
Article Adobe

Tutorial Air / PureMVC / SQLite

Tegnane Ly – Shoun Ichida - Vincent Nguyen-Huu

Version 1.0

22 Pages

20/04/2009

[Mots clés]

Propriétés du document

Auteur	Tegnane Ly – Shoun Ichida - Vincent Nguyen-Huu
Version	1.0
Nombre de pages	22
Références	

Historique du document

Date de révision	Version	Auteur	Changements
20/04/2009	1.0	TRIGRAM	

Sommaire

Introduction	4
AIR.....	4
PureMVC.....	4
SQLite.....	5
Création d'un projet	6
Nouveau projet.....	6
Installation de PureMVC.....	6
Mise en place d'une architecture PureMVC.....	8
La Facade	8
Le Contrôleur	10
Le Modèle	12
La Vue	15
Communication SQLite	19
SQLite Manager	19
Mise à jour du Modèle.....	19
Release.....	21
Conclusion.....	22

Introduction

En se promenant sur internet aujourd'hui nous trouvons de plus en plus de contenu dit « riche ». De nombreuses maisons d'édition ont apporté aux sites web des moyens pour communiquer de façon toujours plus dynamique, fluide et élégante. C'est ce que nous retrouvons par exemple avec la technologie SilverLight de **Microsoft** ou Flash d'**Adobe**.

Afin de rester dans la course et de proposer aux développeurs des possibilités de création toujours plus puissantes, Adobe a apporté la technologie **Flex / AIR** et ses produits dérivés. Initialement créé par **Macromedia** en 2004 et racheté par **Adobe** en 2006, **Flex** est une solution performante dans la création et le déploiement des Applications Internet Riches (RIA), on retrouve son équivalent au niveau Desktop avec **AIR**.

Adobe propose, en plus de ces moteurs, un environnement de développement intégré basé sur l'IDE de référence de Java, Eclipse, et nommé **Flex Builder**. Cet outil est très pratique lorsque l'on développe puisqu'il permet une puissante résolution de problèmes et guide le développeur tout au long de ses projets.

Une autre particularité de ces technologies est qu'elles s'appuient sur un langage événementiel, c'est-à-dire que les solutions n'ont pas besoin de se recharger pour changer leur contenu. C'est ce qui fait que les applications sont fluides et dynamiques.

Cet article a pour but de vous guider dans l'élaboration d'un premier projet AIR reposant sur une architecture MVC (Modèle Vue Contrôleur) et proposant une communication à une base de données. Pour ça nous allons encore introduire quelques notions.

AIR

AIR, acronyme d'Adobe Integrated Runtime est une machine virtuelle qui a vu le jour en mars 2008. Il s'agit de la solution d'application riche de bureau (RDA) d'Adobe. Ce moteur permet d'exécuter les applications Flash comme un programme. Plus besoin de navigateur internet pour lancer son application Flash, AIR donne le même rendu en utilisant les performances de la machine hôte et va même jusqu'à proposer l'intégration de certaines bases de données en local comme SQLite.

PureMVC

PureMVC est un Framework permettant de créer des applications basées sur le design pattern MVC (Modèle Vue Contrôleur). Ce framework Open Source a originellement été développé en ActionScript 3 et est utilisé pour les applications Flex, Flash et AIR.

Nous allons à travers ce tutorial apprendre à le mettre en place dans le cadre d'un projet AIR et voir quels en sont les avantages lorsqu'il est utilisé.

SQLite

SQLite est un Système de Gestion de Base de Données (SGBD) qui ne reproduit pas le schéma habituel client/serveur. Cette technologie repose sur une bibliothèque développée en C, qui sauvegarde toutes les données dans des fichiers. Ce moteur de base de données est utilisé dans de nombreux produits grand public et est surtout connu pour sa portabilité. Il est donc normal de retrouver dans la technologie AIR un système permettant de communiquer avec ce SGBD.

Maintenant que nous avons introduit les outils que nous allons utiliser à travers ce tutorial passons au vif du sujet, à savoir le développement d'une petite application AIR couplé à PureMVC, qui teste si un utilisateur est bien stocké dans une base de données SQLite rattachée au projet.

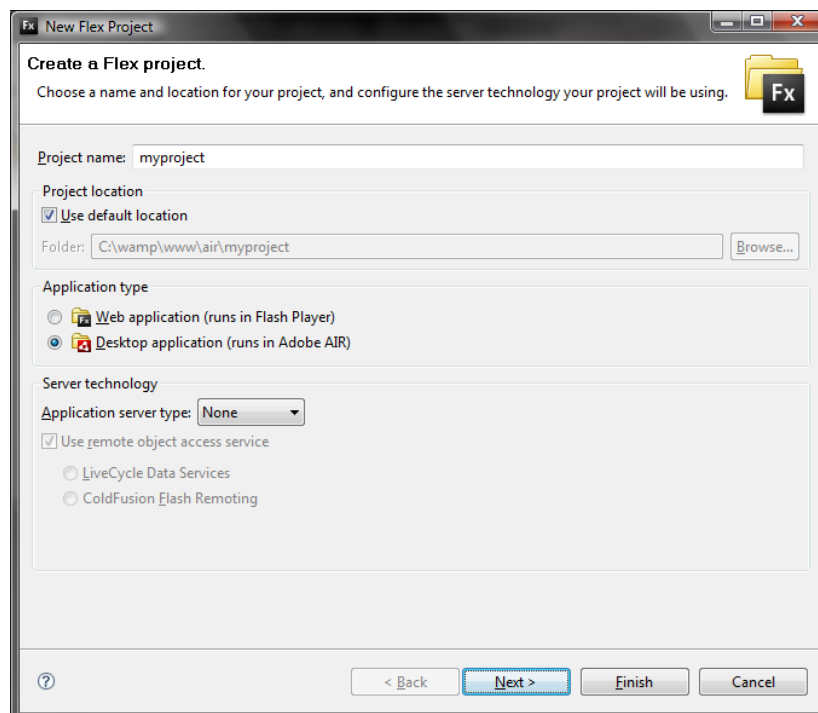
Création d'un projet

La première étape de ce tutorial consiste à créer un nouveau projet **AIR** incorporant tous les éléments nécessaires au bon fonctionnement d'une architecture reposant sur le framework **PureMVC** et une communication **SQLite**.

Commencer, tout d'abord, par vous procurer le **SDK AIR** et l'IDE **Flex Builder**, ces deux outils sont indispensables et disponibles sur le site officiel d'Adobe, à l'adresse suivante : <http://www.adobe.com/products/air/tools/>.

Nouveau projet

Une fois le **SDK AIR** installé, on peut s'attaquer à la création d'un nouveau projet, pour cela lancer votre IDE **Flex Builder**, puis aller dans **File** → **New** → **Flex Project**.

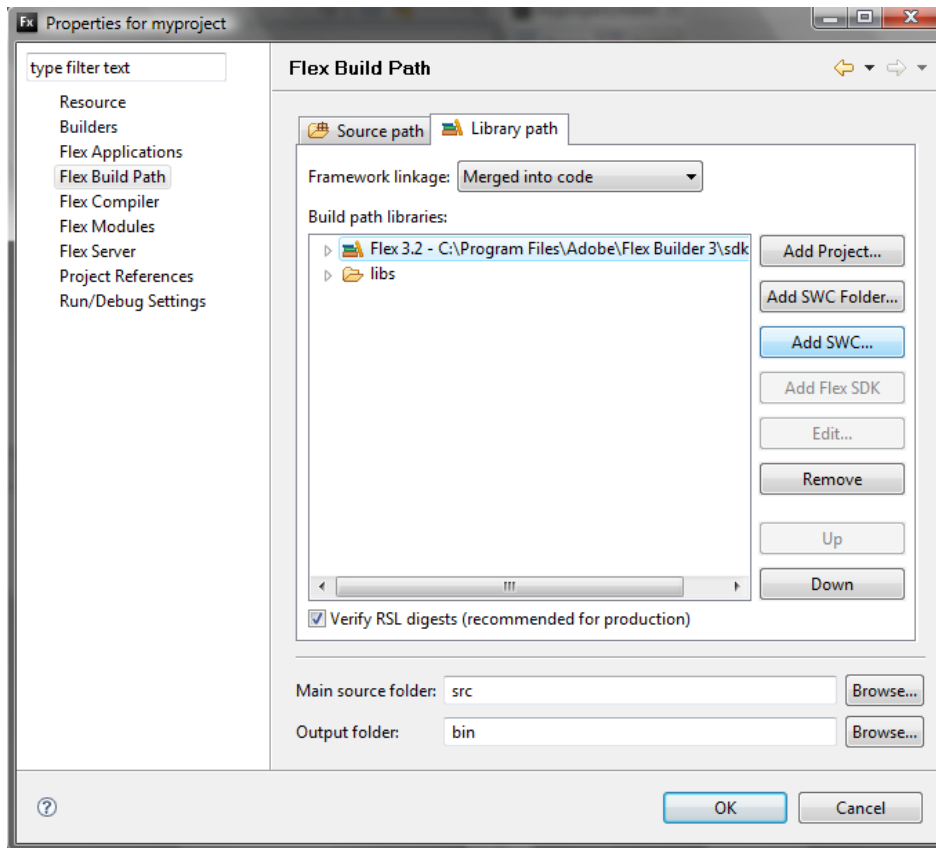


Choisissez un nom de projet, par exemple **myproject**, et le type d'application, **AIR**. Ensuite cliquez sur **Finish** pour créer directement votre projet ou sur **Next** si vous voulez préciser le répertoire de sortie de votre application, le répertoire source, le point d'entrée, ou des bibliothèques externes.

Installation de PureMVC

Maintenant il vous faut télécharger la bibliothèque **PureMVC**, disponible sur le site officiel de **PureMVC** à l'adresse : [http://trac.puremvc.org/PureMVC AS3/wiki/Downloads](http://trac.puremvc.org/PureMVC_AS3/wiki/Downloads).

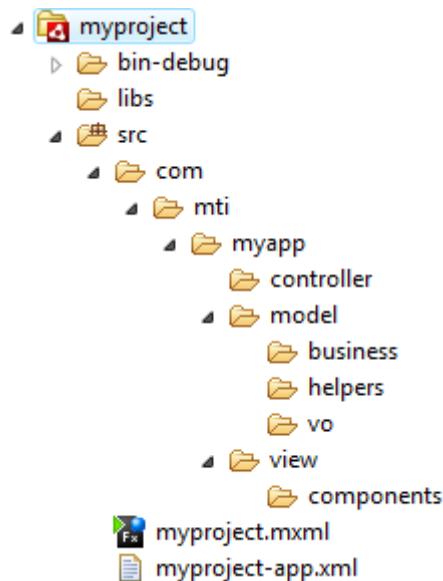
Récupérez l'archive et décompressez là à un endroit de votre choix. Pour inclure concrètement cette bibliothèque au projet, allez dans **Flex Builder**, faites un clic droit sur votre projet, sélectionner **properties**, allez dans la section **Flex Build Path** puis cliquez sur l'onglet **Library Path**.



Cliquez sur **Add SWC** et ajoutez le fichier **.swc** contenu dans l'archive que vous avez décompressé, cliquer sur **OK** et voilà votre bibliothèque est importée et le framework **PureMVC** est installé.

Mise en place d'une architecture PureMVC

Nous allons maintenant construire un exemple d'architecture **PureMVC** pour notre projet **AIR**. Pour cela, commencez par créer une arborescence de répertoires à l'intérieur de votre dossier **src** qui corresponde à l'exemple ci-dessous.



Il s'agit d'une hiérarchie de nom de dossiers typique, ainsi par convention on commencera d'abord par un nom d'extension, ensuite un nom de groupe/entreprise, suivi par un nom d'application et pour finir les noms des répertoires de l'application de type **PureMVC**.

La Facade

Le principal élément et le premier à mettre en place au sein de notre architecture est la classe **ApplicationFacade** qu'on placera à la racine de notre répertoire **myapp** et qui étend de la classe **Facade** du framework **PureMVC**.

Cette classe représentera l'application, contiendra les différentes notifications événementielles du projet et servira de façade pour la communication entre les Contrôleurs (**Commandes**), les Vues (**Médiators**) et les Modèles (**Proxys**) du projet, elle devra contenir au moins 3 méthodes :

- **getInstance** :
Méthode qui permet de récupérer une instance de la Facade (on se servira d'elle comme d'un singleton).
- **startup** :
Méthode qui envoie une notification à propos du démarrage de la Facade, donc de l'application.
- **initializeController** :
Méthode qui surcharge sa méthode parente et qui permet d'enregistrer des Commandes (Contrôleurs) propres à l'application.

ApplicationFacade.as

```

package com.mti.myapp
{
    import com.mti.myapp.controller.ApplicationCommand;
    import com.mti.myapp.controller.LoginCommand;

    import org.puremvc.as3.patterns.facade.Facade;

    public class ApplicationFacade extends Facade
    {
        public static const NAME : String = 'ApplicationFacade';

        /**
         * Notifications values
         */
        public static const STARTUP : String = "Startup";
        public static const LOGIN : String = "Login";

        /**
         * Constructor
         */
        public function ApplicationFacade()
        {
            super();
        }

        /**
         * Get an instance of our application facade
         */
        public static function getInstance() : ApplicationFacade
        {
            if (instance == null)
                instance = new ApplicationFacade();
            return instance as ApplicationFacade;
        }

        /**
         * Initialize the differents controller (register commands/controllers)
         */
        override protected function initializeController() : void
        {
            super.initializeController();
            registerCommand(STARTUP, ApplicationCommand);
            registerCommand(LOGIN, LoginCommand);
        }

        /**
         * Sartup (send notification of startup)
         */
        public function startup(app : myproject) : void
        {
            sendNotification(STARTUP, app);
        }
    }
}

```

Cette **ApplicationFacade** sera instanciée et appelée dans notre fichier **.mxml** principal. Dans l'exemple, le fichier **myproject.mxml** est situé à la racine du projet.

myproject.mxml

```

<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    title="MyProject"

```

```

creationComplete="facade.startup(this);"
xmlns:view="com.mti.myapp.view.components.*"
>

<mx:Script>
<![CDATA[
    import com.mti.myapp.ApplicationFacade;
    private var facade : ApplicationFacade = ApplicationFacade.getInstance();
]]>
</mx:Script>

<mx:ViewStack id="viewStack">
    <!-- Insert a view element which match the component LoginPanel -->
    <view:LoginPanel id="loginPanel" />
</mx:ViewStack>

</mx:WindowedApplication>
    
```

Dès que les composants seront en place la méthode **startup** de l'**ApplicationFacade** sera appelée par l'attribut **creationComplete** de l'élément **WindowedApplication**, ce qui lancera, par le biais de la notification **STARTUP**, le Contrôleur **ApplicationCommand** que nous allons détailler dans la section suivante.

Le Contrôleur

Maintenant que notre **ApplicationFacade** est créée, il s'agit de créer les Contrôleurs/Commandes qu'elle enregistre et qu'on placera dans le répertoire **myapp/controller**, à savoir:

- **ApplicationCommand** :
 Commande principale de l'application, enregistrée par l'**ApplicationFacade** et appelée par la notification **STARTUP**.
 Cette Commande est un peu spéciale car elle implémente l'interface **MacroCommand** de **PureMVC** et sert surtout à initialiser deux sous Commandes rattachées à elle:
 - o **ModelCommand** :
 Commande qui enregistre les Modèles/Proxys liés à l'application (pour ce projet on en aura 2, l'**ApplicationProxy**, Modèle principal de l'application, et le **LoginProxy**, Modèle rattaché aux noms d'utilisateur).
 - o **ViewCommand** :
 Commande qui enregistre les Vues/Médiators liés à l'application (pour ce projet on en aura un, l'**ApplicationMediator** qu'on détaillera dans la section Vue).

- **LoginCommand** :
 Commande associée à la notification **LOGIN** qui permet de vérifier via le Modèle **LoginProxy** si un login utilisateur, récupéré par la vue **LoginMediator** et stocké dans une classe **LoginVO**, est dans la base de données rattachée au projet ou non (ce comportement sera plus détaillé dans les sections suivantes : Modèle et Vue).

ApplicationCommand.as

```
package com.mti.myapp.controller
```

```

{
    import org.puremvc.as3.patterns.command.MacroCommand;

    public class ApplicationCommand extends MacroCommand
    {
        public static const NAME : String = 'ApplicationCommand';

        /**
         * Constructor
         */
        public function ApplicationCommand()
        {
            super();
        }

        /**
         * Initialize all commands/controllers of the application at startup
         */
        override protected function initializeMacroCommand() : void
        {
            addSubCommand(ModelCommand);
            addSubCommand(ViewCommand);
        }
    }
}

```

ModelCommand.as

```

package com.mti.myapp.controller
{
    import com.mti.myapp.model.ApplicationProxy;
    import com.mti.myapp.model.LoginProxy;

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

    public class ModelCommand extends SimpleCommand
    {
        public static const NAME : String = 'SimpleCommand';

        /**
         * Controller
         */
        public function ModelCommand()
        {
            super();
        }

        /**
         * Execute register of model/proxy application elements
         */
        override public function execute(note : INotification) : void
        {
            facade.registerProxy(new ApplicationProxy());
            facade.registerProxy(new LoginProxy());
        }
    }
}

```

ViewCommand.as

```

package com.mti.myapp.controller
{
    import com.mti.myapp.view.ApplicationMediator;
    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.command.SimpleCommand;

    public class ViewCommand extends SimpleCommand
    {

```

```

public static const NAME : String = 'ViewCommand';

/**
 * Constructor
 */
public function ViewCommand()
{
    super();
}

/**
 * Execute register of view/mediator application elements
 */
override public function execute(note : INotification):void
{
    facade.registerMediator(new ApplicationMediator(note.getBody() as myproject));
}
}
    
```

LoginCommand.as

```

package com.mti.myapp.controller
{
    import com.mti.myapp.model.LoginProxy;
    import com.mti.myapp.model.vo.LoginVO;

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

    public class LoginCommand extends SimpleCommand
    {
        public static const NAME : String = 'LoginCommand';

        /**
         * Constructor
         */
        public function LoginCommand()
        {
            super();
        }

        /**
         * Initialize all commands/controllers of the application
         */
        override public function execute(note : INotification) : void
        {
            var loginVO : LoginVO = note.getBody() as LoginVO;
            var loginProxy:LoginProxy = facade.retrieveProxy(LoginProxy.NAME) as LoginProxy;
            loginProxy.check(loginVO);
        }
    }
}
    
```

L'ensemble des Commandes/Contrôleurs est maintenant en place, il ne nous reste donc plus qu'à constituer et expliquer le comportement des Modèles/Proxys et des Vues/Médiators.

Le Modèle

Nous allons donc à présent définir les Modèles/Proxys de l'application, pour cela rendez-vous dans le répertoire **myapp/model**, c'est ici que nous allons placer les classes qui traiteront nos données et qui communiqueront avec notre base de données, de plus pour une meilleure manipulation on pourra également y placer des classes auxiliaires.

Par convention le répertoire suit l'organisation suivante :

- A la racine les **classes Proxy** correspondantes au Modèle de l'application.
- Un répertoire **business** :
Il sert à stocker des classes de type **delegates**, c'est-à-dire qui servent à gérer des interactions avec d'autres services.
- Un répertoire **helpers** :
Il sert à stocker des classes auxiliaires ou **helpers** servant à manipuler les données du Modèle, par exemple des parseurs de XML.
- Un répertoire **vo** :
Il sert à stocker des classes de type **value object**, c'est-à-dire des classes servant justes à stocker des données.
Dans notre cas on aura juste une classe de type value object : **LoginVO** qui sert à stocker un nom d'utilisateur récupéré par la Vue afin d'être transmis au Modèle par le Contrôleur **LoginCommand**.

LoginVO.as

```
package com.mti.myapp.model.vo
{
    [Bindable]
    public class LoginVO
    {
        public var login:String;

        public function LoginVO()
        {
        }
    }
}
```

Maintenant occupons nous concrètement des classes Proxy du Modèle enregistré par la Commande **ModelCommand** :

- **ApplicationProxy** :
Classe principale pour le Modèle de l'application, dans le cas de notre projet elle ne sert à rien car l'application est trop basique et que les traitements sont directement fait dans la classe **LoginProxy**, toutefois sur une application plus grande elle pourrait être utile.
- **LoginProxy** :
Classe servant à vérifier si le nom d'utilisateur contenu dans un objet **LoginVO** fait déjà partie de la base de données rattachée au projet ou pas, en fonction du résultat elle enverra soit une notification **LOGIN_SUCCESS**, soit une notification **LOGIN_FAILED** (ces notifications devront être rattrapées et interprétées dans une Vue/Mediator afin d'afficher un message, cela sera démontré dans la section Vue).

ApplicationProxy.as

```
package com.mti.myapp.model
{
    import org.puremvc.as3.interfaces.IProxy;
```

```

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

public class ApplicationProxy extends Proxy implements IProxy
{
    public static const NAME : String = 'ApplicationProxy';

    /**
     * Constructor
     * Register models/proxys of the application
     */
    public function ApplicationProxy(data : Object = null)
    {
        super(NAME, data);
    }
}

```

LoginProxy.as

```

package com.mti.myapp.model
{

    import com.mti.myapp.model.vo.LoginVO;

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

    public class LoginProxy extends Proxy implements IProxy
    {
        public static const NAME : String = 'LoginProxy';

        /**
         * Notifications values
         */
        public static const LOGIN_SUCCESS : String = "LoginSuccess";
        public static const LOGIN_FAILED : String = "LoginFailed";

        /**
         * Constructor
         * We create the table logins and insert a login by default to test
         */
        public function LoginProxy(data:Object=null)
        {
            super(NAME, new LoginVO());
            // TODO: Initialize the database just to show an example
        }

        /**
         * Check if a login is present in the database
         */
        public function check(loginVO:LoginVO):void
        {
            // TODO: Add a communication with the database
            // to truly check if the login is in it or not.
            // For the moment we send success only if login == admin
            if (loginVO.login == "admin")
                sendNotification(LoginProxy.LOGIN_SUCCESS);
            else
                sendNotification(LoginProxy.LOGIN_FAILED);
        }
    }
}

```

La communication entre une application **AIR** et une base de données **SQLite** étant un peu complexe nous reviendrons sur la classe **LoginProxy** et détaillerons le vrai processus à suivre dans ses méthodes

sur la dernière partie du tutorial « Communication SQLite ». Pour le moment on se contentera de vérifier si le nom d'utilisateur est égal ou non à « admin » et on renverra la notification adaptée en conséquence.

La Vue

Passons maintenant à la dernière couche applicative de votre projet : les Vues, elles sont placées dans le répertoire **myapp/view** et correspondent aux classes Médiateurs.

Ces classes correspondent aux traitements faits au niveau de l'interface utilisateur et pour avoir un rendu visuel concret à utiliser dans notre fichier **myproject.mxml** on pourra associer à chacune d'elle un Composant graphique situé dans un fichier **.mxml** du sous répertoire **myapp/view/components**.

Le rôle des Médiateurs est de servir de relai entre les Composants **.mxml** et les classes Commandes définies précédemment. Dans notre cas on aura un Médiateur principal l'**ApplicationMediator**, déjà enregistré par le Contrôleur **ViewCommand** et qui va servir à enregistrer les autres Médiateurs de l'application tout en leur associant des Composants **.mxml**, utilisés dans notre fichier graphique principal **myproject.mxml**.

Dans notre exemple l'**ApplicationMediator** devra donc enregistrer le **LoginPanelMediator** et le relier au un Composant **LoginPanel.mxml**.

ApplicationMediator.as

```
package com.mti.myapp.view
{
    import org.puremvc.as3.interfaces.IMediator;
    import org.puremvc.as3.patterns.mediator.Mediator;

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

        /**
         * Constructor
         * Register views/mediators of the application
         * and bind them with the id of a component used in our main file .mxml
         */
        public function ApplicationMediator(viewComponent : myproject)
        {
            super(NAME, viewComponent);
            facade.registerMediator(new LoginPanelMediator(application.loginPanel));
        }

        /**
         * Get the application
         */
        private function get application(): myproject
        {
            return viewComponent as myproject;
        }
    }
}
```

Au niveau de la communication entre un Médiateur et son Composant, le Médiateur récupère juste les évènements créés par le Composant lorsqu'une action est effectuée, et lance une action associée

(envoi d'une notification par exemple), mais il peut également modifier les éléments du Composant suite à la récupération de notifications.

Dans le cadre de notre exemple, le Composant **LoginPanel.mxml** est un formulaire qui envoie un événement au Médiateur **LoginPanelMediator** lorsqu'il est soumis, ce-dernier récupère les valeurs du formulaire, à savoir le nom d'un utilisateur, stocke le tout dans un objet **LoginVO**, et envoie une notification au Contrôleur **LoginCommand** pour qu'il soumette les infos au Modèle **LoginProxy** comme vu juste avant.

Une fois que le **LoginProxy** a envoyé la notification résultant de la vérification du nom d'utilisateur, le **LoginPanelMediator** récupère la notification et modifie le Composant **LoginPanel.mxml** pour qu'il affiche un message de validation ou d'erreur.

LoginPanel.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel
  xmlns:mx="http://www.adobe.com/2006/mxml"
  horizontalAlign="center"
>

  <mx:Script>
  <![CDATA[
    public static const LOGIN_ENTER : String = "LoginEnter";

    private function loginUser(event : MouseEvent = null) : void
    {
      this.dispatchEvent(new Event(LOGIN_ENTER));
    }
  ]]>
</mx:Script>

  <mx:Label text="Test if a login exist in your database"
    horizontalCenter="0" verticalCenter="0"/>

  <mx:Form id="loginForm" color="#000000">
    <mx:FormItem label="Login: ">
      <mx:TextInput id="login" />
    </mx:FormItem>
  </mx:Form>

  <mx:Text
    id="loginMessage"
    textAlign="center"
    fontWeight="bold"
    color="#000000"
    text="[admin is always valid]"
  />

  <mx:ControlBar horizontalAlign="right">
    <mx:Button
      id="loginButton"
      label="Test"
      click="loginUser(event);"
    />
  </mx:ControlBar>

</mx:Panel>
```

LoginPanelMediator.as

```
package com.mti.myapp.view
{
  import com.mti.myapp.ApplicationFacade;
  import com.mti.myapp.model.LoginProxy;
```

```

import com.mti.myapp.model.vo.LoginVO;
import com.mti.myapp.view.components.LoginPanel;
import flash.events.Event;
import org.puremvc.as3.interfaces.IMediator;
import org.puremvc.as3.interfaces.INotification;
import org.puremvc.as3.patterns.mediator.Mediator;

public class LoginPanelMediator extends Mediator implements IMediator
{
    public static const NAME : String = 'LoginPanelMediator';

    /**
     * Constructor
     * Listen event of the component to send notification
     */
    public function LoginPanelMediator(viewComponent : LoginPanel)
    {
        super(NAME, viewComponent);

        loginPanel.addEventListener(LoginPanel.LOGIN_ENTER, loginNotification);
    }

    /**
     * Send a login notification with a loginVO loaded with the values entered
     */
    private function loginNotification(event : Event=null):void
    {
        var loginVO:LoginVO = new LoginVO();
        loginVO.login = loginPanel.login.text;
        sendNotification(ApplicationFacade.LOGIN, loginVO);
    }

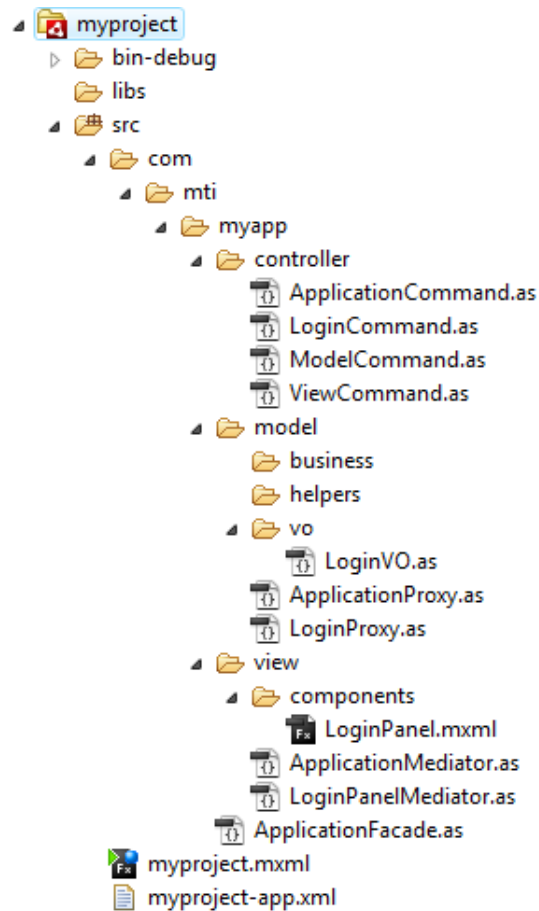
    /**
     * List of the notifications caught by this mediator
     */
    override public function listNotificationInterests():Array
    {
        return [
            LoginProxy.LOGIN_SUCCESS,
            LoginProxy.LOGIN_FAILED
        ];
    }

    /**
     * Actions done when a notifications is caught
     */
    override public function handleNotification(note : INotification):void
    {
        switch (note.getName())
        {
            case LoginProxy.LOGIN_SUCCESS:
                loginPanel.loginMessage.htmlText =
                    "<font color='#00FF00'>Login present in the database</font>";
                break;
            case LoginProxy.LOGIN_FAILED:
                loginPanel.loginMessage.htmlText =
                    "<font color='#FF0000'>No login found</font>";
                break;
            default:
                break;
        }
    }

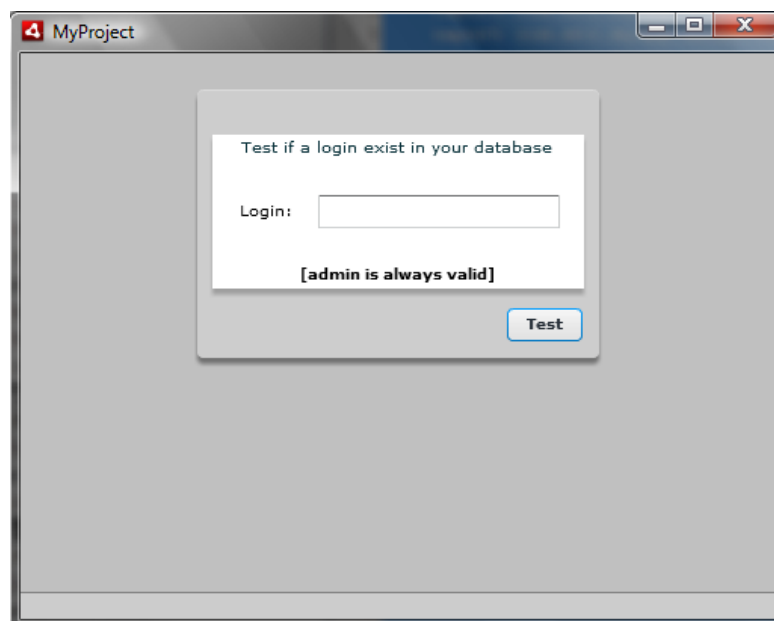
    /**
     * Get the loginPanel component
     */
    private function get loginPanel():LoginPanel
    {
        return viewComponent as LoginPanel;
    }
}

```

Au final, nous obtenons donc un programme **AIR** fonctionnant sur l'architecture du framework **PureMVC** et qui suit l'arborescence de fichiers suivante :



Le programme une fois lancé (**Run** dans **Flex Builder**)



Le programme teste actuellement les noms d'utilisateurs en « dur » et non via une communication par base de données, voilà pourquoi dans la section suivante nous allons détailler comment améliorer le fichier **LoginProxy.as** en le faisant interagir avec une base de données **SQLite**.

Communication SQLite

A présent que l'architecture **PureMVC** et **AIR** n'ont plus de secret pour vous nous allons améliorer la classe **LoginProxy.as** du répertoire **myapp/model** pour qu'elle s'interface avec une base de données **SQLite**.

SQLite Manager

Tout d'abord il faut savoir que la communication entre **AIR** et **SQLite** n'est pas très évidente et un peu fastidieuse à utiliser. Pour couper court à toute forme de problèmes, nous vous invitons à profiter pleinement d'une petite bibliothèque de classes développée par **Elad Elrom** et disponible à l'adresse suivante : <http://elromdesign.com/blog/2009/01/18/adobe-air-sqlite-manager-help-you-handle-your-database-easily/>

Ce **SQLite Manager** est très simple à utiliser et à installer, télécharger simplement l'archive disponible à l'adresse donnée et copier le répertoire **elad** situé dans le dossier **src/com** de l'archive décompressée vers le répertoire **src/com** de votre application.

Voilà, à présent ces classes vont vous permettre de gérer une connexion **SQLite** assez simplement en utilisant une connexion vers une base de données comme un singleton et en lui passant des requêtes (libre à vous plus tard de développer votre propre **SQLite Manager** pour vos projets).

Mise à jour du Modèle

Nous pouvons désormais entamer la dernière étape de ce tutorial. Pour cela, nous allons mettre à jour notre classe **LoginProxy.as** en lui ajoutant un attribut **database** de type **SQLiteManager** qui nous servira à initialiser notre base de données dans le constructeur **LoginProxy** et à tester si un nom d'utilisateur est déjà présent dans la base de données dans la méthode **check**.

LoginProxy.as

```
package com.mti.myapp.model
{
    import com.elad.framework.sqlite.SQLiteManager;
    import com.elad.framework.sqlite.events.StatementSuccessEvent;
    import com.mti.myapp.model.vo.LoginVO;
    import flash.events.Event;
    import mx.collections.ArrayCollection;
    import org.puremvc.as3.interfaces.IProxy;
    import org.puremvc.as3.patterns.proxy.Proxy;

    public class LoginProxy extends Proxy implements IProxy
    {
        public static const NAME : String = 'LoginProxy';

        /**
         * Notifications values
         */
        public static const LOGIN_SUCCESS : String = "LoginSucces";
        public static const LOGIN_FAILED : String = "LoginFailed";

        /**
         * Database Connexion Manager
         */
    }
}
```

```
private var database : SQLiteManager = SQLiteManager.getInstance();
private var loginCollection : ArrayCollection = new ArrayCollection();

/**
 * Constructor
 * We create the table logins and insert a login by default to test
 */
public function LoginProxy(data : Object = null)
{
    super(NAME, new LoginVO());

    var sqlCreate:String = "CREATE TABLE Logins(Login VARCHAR(150) PRIMARY KEY)";

    database.start("myproject.db", "Logins", sqlCreate);
    database.addEventListener(SQLiteManager.COMMAND_EXEC_SUCCESSFULLY,
        onSelectResult);
    database.addEventListener(SQLiteManager.COMMAND_EXEC_FAILED, onFail);
    database.executeSelectAllCommand();

    var loginTest : String = "admin";
    var len : int = loginCollection.length;
    var found:Boolean = false
    for (var i : int; i < len; i++)
    {
        if (loginCollection[i] == loginTest)
            found = true;
    }

    if (!found)
    {
        var sqlInsert : String = "INSERT INTO Logins VALUES('"
            + loginTest + "')";
        database.executeCustomCommand(sqlInsert);
    }
}

/**
 * Check if a login is present in the database
 */
public function check(loginVO : LoginVO):void
{
    loginCollection.removeAll();

    var sqlCreate : String = "CREATE TABLE Logins(Login VARCHAR(150) PRIMARY KEY)";

    database.start("myproject.db", "Logins", sqlCreate);
    database.addEventListener(SQLiteManager.COMMAND_EXEC_SUCCESSFULLY,
        onSelectResult);
    database.addEventListener(SQLiteManager.COMMAND_EXEC_FAILED, onFail);
    database.executeSelectAllCommand();

    var len : int = loginCollection.length;
    var found : Boolean = false
    for (var i : int; i < len; i++)
    {
        if (loginCollection[i] == loginVO.login)
            found = true;
    }

    if (found)
        sendNotification(LoginProxy.LOGIN_SUCCESS);
    else
        sendNotification(LoginProxy.LOGIN_FAILED);
}

/**
 * On a database request successfull
 */
private function onSelectResult(event : StatementSuccessEvent) : void
{
    var result : Array = event.results.data;
    var rowsAffected : int = event.results.rowsAffected;

    if (rowsAffected == 1)
```

```
        database.executeSelectAllCommand();

        if (result == null)
            return;

        var len : int = result.length;
        for (var i : int; i < len; i++)
        {
            loginCollection.addItem(result[i].Login);
        }
    }

    /**
     * On a database request failed
     */
    private function onFail(event : Event) : void
    {
        trace("fail!");
    }
}
```

Comme vous pouvez le constater la connexion à la base se fait de manière très simple, en particulier grâce à la méthode **start** qui prend en arguments un nom de base de données, un nom de table et une requête à effectuer.

Voilà à présent la vérification du nom d'utilisateur ne se fait plus en dur mais directement par rapport à une base de données **SQLite** qu'on initialise directement dans l'exemple si elle ne l'est pas déjà.

Release

Votre application **AIR** est désormais terminée, vous pouvez tester la version **release** de votre projet si vous le souhaitez en allant sur **Project → Export Release Build** de **Flex Builder**. Ceci générera un fichier **.air** que vous pourrez directement installer sur votre machine si vous le désirez.

Conclusion

Félicitations vous êtes arrivés à la fin de ce tutorial et vous disposez désormais d'un exemple de petit projet **AIR** avec une architecture basée sur le framework **PureMVC** et une communication **SQLite**.

Si vous désirez approfondir vos connaissances ou avoir plus d'informations sur les thèmes et technologies abordées durant ce tutorial, nous vous proposons de visiter les liens suivants :

Adobe AIR :

<http://www.adobe.com/products/air/>

PureMVC :

http://trac.puremvc.org/PureMVC_AS3/

SQLiteManager :

<http://elromdesign.com/blog/2009/01/18/adobe-air-sqlite-manager-help-you-handle-your-database-easily/>