6 Flex et les frameworks MVC

Cette étape du tutoriel va nous permettre de mettre en pratique l'architecture Modèle-Vue-Contrôleur dans une application Flex. Pour ce faire, télécharger le Framework **PureMVC** à l'adresse suivante:

```
http://trac.puremvc.org/PureMVC_AS3/wiki/Downloads
```

Il faut ensuite copier le fichier *bin/PureMVC_AS3_{version}.swf* dans le dossier *libs* du projet. Flash Builder référence automatiquement toutes les librairies de ce dossier.

Documentation PureMVC :

```
http://puremvc.org/component/option,com_wrapper/Itemid,175/
```

6.1 Marche à suivre

Les instructions qui suivent permettre de mettre en œuvre une application utilisant le framework PureMVC.

6.1.1 Façade

La première étape consiste à mettre en place une façade concrète pour l'application qui, par convention s'appelle *ApplicationFacade*.

ApplicationFacade.as

```
Package tutorial
{
    import controller.ApplicationStartupCmd;
    import org.puremvc.as3.patterns.facade.Facade;
    import tutorial.view.Application;

    public class ApplicationFacade extends Facade
    {
        // Notification names
        public static const STARTUP:String = "startup";

        // Start the application
        public function startup(app:Application):void
        {
            sendNotification(STARTUP, app);
        }

        protected override function initializeController():void
        {
            super.initializeController();
            registerCommand(STARTUP, ApplicationStartupCmd);
        }

        // Singleton instance
        public static function getInstance():ApplicationFacade
        {
            if (instance == null)
                instance = new ApplicationFacade();
```

```
            return instance as ApplicationFacade;
        }

        // This constructor should be private, use getInstance()
        public function ApplicationFacade()
        {
            super();
        }
    }
}
```

La façade concrète est un singleton. La façade étant le point central des communications entre les différents éléments, c'est ici qu'à lieu la *définition des constantes* pour les noms des notifications. C'est également ici qu'à lieu l'*initialisation du contrôleur* avec un ensemble de commandes qui seront exécutées à la réception de notifications.

6.1.2 Vue principale

Ajouter une application au projet. Ce point d'entrée construit la hiérarchie visuelle, initialise la façade, puis active le mécanisme PureMVC.

View.Application.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               xmlns:components="view.*"
               minWidth="955"
               minHeight="600"
               initialize="onInitialize()">
    <fx:Script>
        <![CDATA[

            import tutorial.ApplicationFacade;

            // Application facade
            private var facade:ApplicationFacade;

            private function onInitialize() : void
            {
                facade = ApplicationFacade.getInstance();
                facade.startup(this);
                facade.sendNotification(ApplicationFacade.LOADUSERS);
            }
        ]]>
    </fx:Script>
</s:Application>
```

On crée l'instance de *ApplicationFacade* afin d'y invoquer la méthode startup en passant l'application en paramètre. A noter que mise à part le bloc principal *Application*, les composants visuels n'auront pas besoin d'intéragir directement avec la façade.

6.1.3 Commandes

Nous allons maintenant créer la commande *ApplicationStartupCmd* dans un package *controller*. Elle permet d'ajouter un proxy fournissant une abstraction du modèle de données ainsi qu'un médiateur prenant en charge un composant visuel.

controller.ApplicationStartupCmd.as

```
package tutorial.controller
{
    import tutorial.model.UsersProxy;

    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.command.SimpleCommand;

    import view.ApplicationMediator;
    import view.Application;

    public class ApplicationStartupCmd extends SimpleCommand
    {
        override public function execute(note:INotification):void
        {
            facade.registerProxy(new UsersProxy());
            facade.registerMediator(new ApplicationMediator(note.getBody()
as Application));
        }
    }
}
```

6.1.4 Proxy

Dans notre cas, le proxy *UserProxy* va charger un fichier XML distant comportant des utilisateurs :

assets/users.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
    <user id="1">
        <name>User 1</name>
        <birthDate>1980-01-01</birthDate>
    </user>
    …
    <user id="8">
        <name>User 8</name>
        <birthDate>1986-03-19</birthDate>
    </user>
</users>
```

Le proxy possède une méthode *load* qui permet de charger le fichier XML. Une fois le chargement du fichier terminé, une notification PureMVC est distribuée.

model/UserProxy.as

```
package tutorial.model
{
    import flash.events.Event;
    import flash.net.URLLoader;
    import flash.net.URLRequest;
```

```
    import mx.controls.Alert;
    import mx.rpc.AsyncToken;
    import mx.rpc.IResponder;
    import mx.rpc.events.FaultEvent;
    import mx.rpc.events.ResultEvent;
    import mx.rpc.http.HTTPService;

    import org.puremvc.as3.interfaces.IProxy;
    import org.puremvc.as3.patterns.proxy.Proxy;

    public class UsersProxy extends Proxy implements IProxy
    {
        public static const NAME:String = "UsersProxy";

        public function UsersProxy()
        {
            super(NAME);
        }

        public function load():void
        {
            var service : HTTPService = new HTTPService();
            service.url = "http://localhost/ria/06_PureMVC/users.xml";
            service.showBusyCursor = true;
            service.resultFormat="xml";
            service.addEventListener(ResultEvent.RESULT, onResult);
            service.addEventListener(FaultEvent.FAULT, onFault);
            service.send();
        }

        private function onResult(event:ResultEvent):void
        {
            var users:XML = new XML(event.result);
            sendNotification(ApplicationFacade.LOADUSERS_COMPLETE, users);
        }

        private function onFault(event:FaultEvent):void
        {
            Alert.show("onFault");
        }
    }
}
```

La méthode *onResult* distribue une notification. Il est donc nécessaire de déclarer une constante pour cette notification *loadusers_complete*. Pour ce faire, ajouter la ligne qui suit dans le fichier *ApplicationFacade.as*

```
public static const LOADUSERS_COMPLETE:String = "loadusers_complete";
```

6.1.5 Vues et médiateurs

Nous allons maintenant nous concentrer sur les vues et médiateurs. Pour ce faire, commençons par créer un fichier *ApplicationMediator.as* dans le package *view* qui nous permettra de prendre en charge la vue de l'application.

view.ApplicationMediator.as

```
package tutorial.view
{
    import org.puremvc.as3.interfaces.IMediator;
    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.mediator.Mediator;

    public class ApplicationMediator extends Mediator implements IMediator
    {
        public static const NAME:String = "ApplicationMediator";

        public function ApplicationMediator(view:Application)
        {
            super(NAME, view);
        }

        override public function listNotificationInterests():Array
        {
            return [ApplicationFacade.LOADUSERS_COMPLETE];
        }

        override public function
handleNotification(note:INotification):void
        {
            switch (note.getName())
            {
                case ApplicationFacade.LOADUSERS_COMPLETE:
                var users:XML = note.getBody() as XML;
                mainView.lstUsers.dataProvider = users.user;
                break;
            }
        }

        protected function get mainView():Application
        {
            return viewComponent as Application;
        }
    }
}
```

Les médiateurs travaillent avec les composants visuels. Ils réagissent aux notifications auxquelles ils se sont « abonnés » par l'intermédiaire de leur méthode *listNotificationInterests.* Les actions à effectuer à la réception d'une notification se font dans la méthode *handleNotification*. Dans le cas de notre application, le médiateur *ApplicationMediator* s'abonne à la notification *loadusers_complete* afin qu'il puisse mettre à jour le composant visuel affichant les utilisateurs.

Pour ce faire, il est encore nécessaire d'ajouter une liste à la vue principale :

view.Application.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               xmlns:components="tutorial.view.*"
               minWidth="955"
               minHeight="600"
               initialize="onInitialize()">
    …

    <mx:List
        id="lstUsers"
        width="400"
        height="300"
        labelField="name"/>

</s:Application>
```

6.1.6 Câblage de tous les points précédents

Créer la commande *LoadUsersCmd* dans le package *controller* qui va s'occuper de charger les données du fichier XML.

controller.LoadUsersCmd

```
package tutorial.controller
{
    import tutorial.model.UsersProxy;

    import org.puremvc.as3.interfaces.ICommand;
    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.command.SimpleCommand;

    public class LoadUsersCmd extends SimpleCommand implements ICommand
    {
        override public function execute(notification:INotification):void
        {
            var usersProxy:UsersProxy;
            usersProxy = facade.retrieveProxy(UsersProxy.NAME) as
UsersProxy;
            usersProxy.load();
        }
    }
}
```

Ajouter la notification *loadusers* dans le fichier *ApplicationFacade.as* et définir la commande associée à la notification :

view.Application.mxml

```
…
public static const LOADUSERS:String = "loadusers";

protected override function initializeController():void
{
    …
    registerCommand(LOADUSERS, LoadUsersCmd);
}
```

Pour terminer, il est encore nécessaire de distribuer la notification *LOADUSERS* au chargement de l'application. Par exemple, ajouter ceci dans la méthode *onInitialize* :

view.Application.mxml

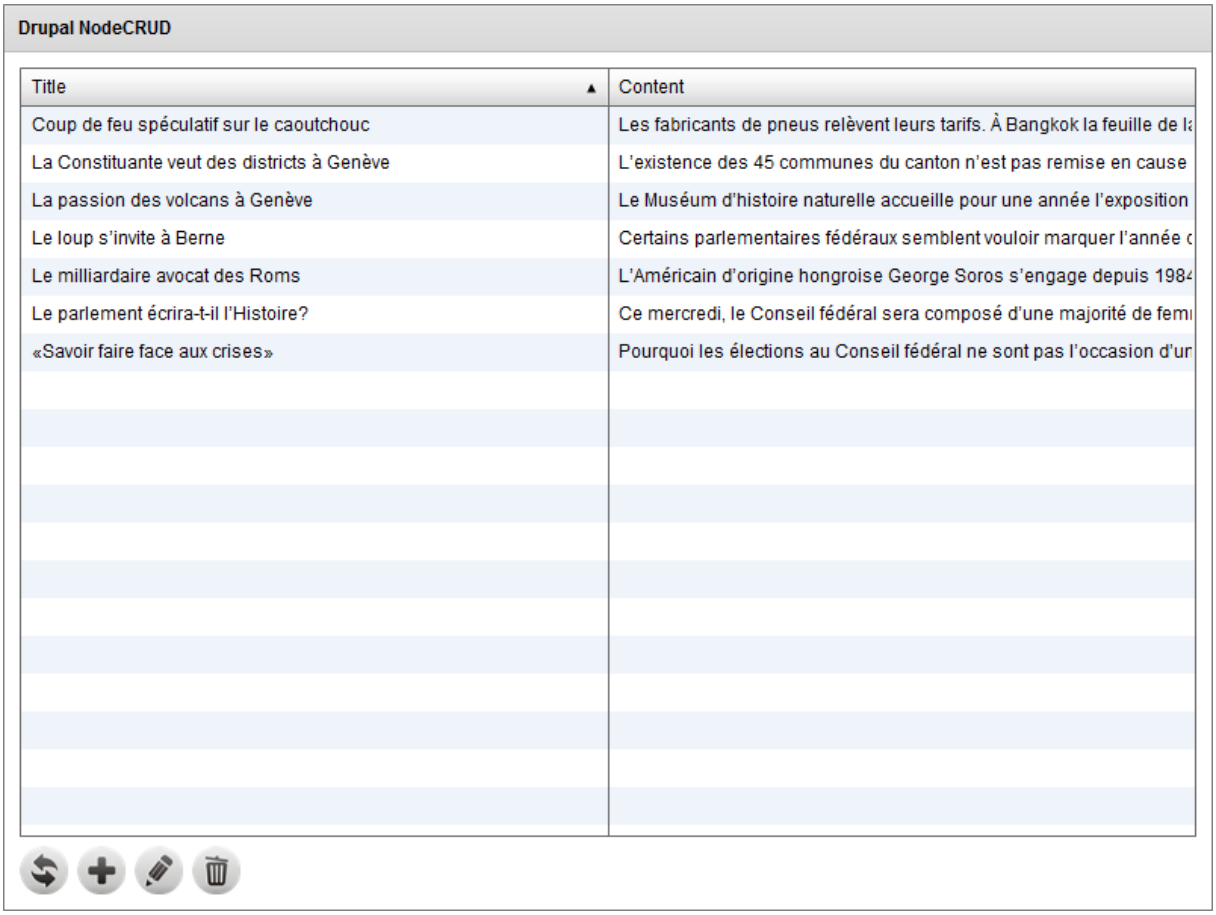
```
private function onInitialize() : void
{
    …
    facade.sendNotification(ApplicationFacade.LOADUSERS);
}
```

6.2 Exercices

Modifier l'exercice *Geonames* du chapitre 3.9 afin de le structurer à la manière PureMVC.

Drupal est un système de gestion de contenu libre et open-source écrit en PHP. Dans cet exercice, nous allons développer une application ria permettant de créer, lire, modifier et supprimer des nœuds du CMS Drupal. Bien entendu, l'application doit mettre en oeuvre le framework PureMVC présenté dans ce chapitre.

Le rendu final de l'application doit être semblable à ceci :



- Pour ce faire, il est possible de se baser sur le fichier **NodesProxy.as** qui va dialoguer avec une installation de Drupal sur le serveur du cours ogo. Le proxy met à disposition les méthodes suivantes :

NodesProxy.all() => Obtenir de la liste de tous les nœuds du système.
Retour en cas de succès : • Envoi de la notification `ApplicationFacade.LOADNODES_SUCCESS` • L'`ArrayCollection` des nœuds est accessible par le getter `nodes()` du proxy
Retour en cas d'échec: • Envoi de la notification `ApplicationFacade.LOADNODES_FAILED`

NodesProxy.create() => Créer un nouveau nœud à partir d'un Object(title, body)
Retour en cas de succès : • Envoi de la notification ApplicationFacade.CREATENODES_SUCCESS avec pour paramètre le nœud créé Object(nid, title, body)
Retour en cas d'échec: • Envoi de la notification `ApplicationFacade.CREATENODES_FAILED`

NodesProxy.update() => Modifier un nœud existant à partir d'un Object(nid, title, body)
Retour en cas de succès : • Envoi de la notification ApplicationFacade.UPDATENODES_SUCCESS avec pour paramètre le nœud modifié Object(nid, title, body)
Retour en cas d'échec: • Envoi de la notification `ApplicationFacade.UPDATENODES_FAILED`

NodesProxy.delete() => Supprimer un nœud Object(nid, title, body)
Retour en cas de succès : • Envoi de la notification ApplicationFacade.DELETENODES_SUCCESS avec pour paramètre le nœud supprimé Object(nid, title, body)
Retour en cas d'échec: • Envoi de la notification `ApplicationFacade.DELETENODES_FAILED`