution, vous pouvez observer une séparation simple et efficace de la gestion des logs de l'application. Il sera ensuite pas plus compliqué d'ajouter des écouteurs spécialisés à notre application pour modifier l'aspect graphique en fonction du résultat de l'enregistrement de l'utilisateur, etc. au moment de l'utilisation de la méthode authenticate() l'événement diffusé ira vers l'écouteur spécialisé et lancera les actions nécessaires à la suite logique de l'application.

Nous bénéficions avec cette technique des points suivant :

- Il est possible de supprimer toutes les fonctions de debug , en supprimant juste l'appel de la méthode **addEventListener()**, ou en utilisant plus loin dans le code la méthode **removeEventListener()**.
- Il est possible d'ajouter des instructions de debug pour d'autres types d'événements (par exemple "onLogout") en ajoutant simplement d'autres écouteurs pour cet événement en particulier.
- Il est possible d'ajouter d'autres fonctionnalités (exemple envoi de mail etc...) sans changer ou alourdir la logique actuelle de l'application.

V - EventListener global

Dans quelques cas rares (spécialement dans des situations de tests ou de debug), il peut être pratique d'ajouter un écouteur (EventListener) qui attrapera tous les événements de l'application.

Pour faciliter l'abonnement de cet écouteur à tous les types d'événements diffusés par un **EventDispatcher**, il ne sera pas pratique d'associer un à un tous les types d'événements que le diffuseur peut propager. Pour simplifier cet enregistrement **global** des événements d'un diffuseur d'événements, il sera pratique d'utiliser la méthode **addGlobalEventListener()** comme ceci :

```
import vegas.events.EventDispatcher ;
import vegas.events.EventListener ;

import test.events.DebugHandler ;

var disp:EventDispatcher = new EventDispatcher() ;

var global:EventListener = new DebugHandler("global") ;

// register the global event listener.
disp.addGlobalEventListener( global ) ;

disp.dispatchEvent( "onLogin"  ) ;
disp.dispatchEvent( "onLogout" ) ;


// unregister the global event listener.
disp.removeGlobalEventListener(global) ;

disp.dispatchEvent( "onLogin"  ) ;
disp.dispatchEvent( "onLogout" ) ;
```

L'écouteur sera invoqué pour tous les types d'événements du diffuseur, dans l'exemple ci-dessus pour les événements de type "onLogin" et "onLogout".

A noter que de façon générale, tous les événements globaux sont diffusés après les événements appartenant au processus local de diffusion des événements vers les écouteurs de bases abonnés au diffuseur avec la méthode **addEventListener**.

La méthode **removeGlobalEventListener()** permet de désabonner un écouteur global.

A noter qu'il est aussi possible dans **VEGAS** d'appliquer un écouteur global avec la méthode **addEventListener** en utilisant le type d'événement "spécial" : **ALL**.

```
import vegas.events.EventDispatcher ;
import vegas.events.EventListener ;

import test.events.DebugHandler ;

var disp:EventDispatcher = new EventDispatcher() ;

var global:EventListener = new DebugHandler() ;

// register the global event listener.
disp.addEventListener( "ALL", global ) ;

disp.dispatchEvent( "onLogin"  ) ;
disp.dispatchEvent( "onLogout" ) ;
```

```
// unregister the global event listener.
disp.removeEventListener( "ALL", global ) ;

disp.dispatchEvent( "onLogin"  ) ;
disp.dispatchEvent( "onLogout" ) ;
```

*Remarque : Dans cet exemple, j'utilise la classe **DebugHandler** définie plus haut dans ce tutorial.*

VI - Multi dispatcher et singletons

Il est possible de créer dans vos applications des instances de la classe **EventDispatcher** mais il est aussi possible de créer et d'utiliser des "références uniques" (ou singletons) en utilisant la méthode statique **getInstance()** de la classe **EventDispatcher**.

Cette approche permet de créer dans vos applications des références globales pour diffuser des événements dans vos applications. Il est possible ainsi d'améliorer la gestion mémoire des applications en utilisant des diffuseurs uniques pour diverses opérations globales.

La méthode **getInstance()** accepte un paramètre "**name**" pour définir quelle référence unique vous voulez utiliser dans votre application. Ce paramètre sert d'identifiant unique pour vos diffuseurs et vous permet de retrouver à tout moment dans votre code votre référence sans avoir de dépendance avec la classe ou l'objet qui doit diffuser des événements dans votre application.

```
import vegas.events.EventDispatcher ;

var default:EventDispatcher = EventDispatcher.getInstance() ; // no value in argument

var user:EventDispatcher  = EventDispatcher.getInstance("user") ;

var debug:EventDispatcher = EventDispatcher.getInstance("debug") ;
```

Le **nom** du singleton de type **EventDispatcher** représente une sorte de canal qui permet de dispatcher les événements de votre application un peu comme on pourrait définir une fréquence spécifique pour émettre des émissions de "radio" via un flux FM.

Il est possible de retrouver sur une référence de type **EventDispatcher** son nom en utilisant la propriété virtuelle en lecture seule **name** ou la méthode **getName()**.

```
var user:EventDispatcher  = EventDispatcher.getInstance("user") ;
trace( "The name of the user dispatcher : " + user.getName() ) ;
```

Pour ajouter des écouteurs vous devez juste définir quel diffuseur global doit prendre la responsabilité sur l'écouteur pour un certain type d'événements.

Par exemple, dans une web application, il est possible de créer un diffuseur d'événements pour chaque session de votre application.

Cette fonctionnalité est très pratique pour implémenter le **Design Pattern FrontController** avec **VEGAS**.

Il est possible de détruire une référence globale de type **EventDispatcher** en utilisant les méthodes statiques **removeInstance()** ou **release()** de la classe.

```
import vegas.events.EventDispatcher ;
import vegas.events.EventListener ;

var logger:EventDispatcher  = EventDispatcher.getInstance("logger") ;
var user:EventDispatcher  = EventDispatcher.getInstance("user") ;

var isRemove:Boolean = EventDispatcher.removeInstance( "user" ) ;
trace( "The 'user' EventDispatcher singleton is removed : " + isRemove ) ;

var logger:EventDispatcher = EventDispatcher.release("logger") ;
trace( "The 'logger' EventDispatcher is released : " + logger ) ;
```

La méthode statique **removeInstance()** renvoie true si l'opération de destruction réussit.

La méthode statique **release()** supprime la référence unique (singleton) dans la **Map** interne statique de la classe **EventDispatcher** et retourne cette référence en cas de besoin pour continuer à l'utiliser hors de son contexte d'origine.

Si vous voulez supprimer toutes les références uniques de la classe **EventDispatcher**, vous pouvez utiliser la méthode statique **flush()** :

```
import vegas.events.EventDispatcher ;

var default:EventDispatcher = EventDispatcher.getInstance() ;
var user:EventDispatcher  = EventDispatcher.getInstance("user") ;
var debug:EventDispatcher = EventDispatcher.getInstance("debug") ;

EventDispatcher.flush() ; // Removes all singletons of the EventDispatcher class
```

VII - Fin de la partie 1

Je vais arrêter ici cette première partie, dans la suite de ce tutoriel je vais vous décrire les points suivants :

- Événement cancellable.
- Supprimer un écouteur
- Priorité des EventListener
- Fonctionnalité Auto-remove
- Event Queue
- La classe Delegate (un EventListener un peu particulier)
- La classe Batch (regrouper plusieurs EventListener ensembles)
- Le design pattern FrontController
- Les flux d'événements : Capturing et bubbling events.

Je vous rappelle que pour le moment il existe une version anglaise de ce tutoriel sur le wiki du projet de VEGAS sur google code : **http://code.google.com/p/vegas/wiki/VegasTutorialsEvents**

Pour toute question sur ce tutoriel, veuillez utiliser le Google Groupe de VEGAS : **http://groups.google.com/group/vegasos**.