6 janvier 2011

Tutoriel RIA Flex

HES-SO University of Applied Sciences Western Switzerland

1 Prise en main de l'environnement de développement

La prise en main de l'environnement ***Adobe Flash Builder 4*** débute par la création d'un nouveau projet. Pour ce faire, utiliser le menu ***Fichier → Nouveau → Projet Flex.***

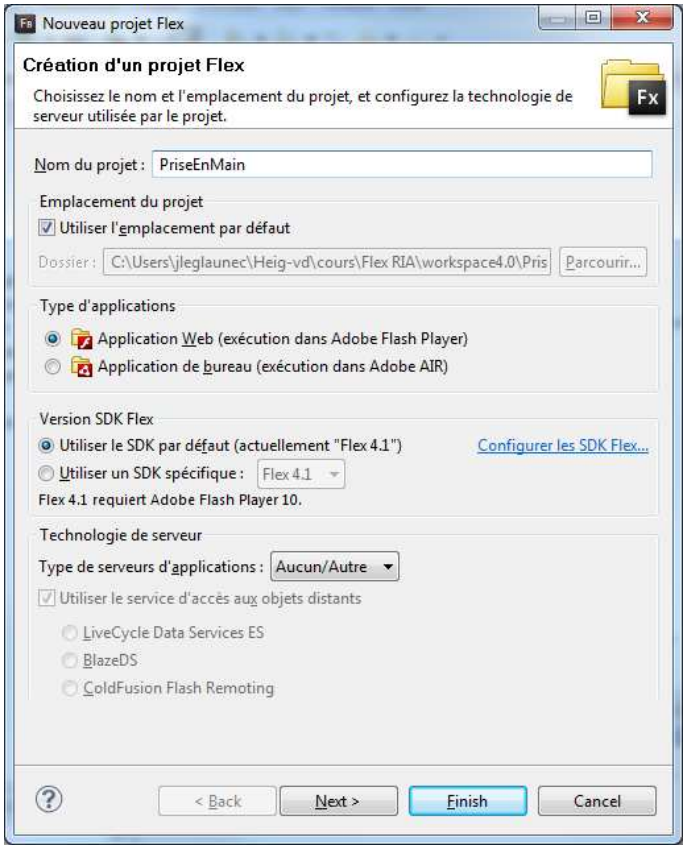
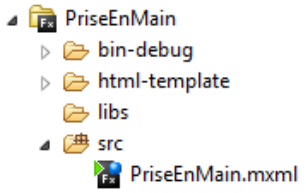


Figure 1 - Création d'un projet

Le projet créé est structuré de la manière suivante



Flash Builder 4 offre la possibilité de spécifier le thème visuel du projet. Ceci peut se faire par l'intermédiaire du menu contextuel ***Properties → Thème Flex.***

1.1 Exemple 1, Application HelloWorld

Créer une application ***HelloWorld*** par l'intermédiaire du menu contextuel **New → Application MXML**.

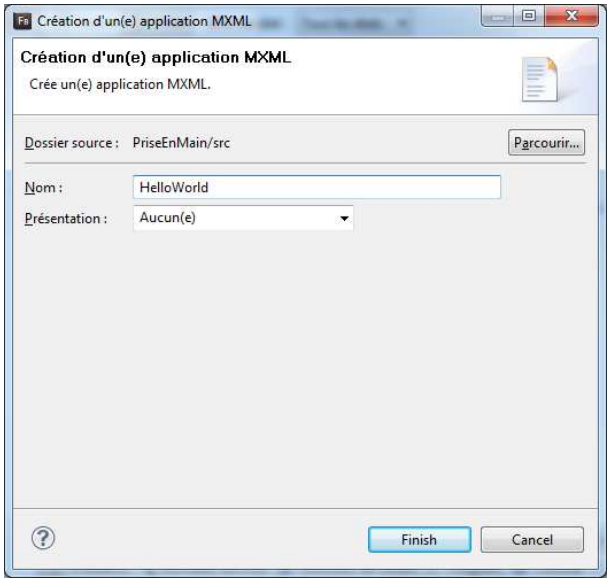


Figure 2 – Création d'une application

Le fichier suivant a été créé dans le dossier ***src***

HelloWorld.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"
               minWidth="955"
               minHeight="600">
    <fx:Declarations></fx:Declarations>
</s:Application>
```

1.1.1 Ajout d'un panel à l'application

La première étape consiste à afficher un panel possédant les caractéristiques suivantes :

- Titre du panel « *Hello world* »
- Layout vertical comportant une marge intérieure de 10px
- Placé au centre de l'application

Panel

```
<s:Panel
    id="pnlHelloWorld"
    title="Hello world"
    verticalCenter="1"
    horizontalCenter="1">
    <s:VGroup
        left="10"
        right="10"
        top="10"
        bottom="10">
    </s:VGroup>
</s:Panel>
```

1.1.2 Ajout d'un label

Le label possède uniquement un identifiant car son texte va être défini lors de l'événement `click` du bouton

Label

```
<s:Label
    id="lblHelloWorld"/>
```

1.1.3 Ajout d'un bouton « Click me »

Ajouter un bouton « *Click me* » qui va se contenter d'afficher « *Hello world !* » dans le label.

Bouton

```
<s:Button
    id="btnClickMe"
    label="Click me"
    click="lblHelloWorld.text = 'Hello world!'"/>
```

1.1.4 Résultat

L'application ***HelloWorld*** produit l'affichage suivant :

Le code source complet de l'application est disponible ci-dessous :

HelloWorld.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"
               minWidth="955"
               minHeight="600">
    <fx:Declarations></fx:Declarations>
    <s:Panel
        id="pnlHelloWorld"
        title="Hello world"
        verticalCenter="1"
        horizontalCenter="1">
        <s:VGroup
            left="10"
            right="10"
            top="10"
            bottom="10">
            <s:Button
                id="btnClickMe"
                label="Click me"
                click="lblHelloWorld.text = 'Hello world!'"/>
            <s:Label
                id="lblHelloWorld"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

1.1.5 Exercice

Toujours dans notre application ***HelloWorld***, afficher le message « *Application chargée* » dans une fenêtre pop-up lorsque l'interface a été entièrement chargée.

Flex Language Reference, Class **Application** :

```
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/spark/
components/Application.html
```

Flex Language Reference, Class **Alert** :

```
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/mx/con
trols/Alert.html?allClasses=1
```

1.2 Exemple 2, Application Perroquet

Le deuxième exemple consiste à créer une application **Perroquet** toujours dans le projet **PriseEnMain**. Cette application a pour but de copier la valeur d'un champ dans un autre. Un bouton est utilisé pour lire la valeur d'un champ texte et l'afficher dans un label.

La méthode suivante permet de copier la valeur d'un champ texte dans un label :

```
private function btnRepeat_click(event : MouseEvent) : void
{
    lblDestination.text = txtSource.text
}
```

Il existe différentes manières d'introduire du code ActionScript dans une application Flex.

1.2.1 Association directe de code ActionScript

```
click="lblHelloWorld.text = 'Hello world!'"
```

Cette méthode peut s'avérer utile lorsque le traitement à effectuer est composé d'une seule commande mais réduit la lisibilité lorsque le code à exécuter est plus conséquent. On l'utilise plus couramment pour appeler une méthode d'un script.

1.2.2 Association avec fonction (fx:Script)

```
<fx:Script>
    <![CDATA[
        import mx.controls.Alert;
        public function sayHelloWorld() : void {
            Alert.show("Hello world!");
        }
    ]]>
</fx:Script>
```

Cette méthode permet de centraliser le code ActionScript dans un fichier MXML.

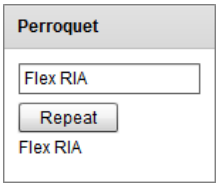
1.2.3 Importation d'un fichier ActionScript externe

```
<fx:Script source="myAsFile.as"/>
```

Cette méthode permet de séparer le code ActionScript du fichier MXML, ceci pour une séparation de la logique et de la présentation tout en permettant la réutilisation du fichier ActionScript.

1.2.4 Résultat

L'application **Perroquet** produit l'affichage suivant :



Le code source complet est disponible ci-dessous :

Perroquet.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"
               minWidth="955"
               minHeight="600">
    <fx:Script>
        <![CDATA[
            private function btnRepeat_click(event : MouseEvent) : void
            {
                lblDestination.text = txtSource.text
            }
        ]]>
    </fx:Script>
    <!-- Panel principal -->
    <s:Panel
        id="pnlPerroquet"
        title="Perroquet"
        verticalCenter="1"
        horizontalCenter="1">
        <s:VGroup
            left="10"
            right="10"
            top="10"
            bottom="10">
            <!-- Champ contenant le texte à copier -->
            <s:TextInput
                id="txtSource"/>
            <!-- Bouton permettant d'effectuer la copie -->
            <s:Button
                id="btnRepeat"
                label="Repeat"
                click="btnRepeat_click(event)"/>
            <!-- Label utilisé dans le but d'afficher le texte copié -->
            <s:Label
                id="lblDestination"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

1.2.5 Exercice

Modifier l'application ***Perroquet*** afin que la copie ne se fasse plus lors du clic sur le bouton mais « *on-the-fly* » à chaque modification du champ texte `txtSource`. De plus, le texte copié sera converti en majuscules.

Flex Language Reference, Class **TextInput** :

```
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/spark/
components/TextInput.html
```

1.3 Exemple 3, Application Login

Le troisième exemple de cette prise en main consiste en une application mettant en œuvre un formulaire d'authentification. Il faut commencer par créer une application **Login** dans le projet **PriseEnMain**. Le but est ici de créer un formulaire de login simple en introduisant les notions d'*états* et de *transitions*.

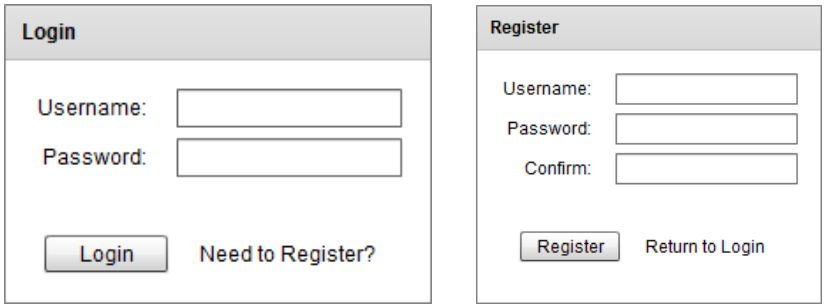
L'utilisation des états permet de dynamiser le comportement des composants visuels. En les couplant avec les transitions, ils permettent de créer des interfaces utilisateurs plus robustes et attrayantes.

Pour plus d'informations à propos des états, se référer à l'adresse suivante :

```
http://help.adobe.com/en_US/flex/using/WS2db454920e96a9e51e63e3d11c0bf690
84-7fb4.html
```

1.3.1 Création d'un formulaire d'authentification

Cette étape consiste à créer un formulaire d'authentification à partir duquel l'utilisateur a la possibilité de s'enregistrer. Les deux actions se font par le même composant visuel en utilisant plusieurs états.



L'état initial du formulaire permet de s'authentifier. Pour permettre à l'utilisateur de s'enregistrer, il est nécessaire de cliquer sur le lien « *Need to Register* » ce qui provoque :

- Le remplacement du titre du panel « Login » par « Register »
- L'ajout d'un champ texte « Confirm »
- Le remplacement du label du bouton « Login » par « Register »
- Le remplacement du lien « Need to Register ? » par « Return to Login »

La création des états avec Flex 4 a été relativement simplifiée par rapport à la version précédente. Dans un premier temps, il est nécessaire de déclarer les différents états du composant visuel :

```
<s:states>
    <s:State name="Login"/>
    <s:State name="Register"/>
</s:states>
```

Il est ensuite possible de déterminer plusieurs fois la même propriété d'un composant visuel. L'exemple suivant met en œuvre un panel qui n'aura pas le même titre selon l'état courant de l'interface:

```
<s:Panel
    id="loginPanel"
    title.Login="Login"
    title.Register="Register"
```

Nous pouvons également déterminer qu'un composant sera ajouté à l'interface uniquement lorsqu'on se trouve dans un état spécifique avec la propriété `includeIn`:

```
<mx:FormItem
    id="confirm"
    label="Confirm:"
    includeIn="Register">
```

1.3.2 Ajout des transitions

Les **transitions** Flex sont utilisées lorsqu'il s'agit de dynamiser une interface. Une transition s'effectue lors du passage d'un état à un autre et s'applique sur un ou plusieurs objet(s).

Transitions

```
<s:transitions>
    <s:Transition fromState="Login">
        <s:Parallel>
            <mx:Resize target="{loginPanel}" duration="100"/>
            <s:Wipe target="{confirm}" direction="right"/>
        </s:Parallel>
    </s:Transition>
    <s:Transition fromState="Register">
        <s:Sequence>
            <mx:Resize target="{loginPanel}" duration="100"/>
        </s:Sequence>
    </s:Transition>
</s:transitions>
```

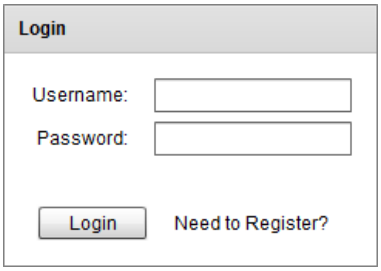
Il est possible de personnaliser les transitions en y ajoutant un certain nombre d'effets comme par exemple Blur, Move, Fade, Dissolve, Glow, Resize, Rotate, Zoom, etc.

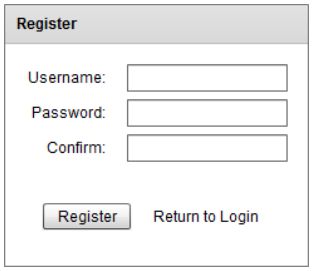
Pour plus d'informations à propos des effets et transitions, se référer aux adresses suivantes :

```
http://www.flex-tutorial.fr/2009/03/06/flex-4-les-nouveaux-effets-flex-
de-gumbo/

http://help.adobe.com/en_US/flex/using/WS4809A78C-9738-465d-B875-
B0049C9B0ED4.html
```

1.3.3 Résultat

L'application **Login** produit les affichages suivants :





Le code source complet est disponible ci-dessous :

Login.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"
               minWidth="955"
               minHeight="600">
    <!-- Définition des états -->
    <s:states>
        <s:State name="Login"/>
        <s:State name="Register"/>
    </s:states>
    <!-- Définition des transitions -->
    <s:transitions>
        <s:Transition fromState="Login">
            <s:Parallel>
                <mx:Resize target="{loginPanel}" duration="100"/>
                <s:Wipe target="{confirm}" direction="right"/>
            </s:Parallel>
        </s:Transition>
        <s:Transition fromState="Register">
            <s:Sequence>
                <mx:Resize target="{loginPanel}" duration="100"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>
    <!-- Panel principal -->
    <s:Panel
        id="loginPanel"
        title.Login="Login"
        title.Register="Register"
        verticalCenter="1"
        horizontalCenter="1"
        height.Login="170"
        height.Register="210">
        <s:VGroup>
            <!-- Formulaire de login -->
            <mx:Form
                id="loginForm">
                <mx:FormItem
                 label="Username:">
                 <s:TextInput
                        id="txtUsername"/>
                </mx:FormItem>
                <mx:FormItem
                 label="Password:">
                 <s:TextInput
```

```
                    id="txtPassword"
                    displayAsPassword="true"/>
            </mx:FormItem>
            <mx:FormItem
            id="confirm"
            label="Confirm:"
            includeIn="Register">
            <s:TextInput
                    displayAsPassword="true"/>
            </mx:FormItem>
        </mx:Form>
        <!-- Barre de contrôle (Login/Register) -->
        <mx:ControlBar
            id="cbrControls"
            horizontalAlign="right">
            <s:Button
            id="btnLoginRegister"
            label.Login="Login"
            label.Register="Register"/>
            <mx:LinkButton
            id="registerLink"
            label.Login="Need to Register?"
            click.Login="{currentState='Register'}"
            label.Register="Return to Login"
            click.Register="{currentState='Login'}"/>
        </mx:ControlBar>
    </s:VGroup>
</s:Panel>
</s:Application>
```

1.3.4 Exercices

Modifier l'application ***Login*** afin de vérifier lors du clic sur le bouton « *Login* » que le *« Username = root »* et le « *Password = 1234 »*. Afficher un message à l'utilisateur indiquant si l'authentification est réussie.

Flex Language Reference, Class **TextInput** :

```
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/spark/
components/TextInput.html
```

Ajouter un état à l'application ***Login*** qui permet de faire apparaître un label *« Bad Login »* lorsque l'authentification a échoué. De plus, ajouter un effet sur l'apparition du « Bad Login ».

Flex Language Reference, Class **State** :

```
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/mx/sta
tes/State.html
```

1.4 Skinning de composants

Ce chapitre est axé sur la composante visuelle des interfaces *« Rich »*. Le SDK Flex 4 a pour objectif d'établir une séparation entre l'aspect visuel d'un composant et son comportement. Prenons comme exemple la classe `spark.components.Button` : Cette classe contient uniquement la logique liée au comportement d'un bouton. Les aspects visuels quant à eux sont définis dans la classe d'habillage `spark.skins.spark.ButtonSkin`.

1.4.1 Skinning de bouton

Nous allons commencer par créer un habillage de bouton simple (texte placé dans un rectangle arrondi). Pour ce faire, créer un *« composant MXML »* que l'on appellera *« ButtonSkin.mxml ».*

Le composant en question est un **Skin** applicable au composant **Button :**

```
<fx:Metadata>
    [HostComponent("spark.components.Button")]
</fx:Metadata>
```

Il doit obligatoirement implémenter les 4 états suivants :

```
<s:states>
    <s:State name="up" />
    <s:State name="over" />
    <s:State name="down" />
    <s:State name="disabled" />
</s:states>
```

La documentation définit les états à implémenter :

```
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/spark/
components/Button.html#SkinStateSummary
```

Nous allons maintenant utiliser la primitive `spark.primitives.Rect` afin de dessiner un rectangle. Un dégradé est utilisé pour la couleur de fond. On peut remarquer que le gradient est modifié en fonction de l'état du bouton.

```
<s:Rect id="r1" radiusX="4" radiusY="4" width="100%" height="100%">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry
                id="fillColorTop"
                color="0x393939"
                color.over="0x131313"
                color.down="0x000000"
                ratio="0"
                alpha="1"/>
```

```
            <s:GradientEntry
                id="fillColorBottom"
                color="0x131313"
                color.over="0x393939"
                color.down="0x000000"
                ratio="1"
                alpha="1"/>
        </s:LinearGradient>
    </s:fill>
</s:Rect>
```

Il est encore nécessaire d'ajouter un label permettant d'afficher le texte du bouton :

```
<s:Label
    text="{hostComponent.label}"
    color="0xD8D8D8"
    textAlign="center"
    verticalAlign="middle"
    horizontalCenter="0"
    verticalCenter="1" />
```

`hostComponent.label` permet à notre classe d'habillage d'accéder à la valeur de la propriété label du composant Button.

Créer ensuite l'application **UseMyButtonSkin.mxml** afin de mettre en œuvre un bouton utilisant le skin précédemment créé.

UseMyButtonSkin.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">

    <s:VGroup
        verticalCenter="1"
        horizontalCenter="1">
        <!-- Permet d'activer/désactiver le bouton -->
        <s:CheckBox
            id="cbx"
            label="disable Button"/>

        <!-- Bouton personnalisé-->
        <s:Button
            id="btn"
            skinClass="skin.ButtonSkin"
            width="200"
            height="60"
            label="Button"
            enabled="{!cbx.selected}"/>
    </s:VGroup>
</s:Application>
```

`skinClass="skin.ButtonSkin"` permet de spécifier le skin à utiliser pour le bouton en question.

1.4.2 Résultat

L'application produit l'affichage suivant :

Up Over Down Disabled

Le code source complet du fichier **ButtonSkin.mxml** est disponible ci-dessous :

skin.ButtonSkin.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx"
        alpha.disabled=".5">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>
    <s:states>
        <s:State name="up" />
        <s:State name="over" />
        <s:State name="down" />
        <s:State name="disabled" />
    </s:states>
    <s:Rect id="r1" radiusX="4" radiusY="4" width="100%" height="100%">
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry
                id="fillColorTop"
                color.up="0x393939"
                color.over="0x131313"
                color.down="0x000000"
                ratio="0"
                alpha="1"/>
                <s:GradientEntry
                id="fillColorBottom"
                color.up="0x131313"
                color.over="0x393939"
                color.down="0x000000"
                ratio="1"
                alpha="1"/>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>
    <s:Label
        text="{hostComponent.label}"
        color="0xD8D8D8"
        textAlign="center"
        verticalAlign="middle"
        horizontalCenter="0"
        verticalCenter="1" />
</s:Skin>
```

1.4.3 Skinning de cases à cocher

L'objectif est de créer des cases à cocher différentes des traditionnelles CheckBox que l'on rencontre dans la plupart des interfaces. Pour ce faire, nous allons commencer par créer l'application que l'on appellera *« UseMyCheckboxSkin.mxml »*.

UseMyCheckboxSkin.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">
    <s:HGroup gap="25" horizontalCenter="0" verticalCenter="0">
        <s:VGroup gap="5">
            <s:Label text="Récurrence :" paddingBottom="4"
                 fontSize="11" textDecoration="underline" />
            <s:CheckBox label="Lundi" selected="true"
                        skinClass="skin.CheckboxSkin"/>
            <s:CheckBox label="Mardi"
                        skinClass="skin.CheckboxSkin"/>
            <s:CheckBox label="Mercredi" selected="true"
                        skinClass="skin.CheckboxSkin"/>
            <s:CheckBox label="Jeudi" selected="true"
                        skinClass="skin.CheckboxSkin"/>
            <s:CheckBox label="Vendredi"
                        skinClass="skin.CheckboxSkin"/>
            <s:CheckBox label="Samedi"
                        skinClass="skin.CheckboxSkin"/>
            <s:CheckBox label="Dimanche"
                        skinClass="skin.CheckboxSkin"/>
        </s:VGroup>
    </s:HGroup>
</s:Application>
```

Créons maintenant le skin *« CheckboxSkin.xml »* applicable au composant `spark.components.CheckBox`. Il s'occupe de dessiner les symboles ✓ et ✗. Plusieurs étapes sont nécessaires pour obtenir la case à cocher personnalisée :

- Affichage du symbole à partir d'une collection de segments `spark.primitive.Path`
- Affichage d'une ombre à l'aide du filtre `spark.filters.DropShadowFilter`
- Ajout d'une zone cliquable invisible au dessus du symbole
- Affichage du label associé à la case à cocher

skin.CheckboxSkin.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Metadata>
        [HostComponent("spark.components.CheckBox")]
    </fx:Metadata>

    <s:states>
        <s:State name="up" stateGroups="unchecked" />
        <s:State name="over" stateGroups="unchecked" />
```

```
        <s:State name="down" stateGroups="unchecked" />
        <s:State name="disabled" stateGroups="unchecked" />
        <s:State name="upAndSelected" stateGroups="checked" />
        <s:State name="overAndSelected" stateGroups="checked" />
        <s:State name="downAndSelected" stateGroups="checked" />
        <s:State name="disabledAndSelected" stateGroups="checked" />
    </s:states>

    <s:Group
        id="marks"
        verticalCenter="0"
        width="11"
        height="11"
        left="0">

        <!-- Checked-->
        <s:Path
            id="pthChecked"
            includeIn="checked"
            horizontalCenter="0"
            verticalCenter="0"
            width="11"
            height="11"
            data="M 100 0 C 75.148 24.853 46.191 87.574 46.191 87.574 C
46.191 87.574 14.204 40.716 0 40.716 L 25.11 41.012 L 43.787 62.213 L
79.29 0 L 100 0 Z" >
            <s:fill>
                <s:SolidColor color="0x70d000"/>
            </s:fill>
        </s:Path>

        <!-- Unchecked -->
        <s:Path
            id="pthUnchecked"
            includeIn="unchecked"
            horizontalCenter="0"
            verticalCenter="0"
            width="9"
            height="9"
            data="M 100 90.29 L 60.694 42.28 C 72.2 26.205 84.896 9.838
95.214 0 L 74.28 0.1 L 51.336 30.851 L 26.922 1.031 L 0 1.031 C 13.126
7.917 29.115 24.561 43.297 41.625 L 7.425 89.702 L 28.995 89.617 C 28.995
89.617 39.309 73.04 52.832 53.468 C 68.081 72.987 79.403 90.131 79.403
90.131 L 100 90.29 Z" >
            <s:fill>
                <s:SolidColor color="0xe83800"/>
            </s:fill>
        </s:Path>

        <!-- Zone de clic (invisible) -->
        <s:Group
            width="100%"
            height="100%"
            alpha="0">
            <s:Rect
                width="100%"
                height="100%">
                <s:fill>
                 <s:SolidColor color="#FFFFFF"/>
                </s:fill>
            </s:Rect>
        </s:Group>

        <!-- Ajout d'une ombre -->
        <s:filters>
```

```
            <s:DropShadowFilter
                distance="1"
                strength="0.75"
                blurX="1"
                blurY="1" />
        </s:filters>
    </s:Group>

    <!-- Label -->
    <s:Label
        text="{hostComponent.label}"
        textAlign="start"
        color.over="#CCCCCC"
        color.down="#000000"
        color.overAndSelected="#CCCCCC"
        color.downAndSelected="#000000"
        left="16" right="0"
        top="3" bottom="3"
        verticalCenter="2"/>

</s:SparkSkin>
```

1.5 Exercices

Appliquer le « *ButtonSkin* » à l'interface de login créée précédemment.

Modifier ce « *ButtonSkin* » afin d'ajouter un effet « *Fade* » pour toute transition vers l'état « disabled ».

Spark Property effects:

```
http://help.adobe.com/en_US/flex/using/WS6461B3EC-1AED-486d-A4B8-
8EBD1993CE22.html
```

2 Skinning avancé & CSS

Dans ce chapitre, nous allons voir plus en détail l'architecture de skinning offerte par Flex 4. Comme nous avons pu le constater précédemment, le framework fourni une séparation claire entre les éléments « logiques » et « visuels » d'un composant. Dès maintenant, nous n'allons plus seulement nous contenter de créer des Skin pour des composants existants, mais également pour des composants personnalisés.

2.1 Liaisons « Component/Skin »

Il y a plusieurs types de liaisons entre le composant et sa vue :

	Définit par le composant	***Définit par le skin***
Data	`[Bindable]` `public var title:String`	`text="{hostComponent.title}"`
Parts	`[SkinPart]` `public var upButton:Button`	`<s:Button id="upButton"/>`
States	`[SkinStates("up")]`	`<s:State name="up"/>`

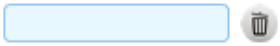
Data : Ce type de liaison permet à la classe d'habillage d'accéder à des propriétés publiques du composant hôte.

Parts : Les skins possèdent également un ensemble de « SkinParts » qui aident à définir le composant. Ces « SkinParts » sont définis par le **skin** mais contrôlés par le **composant**.

States : Les composants possèdent un ensemble d'états qui permettent de modifier leur apparence. Les états sont définis par le **composant** mais contrôlés par le **skin**.

2.2 Personnalisation d'un composant

Pour illustrer ceci, nous allons personnaliser un composant TextInput. L'idée est d'obtenir le composant suivant :



Pour ce faire, commençons par créer un composant CTextInput basé sur le composant de base TextInput.

```
package components
{
    import spark.components.TextInput;

    public class CTextInput extends TextInput
    {
        // SkinParts du composant
        [SkinPart(required="true")]
        public var btn:Image;

        // Data
        [Bindable]
        public var icon:String;

        public function CTextInput()
        {
            super();
        }
    }
}
```

L'annotation `[SkinPart(required="true")]` définit un SkinPart qui devra obligatoirement être présent dans le Skin associé. Cette image sera cliquable et c'est la responsabilité du composant de gérer l'action associée. Quant à l'annotation `[Bindable]`, elle déclare une donnée que l'on pourra utiliser dans le Skin.

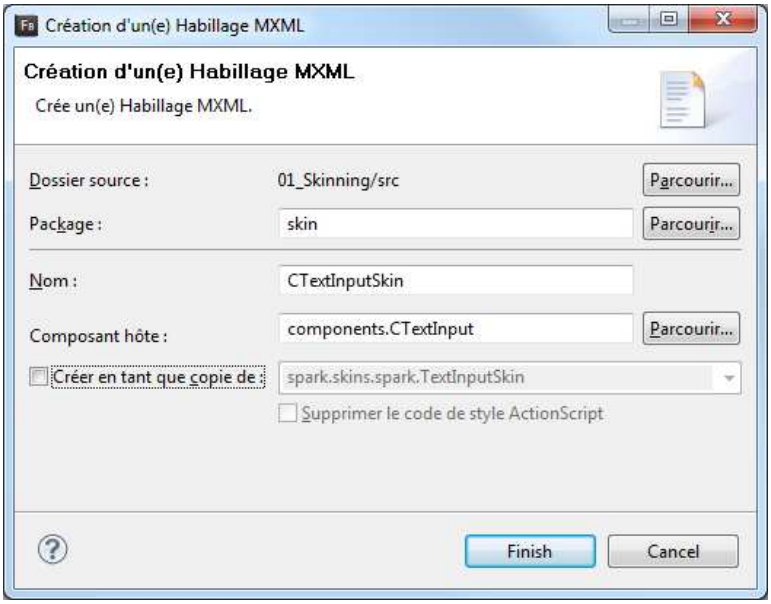
Ensuite, il est nécessaire de traiter le click sur le bouton. Pour ce faire, ajouter les méthodes suivantes :

```
// Ajout les events listeners
override protected function partAdded(partName:String,
                                      instance:Object):void
{
    super.partAdded(partName, instance);
    if (instance == btn)
        btn.addEventListener(MouseEvent.CLICK, btnClick_handler);
}

// Supprime les events listeners
override protected function partRemoved(partName:String,
                                        instance:Object):void
{
    super.partRemoved(partName, instance);
    if (instance == btn)
        btn.removeEventListener(MouseEvent.CLICK, btnClick_handler);
}
```

```
// Handler sur le bouton
private function btnClick_handler(event : MouseEvent):void
{
    this.textDisplay.text = "";
}
```

La méthode `partAdded` est automatiquement appelée pour chacun des SkinParts de notre composant personnalisé. C'est ici que l'on ajoute les écouteurs d'événements qui permettent à l'utilisateur d'interagir avec le skinPart que nous avons ajouté. Dans notre cas, nous nous contentons de réinitialiser le texte du composant.

Nous allons maintenant créer le skin CTextInputSkin.mxml. Pour ce faire, utiliser le menu contextuel ***New → Habillage MXML***.



Le fichier mxml généré comporte:

- La métadonnée `[HostComponent("components.CTextInput")]` déterminant le composant hôte du skin.
- Les états définis par le composant hôte (disabled et normal). Dans notre cas, ces états sont hérités d'un composant de base car nous n'avons pas défini de nouveaux états dans CTextInput.
- Les SkinParts définis par le composant hôte sont btn et textDisplay. Dans notre cas, le skinPart **textDisplay** est hérité d'un composant de base alors que le skinPart **btn** correspond à l'image cliquable définie dans notre CTextInput.

Le composant CTextInput est composé d'un background (Rect), d'une zone de texte (RichEditableText) ainsi que notre SkinPart (Image).

skin.CTextInput

```
<?xml version="1.0" encoding="utf-8"?>
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">
    <!-- host component -->
    <fx:Metadata>
        [HostComponent("components.CTextInput")]
    </fx:Metadata>

    <!-- states -->
    <s:states>
        <s:State name="disabled" />
        <s:State name="normal" />
    </s:states>

    <!-- background -->
    <s:Rect
        left="1" right="27"
        top="1" bottom="1"
        radiusX="3" radiusY="3">
        <s:fill>
            <s:SolidColor color="0xe8f8ff"/>
        </s:fill>
        <s:stroke>
            <s:SolidColorStroke color="0x92c2ef"/>
        </s:stroke>
    </s:Rect>

    <!-- Textcomponent -->
    <s:RichEditableText
        id="textDisplay"
        verticalAlign="middle"
        left="1" right="27"
        top="1" bottom="1"
        paddingLeft="3" paddingTop="5"
        paddingRight="3" paddingBottom="3"/>

    <!-- Image button -->
    <mx:Image
        id="btn"
        source="{hostComponent.icon}"
        right="1"
        top="1" bottom="1"/>

</s:Skin>
```

Créer ensuite l'application **UseMyCTextInput.mxml** afin de mettre en œuvre un champ texte utilisant le skin précédemment créé.

UseMyCTextInput.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="components.*">
    <s:HGroup
        horizontalCenter="0"
        verticalCenter="0">
        <components:CTextInput
            text="Customized"
            width="150"
            icon="@Embed(source='assets/trash.png')"
            skinClass="skin.CTextInputSkin"/>
    </s:HGroup>
</s:Application>
```

2.3 Ajout des styles

Toujours sur la base du même exemple, nous allons introduire les notions de style. En effet, il est possible d'utiliser des CSS pour définir le style des différents composants. Deux choix sont alors possibles : Définir les styles en **local** ou dans un fichier **externe**, mais il faut préciser que dans les deux cas, les styles sont compilés dans l'application.

Dans notre cas, ajouter la balise de style suivante au fichier `UseMyCTextInput.mxml` :

```
<fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    @namespace mx "library://ns.adobe.com/flex/mx";
    @namespace components "components.*";

    components|CTextInputCSS
    {
        skinClass: ClassReference("skin.CTextInputSkinCSS");
        backgroundColor: #E8F8FF;
        borderColor: #92C2EF;
    }

</fx:Style>
```

Le style s'applique à tous les composants CTextInputCSS du fichier. Il permet de définir le skin à appliquer `skinClass: ClassReference("skin.CTextInputSkinCSS")`, ainsi que des couleurs qui seront utilisées dans le skin en question.

Dans le skin, il est ensuite possible d'accéder aux différentes propriétés définies dans la CSS. Par exemple, on utilise la valeur de la propriété **backgroundColor** en tant que couleur de fond de la zone de texte `hostComponent.getStyle('backgroundColor')`.

```
<s:fill>
    <s:SolidColor color="{hostComponent.getStyle('backgroundColor')}"/>
</s:fill>
<s:stroke>
    <s:SolidColorStroke color="{hostComponent.getStyle('borderColor')}"/>
</s:stroke>
```

2.4 Résultat

Le composant personnalisé ressemble à ceci : 

Le clic sur le bouton a pour effet de réinitialiser le contenu du champ texte.

UseMyCTextInputSkinCSS.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="components.*">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        @namespace components "components.*";

        components|CTextInputCSS
        {
            skinClass: ClassReference("skin.CTextInputSkinCSS");
            backgroundColor: #E8F8FF;
            borderColor: #92C2EF;
        }

    </fx:Style>

    <s:HGroup
        horizontalCenter="0"
        verticalCenter="0">
        <components:CTextInputCSS
            icon="@Embed(source='assets/trash.png')"
            text="Customized"
            width="200"/>
    </s:HGroup>

</s:Application>
```

components.CTextInputCSS.as

```
package components
{
    import flash.events.MouseEvent;

    import mx.controls.Image;

    import spark.components.TextInput;

    public class CTextInputCSS extends TextInput
```

```
    {
        // SkinParts du composant
        [SkinPart(required="true")]
        public var btn:Image;

        // Data
        [Bindable]
        public var icon:String;

        public function CTextInputCSS()
        {
            super();
        }

        // Ajout les events listeners
        override protected function partAdded(partName:String,
                                              instance:Object):void
        {
            super.partAdded(partName, instance);
            if (instance == btn)
                btn.addEventListener(MouseEvent.CLICK, btnClick_handler);
        }

        // Supprime les events listeners
        override protected function partRemoved(partName:String,
                                                instance:Object):void
        {
            super.partRemoved(partName, instance);
            if (instance == btn)
                btn.removeEventListener(MouseEvent.CLICK,
                                        btnClick_handler);
        }

        // Handler sur le bouton
        private function btnClick_handler(event : MouseEvent):void
        {
            this.textDisplay.text = "";
        }
    }
}
```

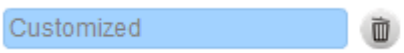
skin.CTextInputSkinCSS.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">

    <!-- host component -->
    <fx:Metadata>
        [HostComponent("components.CTextInputCSS")]
    </fx:Metadata>

    <!-- states -->
    <s:states>
        <s:State name="disabled" />
        <s:State name="normal" />
    </s:states>

    <!-- background -->
    <s:Rect
        left="1" right="27"
        top="1" bottom="1"
```

```
        radiusX="3" radiusY="3">
        <s:fill>
            <s:SolidColor
color="{hostComponent.getStyle('backgroundColor')}"/>
        </s:fill>
        <s:stroke>
            <s:SolidColorStroke
color="{hostComponent.getStyle('borderColor')}"/>
        </s:stroke>
    </s:Rect>

    <!-- Textcomponent -->
    <s:RichEditableText
        id="textDisplay"
        verticalAlign="middle"
        left="1" right="27"
        top="1" bottom="1"
        paddingLeft="3" paddingTop="5"
        paddingRight="3" paddingBottom="3"/>

    <!-- Image button -->
    <mx:Image
        id="btn"
        source="{hostComponent.icon}"
        right="1"
        top="1" bottom="1"/>

</s:Skin>
```

2.5 Exercices

Modifier le composant et le skin afin que la couleur de fond de la zone de texte soit différente lorsque le composant à le focus ou non. De plus ajouter un effet de fondu lorsque la couleur de fond est modifiée.



normal

focused

3 Application client/serveur

L'exemple qui suit met en œuvre une application Flex qui accède à un fichier XML distant. Le contenu du fichier étant utilisé dans le but de peupler une grille de données.

Jusqu'à maintenant, nous avons exécuté nos applications Flex en local. L'objectif est de distribuer ces applications et pour ce faire, nous allons utiliser un serveur web.

3.1 Pré requis

Il est nécessaire d'installer un serveur **Apache** ou tout autre serveur web équivalent.

Dans la suite du document, ***{HTDOCS}*** désigne le chemin du répertoire racine des fichiers du serveur Web (Par exemple *C:/easyphp/www*). Quant à ***{SERVER URL}***, il désigne l'url utilisée pour accéder aux fichiers du serveur (Par exemple *http://localhost*).

3.2 Création de la ressource distante

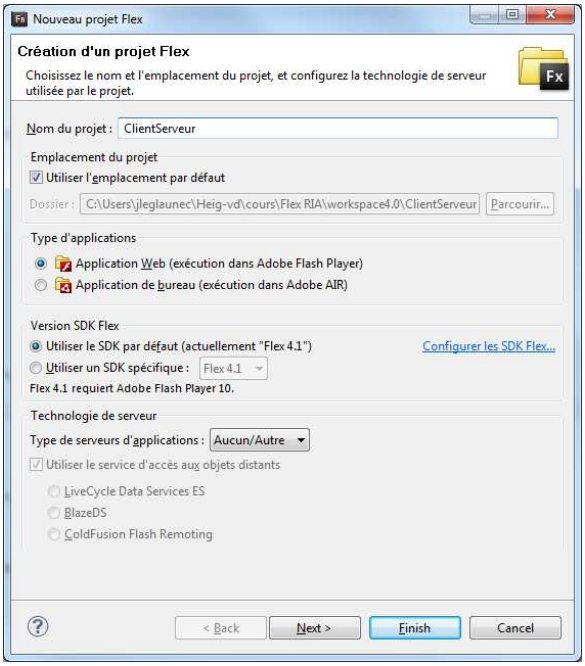
Pour commencer, le fichier **users.xml** doit être déposé dans le dossier ***{HTDOCS} /clientserveur*** du serveur et accessible par une URL semblable à ***{SERVER URL} /clientserveur/users.xml***.

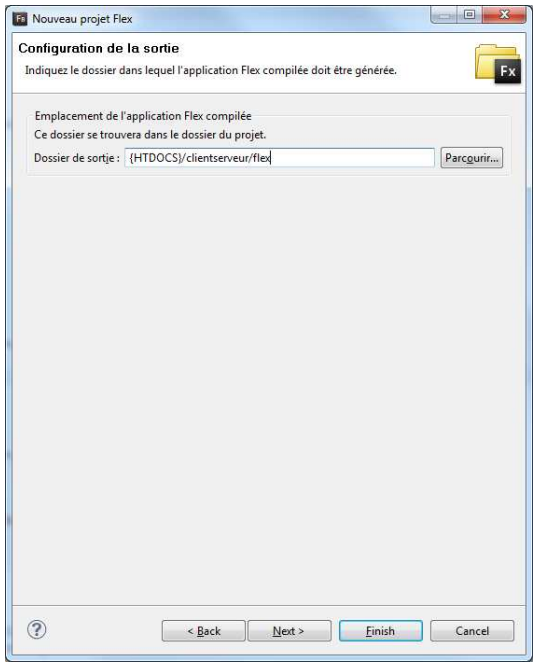
users.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
    <user id="1">
        <name>User 1</name>
        <birthDate>1980-01-01</birthDate>
    </user>
    <user id="2">
        <name>User 2</name>
        <birthDate>1985-04-23</birthDate>
    </user>
    <user id="3">
        <name>User 3</name>
        <birthDate>1986-07-09</birthDate>
    </user>
    <user id="4">
        <name>User 4</name>
        <birthDate>1986-03-19</birthDate>
    </user>
</users>
```

3.3 Création du projet

Revenons à **Flex Builder** afin de créer un nouveau projet.





Attention de bien modifier le champ **Output folder** afin qu'il pointe dans un répertoire du serveur Web. ***{HTDOCS} /clientserveur/flex***

3.4 Chargement du fichier XML distant

Pour charger un fichier XML distant, nous utilisons le composant `HTTPService` avec les paramètres suivants :

Service distant récupérant un fichier XML

```
<s:HTTPService
    id="hseService"
    url="http://localhost/clientserveur/users.xml"
    result="onResult(event)"
    fault="onFault(event)"
    showBusyCursor="true">
```

On transmet l'url de la ressource, le format attendu en retour ainsi que les méthodes de callback associées.

Définition des méthodes de callback en ActionScript

```
[Bindable]
private var remoteXmlFile : XML;

public function onResult(event : ResultEvent) : void
{
    remoteXmlFile = new XML(event.message.body);
}
```

```
public function onFault(event : FaultEvent) : void
{
    Alert.show("error :"+event.message);
}
```

La variable `remoteXmlFile` est précédée de la métadonnée `[Bindable]`. Cela implique que le compilateur Flex va générer automatiquement un événement `propertyChange`. Ceci est indispensable étant donné que la construction de l'interface et le chargement du fichier sont asynchrones. Il est nécessaire qu'un moment donné, l'application informe les composants visuels que la variable a été modifiée et, en d'autres termes, que le fichier a été chargé.

Il suffit maintenant d'appeler la méthode `hseService.send()` à l'initialisation de l'application afin d'envoyer la requête qui nous permettra d'obtenir le fichier distant.

Appel du fichier distant

```
<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"
               minWidth="955" minHeight="600"
               initialize="{hseService.send()}">
```

3.5 Affichage des utilisateurs dans une liste

Maintenant que le fichier XML des utilisateurs a été chargé dans l'application Flex, nous allons afficher son contenu dans un composant **DataGrid**.

Pour ce faire, ajouter le composant suivant

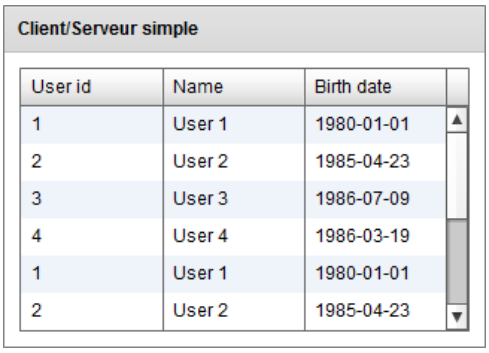
```
<mx:DataGrid
    id="dgdUsers"
    dataProvider="{remoteXmlFile.user}"
    width="100%">
    <mx:columns>
        <mx:DataGridColumn
            dataField="@id"
            headerText="User id"/>
        <mx:DataGridColumn
            dataField="name"
            headerText="Name"/>
        <mx:DataGridColumn
            dataField="birthDate"
            headerText="Birth date"/>
    </mx:columns>
</mx:DataGrid>
```

On remarque que le `dataProvider` fait référence aux utilisateurs présents dans le fichier chargé à l'étape précédente. Les `columns` associent un élément du XML à une colonne de la grille.

3.6 Résultat

Avant d'exécuter l'application, il est nécessaire de se rendre dans l'interface *« Run configurations »* afin de modifier le champ *« URL ou chemin à lancer ».* Dans notre cas, introduire *« {SERVER URL}/clientserveur/flex/ClientServeur.html »*

L'application **ClientServeur** produit l'affichage suivant :



Le code source complet est disponible ci-dessous :

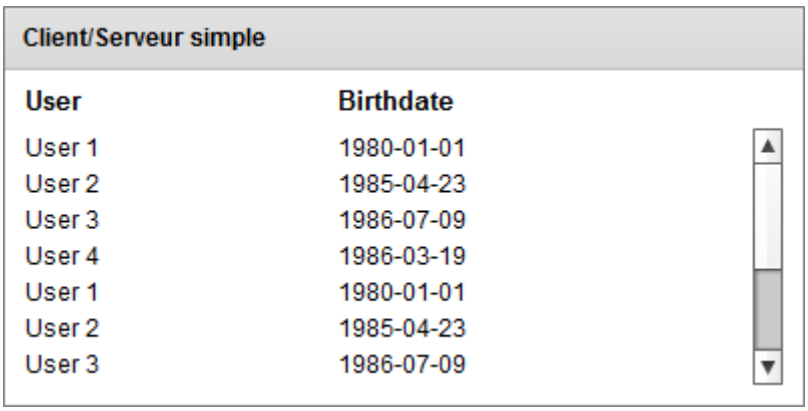
ClientServeur.mxml

```
<?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"
               minWidth="955" minHeight="600"
               initialize="{hseService.send()}">
    <fx:Declarations>
        <!-- Service distant récupérant un fichier XML -->
        <s:HTTPService
            id="hseService"
            url="http://localhost/clientserveur/users.xml"
            result="onResult(event)"
            fault="onFault(event)"
            showBusyCursor="true"/>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            [Bindable]
            private var remoteXmlFile : XML;

            private function onResult(event : ResultEvent) : void
            {
                remoteXmlFile = new XML(event.message.body);
            }

            private function onFault(event : FaultEvent) : void{
                Alert.show("error : "+event.message);
            }
        ]]>
    </fx:Script>
    <!-- Panel principal -->
    <s:Panel
```

```
        id="pnlClientServerSimple"
        title="Client/Serveur simple"
        verticalCenter="1"
        horizontalCenter="1">
        <s:layout>
            <s:VerticalLayout horizontalAlign="center" paddingBottom="10"
paddingLeft="10" paddingRight="10" paddingTop="10"/>
        </s:layout>
        <mx:DataGrid
            id="dgdUsers"
            dataProvider="{remoteXmlFile.user}"
            width="100%">
            <mx:columns>
                <mx:DataGridColumn
                dataField="@id"
                headerText="User id"/>
                <mx:DataGridColumn
                dataField="name"
                headerText="Name"/>
                <mx:DataGridColumn
                dataField="birthDate"
                headerText="Birth date"/>
            </mx:columns>
        </mx:DataGrid>
    </s:Panel>
</s:Application>
```

3.7 Exercice

Créer une application similaire toujours basée sur le fichier XML ***users.xml***. Cette fois-ci, on n'utilisera pas l'objet DataGrid pour afficher le contenu du fichier XML mais uniquement des labels créés « *on-the-fly* » à la réception des données. Le rendu final de l'application doit ressembler à ceci :



3.8 HTTPService avec paramètres

Dans cette application, nous n'allons pas charger un fichier statique mais faire appel à un service dont le résultat dépendra d'un paramètre.

Le service appelé est une page PHP, prenant un paramètre `id`.

*{SERVER URL}/ria/*getUserById.php

```
<?php
    header('Content-Type: application/xml');
    echo '<?xml version="1.0" encoding="UTF-8"?>';
    echo '<user id="'.$_GET["id"].'">';
    echo '<name>User "'.$_GET["id"].'"</name>';
    echo '<birthDate>1980-01-01</birthDate>';
    echo '</user>';
?>
```

Il s'agit de soumettre des valeurs de paramètres au service :

- En utilisant la propriété `request`

```
hseService.request = new Object();
hseService.request.name = txtSrc.text;
```

ou par un objet litéral :

```
hseService.request = {name:txtSrc.text};
```

- En utilisant le sous-élément `<request>`

```
<s:HTTPService
    id="hseService"
    url="http://localhost/ria/sayHello.php"
    …>
    <s:request>
        <name>{txtSrc.text}</name>
    </s:request>
</s:HTTPService>
```

Code source complet (avec paramètre par la propriété request)

```
<?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"
               minWidth="955" minHeight="600"
xmlns:components="components.*">
    <fx:Declarations>
        <s:HTTPService
            id="hseService"
            url="http://localhost/ria/getUserById.php"
            result="onResult(event)"
            fault="onFault(event)"
            showBusyCursor="true">
        </s:HTTPService>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            protected function cmdCallService_click(event:MouseEvent):void
            {
                // V1(a) - Propriété request
                hseService.request = new Object();
                hseService.request.id = txtSrc.text;
                hseService.send();
            }

            private function onResult(event:ResultEvent):void
            {
                var xml:XML = new XML(event.message.body);
                myUserRecord.userId.text = xml.@id;
                myUserRecord.username.text = xml.name;
                myUserRecord.birthDate.text = xml.birthDate;
            }

            private function onFault(event:FaultEvent):void
            {
                Alert.show('Broken service');
            }

        ]]>
    </fx:Script>

    <!-- Panel principal -->
    <s:Panel
        id="pnlClientServerSimple"
        title="Client/Serveur avec paramètres"
        verticalCenter="1"
        horizontalCenter="1">

        <s:layout>
            <s:VerticalLayout
                paddingBottom="10"
                paddingLeft="10"
                paddingRight="10"
                paddingTop="10"/>
        </s:layout>
```

```
        <!-- Zone de recherche -->
        <s:HGroup
            verticalAlign="middle">
            <s:Label
                text="User Id:"/>
            <s:TextInput
                id="txtSrc"/>
            <s:Button
                id="cmdCallService"
                label="Rechercher"
                enabled="{txtSrc.text!=''}"
                click="cmdCallService_click(event)"/>
        </s:HGroup>

        <!-- Colonnes -->
        <s:Group>
            <s:layout>
                <s:HorizontalLayout/>
            </s:layout>
            <s:Label
                width="25"
                text="Id"
                fontWeight="bold"
                fontSize="13"/>
            <s:Label
                width="150"
                text="User"
                fontWeight="bold"
                fontSize="13"/>
            <s:Label
                text="Birthdate"
                fontWeight="bold"
                fontSize="13"/>
        </s:Group>

        <!-- Zone d'affichage -->
        <components:UserRecord
            id="myUserRecord"/>

    </s:Panel>
</s:Application>
```

3.9 Exercices

GeoNames offre un ensemble de services géographiques permettant d'exploiter une base de toponymes avec couverture internationale. Il s'agit ici d'exploiter le service Postal Code Search et notamment son paramètre `placename_startsWith` permettant d'interroger la base pour trouver le nom d'un lieu (une ville). Exemple:

- Créer une RIA permettant de soumettre une telle requête avec un champ de saisie et un bouton d'envoi. Le flux XML résultat sera alors affiché dans une grille de donnée (colonnes = postalcode, name, countryCode, lat, lng).
- Créer une RIA sur la base de la précédente en enlevant le bouton, mais en invoquant la requête à chaque nouveau caractère saisi à partir de 3 caractères. Le résultat doit à présent alimenter un composant `<s:DropDownList>`.
- Créer la même RIA en utilisant le composant `<s:ComboBox>` permettant d'offrir à l'utilisateur une sorte d'auto-complétion à choix multiples.

```
API :
http://www.geonames.org/export/ws-overview.html

Exemple d'appel:
http://ws.geonames.org/postalCodeSearch?placename_startsWith=Frib&maxRows
=10
```

Google offre une API web pour soumettre des recherches depuis n'importe quelle application. L'interface web supporte la méthode GET et fournit un résultat au format JSON.

- Créer une RIA permettant d'initier une recherche sur la base d'un champ de saisie et d'un bouton d'envoi. Le résultat est alors représenté, notamment la propriété "title" au travers du composant `<s:RichText>`.

```
API :
http://code.google.com/apis/ajaxsearch/documentation/reference.html#_intr
o_fonje

as3corelib :
http://github.com/mikechambers/as3corelib
```

Remarque: il est nécessaire d'utiliser la librairie as3corelib et ce afin de décoder le flux JSON résultat.

4 Validation et mise en forme

Nous aurons tôt ou tard besoin de traiter des données fournies par l'utilisateur, cependant il sera nécessaire de vérifier qu'elles respectent un format précis. Ce chapitre introduit la validation de formulaire et le formatage des données (classes `Validator` et `Formatter` du framework Flex).

L'objet `Validator` est un diffuseur d'événement qui contrôle un champ afin de s'assurer qu'il respecte le format requis. Le framework fourni des validateurs prédéfinis (dates, cartes de crédit, emails, …)

Flex Language Reference, package **mx.validators** :

```
http://help.adobe.com/fr_FR/AS3LCR/Flex_4.0/mx/validators/package-
detail.html
```

L'objet `Formatter` quant à lui a pour but de mettre en forme une valeur afin qu'elle soit présentée dans un format spécifique. Il existe également des formatteurs prédéfinis dans le framework.

Flex Language Reference, package **mx.formatters** :

```
http://help.adobe.com/fr_FR/AS3LCR/Flex_4.0/mx/formatters/package-
detail.html
```

4.1 DateValidator

Dans un premier temps, nous allons mettre en œuvre le validateur de date `DateValidator`. L'objectif est de vérifier que le format de saisie de deux dates est `DD.MM.YYYY`. Pour ce faire, nous allons procéder de plusieurs manières :

4.1.1 Validateur pour chaque date

Cette première manière de faire présente l'utilisation d'un validateur par champ à vérifier.

```
<fx:Declarations>
    <mx:DateValidator
        id="date1Validator"
        source="{inputDate1}"
        property="text"
        inputFormat="DD.MM.YYYY"/>
    <mx:DateValidator
        id="date2Validator"
        source="{inputDate2}"
        property="text"
        inputFormat="DD.MM.YYYY"/>
</fx:Declarations>
```

Cela signifie que chaque validateur s'occupe de contrôler son champ date. Les propriétés `source` et `property` permettent de définir la propriété du composant à valider. En l'occurrence, les validateurs s'intéressent à la propriété `text` des deux composants `TextInput`.

Ajoutons ensuite deux `TextInput` à l'interface avec chacun leur écouteur d'événement lorsqu'ils perdent le focus. En effet, c'est à cet instant qu'il faudra vérifier si la date est valide.

```
<!-- Date 1 -->
<s:HGroup
    verticalAlign="middle">
    <s:Label
        text="Date Input 1"/>
    <s:TextInput
        id="inputDate1"
        focusOut="date1Validate(event)"
        width="100"/>
</s:HGroup>

<!-- Date 2 -->
<s:HGroup
    verticalAlign="middle">
    <s:Label
        text="Date Input 2"/>
    <s:TextInput
        id="inputDate2"
        focusOut="date2Validate(event)"
        width="100"/>
</s:HGroup>
```

Event handler

```
private function date1Validate(event : FocusEvent):void
{
    vResult = date1Validator.validate();
    if (vResult.type == ValidationResultEvent.VALID)
        trace("Valid date");
    else
        trace("Invalid date");
}

private function date2Validate(event : FocusEvent):void
{
    vResult = date2Validator.validate();
    if (vResult.type == ValidationResultEvent.VALID)
        trace("Valid date");
    else
        trace("Invalid date");
}
```

Comme nous pouvons le constater, `date1Validator` et `date2Validator` jouent exactement le même rôle, c'est pourquoi il pourrait être intéressant de n'utiliser qu'un validateur pour contrôler la validité des deux dates.

4.1.2 Validateur partagé pour toutes les dates

Cette méthode présente la seconde manière de faire permettant de valider plusieurs dates à l'aide d'un seul et unique validateur. Tout d'abord, créer un `DateValidator` sans spécifier les propriétés `source` et `property`.

```
<fx:Declarations>
    <mx:DateValidator
        id="dateValidator"
        inputFormat="DD.MM.YYYY"/>
</fx:Declarations>
```

Ajoutons ensuite deux TextInput à l'interface avec un écouteur d'événement lorsqu'ils perdent le focus.

```
<s:TextInput
    id="inputDate1"
    focusOut="dateValidate(event)"
    width="100"/>
…
<s:TextInput
    id="inputDate2"
    focusOut="dateValidate(event)"
    width="100"/>
```

Pour terminer, la méthode `dateValidate()` permet de démarrer la validation du champ qui vient de perdre le focus.

```
private function dateValidate(event : FocusEvent):void
{
    dateValidator.listener = event.currentTarget;
    vResult = dateValidator.validate(event.currentTarget.text);

    if (vResult.type == ValidationResultEvent.VALID)
        trace("Valid date");
    else
        trace("Invalid date");
}
```

Etant donné que nous utilisons un validateur pour plusieurs champs, il est nécessaire de définir le composant sur lequel les éventuelles erreurs de validation seront affichées (`dateValidator.listener`).

Il est possible de personnaliser les messages d'erreurs en définissant les propriétés suivantes :

```
invalidCharError="The date contains invalid characters."
wrongDayError="Enter a valid day for the month."
wrongLengthError="Type the date in the format inputFormat."
wrongMonthError="Enter a month between 1 and 12."
wrongYearError="Enter a year between 0 and 9999."
```

4.2 DateFormatter

Nous allons maintenant mettre en oeuvre le formateur de date. Pour ce faire commençons par déclarer un objet DateFormatter.

```
<fx:Declarations>
    <mx:DateFormatter
        id="dateFormatter"/>
</fx:Declarations>
```

Ensuite, définir le format d'affichage désiré en utilisant les « lettres de modèle » provenants de la documentation.

```
private function onComplete():void{
    var today:Date = new Date();

    dateFormatter.formatString = "DD.MM.YYYY";
    lblDateT_1.text = dateFormatter.formatString;
    lblDate_1.text = dateFormatter.format(today);

    dateFormatter.formatString = "EEEE D MMMM YYYY";
    lblDateT_2.text = dateFormatter.formatString;
    lblDate_2.text = dateFormatter.format(today);
}
```

L'affichage des dates s'effectue dans les labels suivants :

```
<mx:HBox>
    <s:Label id="lblDateT_1"/>
    <s:Label id="lblDate_1"/>
</mx:HBox>

<mx:HBox>
    <s:Label id="lblDateT_2"/>
    <s:Label id="lblDate_2"/>
</mx:HBox>
```

Flex Language Reference, Class **DateFormatter** :

```
http://help.adobe.com/fr_FR/AS3LCR/Flex_4.0/mx/formatters/DateFormatter.h
tml
```

4.3 Custom Validator

Nous allons créer un validateur personnalisé qui va s'occuper de vérifier que le format d'une String correspond à un « Language Code » (fr_FR, en_US, …). La vérification s'effectuera à l'aide d'une expression régulière.

Pour commencer, créer un fichier `LanguageCodeValidator.as` qui étend la classe `mx.validators.Validator`. Ce validateur définit une expression régulière définissant la syntaxe d'un code de language.

```
public class LanguageCodeValidator extends Validator
{
    private var languageCode:RegExp = /^[a-z]{2}([_-][A-Z]{2})?$/;
    …
```

Il est nécessaire de surcharger la méthode `doValidation()` afin d'effectuer le traitement de validation. Cette méthode doit retourner un tableau d'objets `ValidationResult`.

Méthode surchargée doValidation()

```
override protected function doValidation(value:Object):Array
{
    // Initialise le tableau de résultats
    var results:Array = [];

    if (languageCode.test(value as String))
        return results;
    else
    {
        // Ajout d'une erreur de validation
        var err:ValidationResult = new ValidationResult(true, "",
                "", "Please enter a correct Country Code");
        results.push(err);
        return results;
    }
}
```

Si la valeur passée en paramètre à la méthode `doValidation()` ne correspond pas à l'expression régulière, un nouvel objet `ValidationResult` est créé. La vérification de la correspondance s'effectue par la méthode `test()` de notre objet `RegExp`.

L'outil « *Flex 3 Regular Expression Explorer* » propose un grand nombre d'exemples d'expressions régulières et permet notamment de tester nos propres expressions.

```
http://ryanswanson.com/regexp/#start
```

4.4 Custom Formatter

Nous allons créer ici un formateur personnalisé qui va accepter une chaîne de caractère afin de la retourner dans un format spécifique. Il permettra de mettre en forme des numéros ISBN (10 ou 13 chiffres). Pour commencer, créer un fichier `ISBNFormatter.as` qui étend la classe `mx.formatters.Formatter`, et qui déclare une variable `formatString`.

```
public class ISBNFormatter extends Formatter
{
    public var formatString : String;// = "####-##-####";
    …
```

Il est nécessaire de surcharger la méthode `format()` afin d'effectuer notre traitement de mise en forme. Ceci se fait en plusieurs étapes :

- Vérification de la longueur de la String à mettre en forme
- Vérification du nombre de # dans le formatString
- Formattage du contenu à l'aide de l'utilitaire `SwitchSymbolFormatter`

Méthode format()

```
override public function format(value:Object):String
{
    // Vérifie la longueur de la string ISBN (10 ou 13 caractères)
    if (!(value.toString().length == 10 ||
          value.toString().length == 13))
    {
        error = "Invalid String Length";
        return "";
    }
    // Vérifie le nombre de # dans la formatString (10 ou 13)
    var numCharCnt:int = 0;
    for (var i:int = 0; i<formatString.length; i++)
    {
        if ( formatString.charAt(i) == "#")
            numCharCnt++;
    }
    if (!(numCharCnt == 10 || numCharCnt == 13))
    {
        error = "Invalid Format String";
        return "";
    }
    // Si la valeur et le formatString  sont valides, formatter le contenu
    var dataFormatter:SwitchSymbolFormatter = new SwitchSymbolFormatter();
    return dataFormatter.formatValue(formatString, value);
}
```

La classe utilitaire SwitchSymbolFormatter est très utilisée lors de la création de formateurs personnalisés. Elle permet de substituer les caractères réservés d'une chaine par des nombres issus d'une autre chaîne. Par exemple, l'appel de `dataFormatter.formatValue("#-#","12")` retourne une String `"1-2"`.

4.5 Exercice

Créer un validateur personnalisé permettant de vérifier qu'un groupe de checkbox contient entre *min* et *max* éléments cochés.

5 Application Client/Serveur avec AMFPHP

5.1 Pré requis

Pour cet exemple, il est nécessaire:

- D'installer **Apache avec un module PHP5** ou tout autre serveur web équivalent.
- Télécharger **AMFPHP 1.9** et déposer le contenu de l'archive dans ***{HTDOCS}/amfphp***

5.2 Création d'un service AMFPHP

Dans *{HTDOCS}/amfphp/services*, créer un dossier myservice dans lequel on ajoute le script *MyService.php*

MyService.php

```
<?php
class MyService{
    public function hello(){
        return "Salut !";
    }
}
?>
```

Il est possible de tester l'appel de la méthode en accédant à l'application Flex livrée avec AMFPHP : ***{SERVER URL}/amfphp/browser***

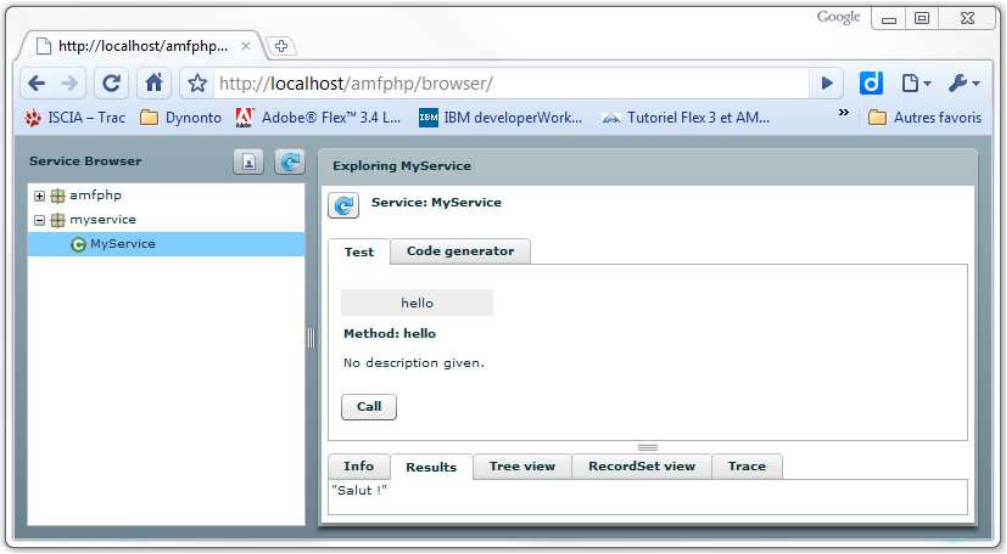


Figure 3 - Interface de test AMFPHP

5.3 Création du projet Flex

Créer un projet **ClientServeurAMFPHP** en spécifiant le répertoire de sortie à ***{HTDOCS} /clientserveur_amf***

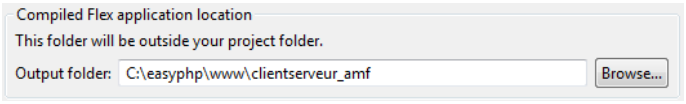


Figure 4 - Répertoire de sortie

La première étape consiste à créer un fichier *services-config.xml* à la racine du projet permettant de faire le lien entre le back-end AMFPHP et le front-end Flex.

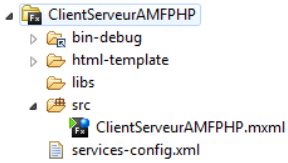


Figure 5 - Fichier services-config.xml

services-config.xml

```
<?xml version="1.0" encoding="utf-8"?>
<services-config>
    <services>
        <service
            id="amfphp-remoting"
            class="flex.messaging.services.RemotingService"
            messageTypes="flex.messaging.messages.RemotingMessage">
            <destination id="amfphp">
                <channels>
                    <channel ref="my-amfphp"/>
                </channels>
                <properties>
                    <source>*</source>
                </properties>
            </destination>
        </service>
    </services>
    <channels>
        <channel-definition
            id="my-amfphp"
            class="mx.messaging.channels.AMFChannel">
            <endpoint
                uri="http://localhost/amfphp/gateway.php"
                class="flex.messaging.endpoints.AMFEndpoint"/>
        </channel-definition>
    </channels>
</services-config>
```

`amfphp` : Nom identifiant la destination
`http://localhost/amfphp/gateway.php` : URL du gateway AMFPHP

Pour que le fichier XML soit pris en considération, il est nécessaire d'ajouter un argument dans ***Project properties → Compilateur Flex → Arguments de compilateur supplémentaires***.

Argument à ajouter

```
-services "{Path_to_FlexBuilderWorkspace}/ClientServeurAMFPHP/services-config.xml"
```

{Path_to_FlexBuilderWorkspace} correspond au chemin absolu vers le workspace en cours d'utilisation par Flash Builder.

5.4 Appel d'un service sans argument

Pour appeler le service AMFPHP précédemment créé, nous utilisons le composant Flex ***RemoteObject***. Cet objet fait référence à la destination définie dans le fichier *services-config.xml* et détermine les méthodes utilisées dans l'application.

Déclaration d'un RemoteObject

```
<fx:Declarations>
    <s:RemoteObject
        id="backendService"
        showBusyCursor="true"
        destination="amfphp"
        source="myservice.MyService">
        <s:method
            name="hello"
            result="trace(event.result as String)"/>
    </s:RemoteObject>
</fx:Declarations>
```

Invocation de méthode distante

```
<s:Button
    label="Call hello"
    click="backendService.getOperation('hello').send()"/>
```

Pour tester le fonctionnement, démarrez l'application en mode **debug**, puis cliquez sur le bouton. Si tout se passe comme prévu, vous devriez voir *« Salut ! »* dans la console.

5.5 Création et appel d'un service avec argument

Nous allons maintenant créer un nouveau service côté serveur prenant un *username* ainsi qu'un *password* en argument et retournant vrai si le couple de valeurs correspond à « *root/1234* ».

Dans le fichier MyService.php, ajouter la fonction suivante

```
public function authenticate($user, $pass){
    return strcmp($user, "root") == 0 && strcmp($pass, "1234") == 0;
}
```

Ajouter la méthode suivante au RemoteObject du fichier ClientServeurAMFPHP.mxml

```
<s:method
    name="authenticate"
    result="trace(event.result as Boolean)">
    <s:arguments>
        <user>root</user>
        <pass>1234</pass>
    </s:arguments>
</s:method>
```

Ajouter le bouton suivant à l'application

```
<s:Button
    label="Check password"
    click="backendService.getOperation('authenticate').send();"/>
```

Le passage d'un paramètre s'effectue à l'aide des balises `<s:arguments>`

5.6 Exercice

Modifier le code de «

Exemple 3, Application Login » afin d'implémenter une authentification au niveau du service amfphp précédemment mis en place. L'authentification se fait de manière statique (vérifier que le « *Username = root* » et le « *Password = 1234* ») mais obligatoirement par un service amfphp.

5.7 Mapping de classes

Jusqu'à maintenant, nous avons échangé entre le client et le serveur uniquement des objets de type primitif. Si nous voulons transmettre des objets personnalisés, il est intéressant de mapper les classes afin d'éviter de devoir reconstruire les objets manuellement à partir de tableaux ou d'un flux XML.

Commençons par créer un dossier *vo/myservice* dans *{HTDOCS}/amfphp/services.*

Créer un fichier Person.php dans ce nouveau dossier

```
<?php
class Person
{
    var $firstName;
    var $lastName;
    var $_explicitType = "myservice.Person";

    function getFirstName(){
        return $this->firstName;
    }
}
?>
```

Inclure le fichier Person.php dans le fichier MyService.php

```
require_once "../vo/myservice/Person.php";
```

Toujours dans le fichier MyService.php, ajouter les fonctions suivantes

```
public function getFirstName($p){
    return $p->getFirstName();
}

public function getPerson(){
    $p = new Person();
    $p->firstName = "myFirstName";
    $p->lastName = "myName";
    return $p;
}
```

Dans le projet Flex, créer un fichier *src/myservice/Person.as*.

Person.as

```
package myservice
{
    [RemoteClass(alias="myservice.Person")]
    public class Person
    {
        public var firstName:String;
        public var lastName:String;
    }
}
```

Modifier l'application pour appeler les nouveaux services.

ClientServeurAMFPHP.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"
               minWidth="955" minHeight="600"
               creationComplete="init()">
    <fx:Declarations>
        <!-- Services distants -->
        <s:RemoteObject
            id="backendService"
            showBusyCursor="true"
            destination="amfphp"
            source="myservice.MyService">
            <s:method
                name="hello"
                result="trace(event.result as String)"/>
            <s:method
                name="authenticate"
                result="trace(event.result as Boolean)">
                <s:arguments>
                      <user>root</user>
                      <pass>1234</pass>
                </s:arguments>
            </s:method>
            <s:method
                name="getPerson"
                result="trace(event.result as Person)"/>
            <s:method
                name="getFirstName"
                result="trace(event.result as String)">
                <s:arguments>
                      <arg>{person}</arg>
                </s:arguments>
            </s:method>
        </s:RemoteObject>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[

        import myclient.Person;
        import mx.controls.Alert;

        [Bindable]
```

```
        public var person:Person = new Person();

        public function init():void
        {
            person.firstName = "Foo";
            person.lastName = "Bar";
        }

        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout horizontalAlign="center"/>
    </s:layout>

    <!-- Boutons utilisés pour invoquer les services -->
    <s:Button
        label="Call hello"
        click="backendService.getOperation('hello').send()"/>
    <s:Button
        label="Check password"
        click="backendService.getOperation('authenticate').send();"/>
    <s:Button
        label="Get person"
        click="backendService.getOperation('getPerson').send();"/>
    <s:Button
        label="Get first name"
        click="backendService.getOperation('getFirstName').send();"/>
</s:Application>
```

5.8 Exercice récapitulatif

Sur la base des réalisations précédentes autour du scénario login/register (tutoriel n°1), il s'agit de créer la RIA complète y correspondant, avec dialogue client/serveur. Le formulaire d'enregistrement sera aussi plus étoffé, avec : nom, prenom, date de naissance, login, password, et email. Ces champs sont tous obligatoires et doivent être validés (règles de validation à définir).

- La RIA doit interagir avec des services, pour l'authentification et pour l'enregistrement. Pour simplifier, la base des utilisateurs est stockée sous forme de fichier type CSV (on ignore les problèmes d'accès concurrentiel).
- L'opération de login affiche comme résulat le message approprié, et l'enregistrement affiche un rappel du login/password nouvellement créé.

Remarque: l'idée de cette réalisation est de mettre en place un échange client/serveur basé sur AMFPHP et utilisant un accès aux services par le composant RemoteObject.

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

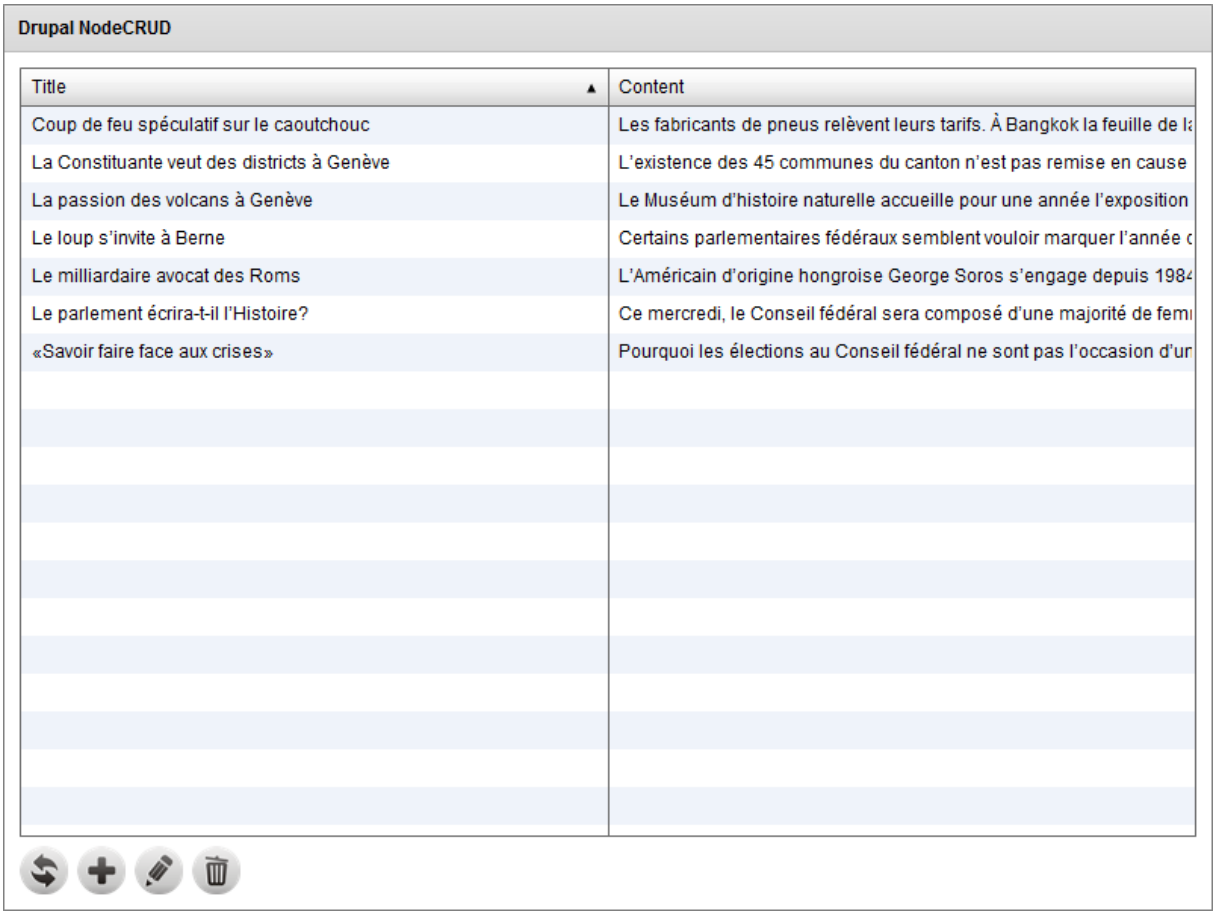
```
private function onInitialize() : void
{
    …
    facade.sendNotification(ApplicationFacade.LOADUSERS);
}
```

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.CREATENODE_SUCCESS avec pour paramètre le nœud créé Object(nid, title, body)
Retour en cas d'échec: • Envoi de la notification `ApplicationFacade.CREATENODE_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.UPDATENODE_SUCCESS avec pour paramètre le nœud modifié Object(nid, title, body)
Retour en cas d'échec: • Envoi de la notification `ApplicationFacade.UPDATENODE_FAILED`

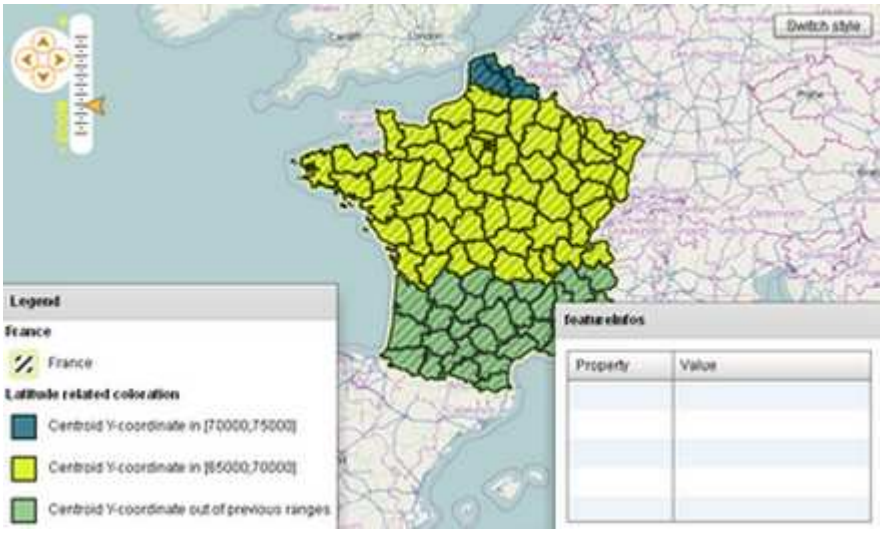
NodesProxy.delete() => Supprimer un nœud Object(nid, title, body)
Retour en cas de succès : • Envoi de la notification ApplicationFacade.DELETENODE_SUCCESS avec pour paramètre le nœud supprimé Object(nid, title, body)
Retour en cas d'échec: • Envoi de la notification `ApplicationFacade.DELETENODE_FAILED`

7 OpenScales

OpenScales est un framework cartographique open source écrit en ActionScript 3/Flex permettant aux développeurs de créer des **Rich Internet Mapping Application**. Dans ce tutoriel, nous allons mettre en œuvre le framework à travers une série d'exemples.

Pour commencer, télécharger **OpenScales 1.2** à l'adresse ci-dessous, puis copier les fichiers *swc* dans le dossier *libs* du projet Flash Builder.

```
http://openscales.org/downloads/index.html
```



Documentation:

```
http://openscales.org/documentation/index.html
```

Application de démonstration :

```
http://openscales.org/demo/index.html
```

L'archive téléchargée contient également les sources des exemples du démonstrateur.

8 Premiers pas

8.1 Basic OpenStreetMap

Pour commencer, nous allons créer une carte basée sur la couche Mapnik OpenStreetMap. Pour ce faire, créer un objet Map dans lequel on ajoute une couche Mapnik (on parle de « layer »). Ces composants proviennent de la librairie OpenScales et il est nécessaire d'ajouter le namespace http://openscales.org.

BasicOsm.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"
               minWidth="955" minHeight="600"
               xmlns:os="http://openscales.org">
    <os:Map
        id="fxmap"
        width="100%"
        height="100%">
        <os:Mapnik
            name="Mapnik"
            proxy="http://openscales.org/proxy.php?url="/>
    </os:Map>
</s:Application>
```

8.1.1 Résultat

L'application affiche une carte figée dans laquelle il n'est pas possible de se déplacer/zoomer :

Le composant de couche Mapnik permet de charger des tuiles cartographiques prégénérées sur toute une zone à différents niveaux de zoom depuis un cache serveur.

8.2 MouseControls, MousePosition et PanZoom

Nous allons maintenant ajouter des ***MouseControls*** permettant de naviguer dans la carte, à savoir:

- Zoomer à l'aide de la molette de la souris (WheelHandler)
- Déplacer la carte à l'aide du Drag and Drop. (DragHandler)

```
<os:WheelHandler/>
<os:DragHandler/>
```

Nous allons également ajouter un composant ***MousePosition*** permettant d'afficher les coordonnées géographiques correspondant à la position de la souris.

```
<os:MousePosition
    x="10"
    y="{fxmap.height-20}"
    displayProjection="EPSG:4326"/>
```

Pour en savoir plus sur les codes EPSG :

```
http://fr.wikipedia.org/wiki/Syst%C3%A8me_de_coordonn%C3%A9es_g%C3%A9or%C
3%A9f%C3%A9renc%C3%A9es#Les_codes_EPSG
```

Puis nous allons ajouter l'outil ***PanZoom*** à la carte. Pour cela, il est nécessaire d'ajouter du code ActionScript permettant de récupérer l'instance associée au composant MXML Map et de la stocker dans une variable « bindée ». Cette méthode `initMap()` est appelée à l'initialisation de l'application.

```
<fx:Script>
    <![CDATA[
        import org.openscales.core.Map;
        [Bindable] private var map:Map = null;
        private function initMap():void
        {
            map = fxmap.map;
        }
    ]]>
</fx:Script>
```

Puis il faut créer un composant `PanZoom` basé sur l'instance `map` (variable « bindée » créée précédemment).

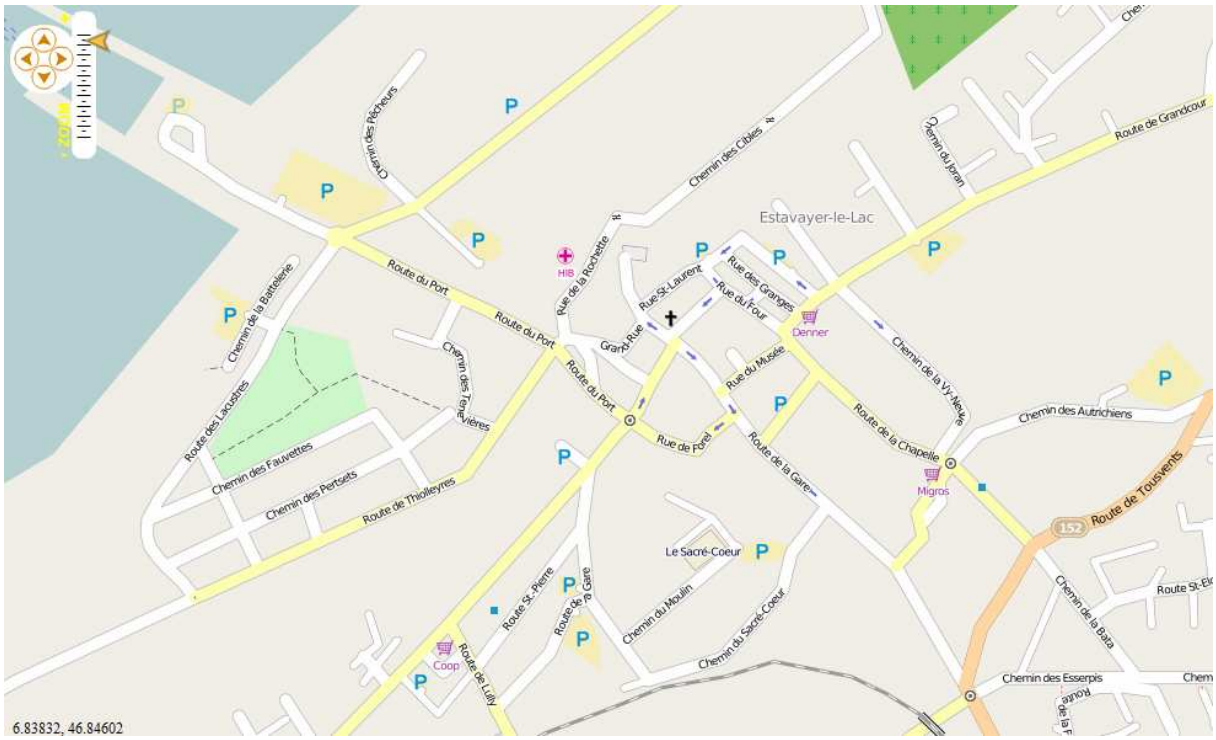
```
<os:PanZoom
    map="{map}"
    x="{fxmap.x+10}"
    y="{fxmap.y+10}"/>
```

BasicOsm_Controls.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"
               minWidth="955" minHeight="600"
               xmlns:os="http://openscales.org"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="100%"
        height="100%">

        <os:Mapnik
            name="Mapnik"
            proxy="http://openscales.org/proxy.php?url="/>

        <!-- Mouse controls -->
        <os:DragHandler/>
        <os:WheelHandler/>

        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>

    </os:Map>

    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            [Bindable] private var map:Map = null;
            private function initMap():void
            {
                map = fxmap.map;
            }
        ]]>
    </fx:Script>

</s:Application>
```

8.2.1 Résultat

Il est maintenant possible de se déplacer sur la carte à l'aide de la souris ou directement depuis l'outil de navigation situé en haut à gauche. Quant aux coordonnées géographiques de la position de la souris, elles sont affichées en bas à gauche de la carte.



8.3 OpenStreetMap centré sur Yverdon

Nous allons maintenant centrer la carte sur Yverdon. Pour ce faire, ajouter les propriétés **center** et **zoom** à notre composant Map :

- **center** correspond à la coordonnée géographique du centre de la carte.
- **zoom** correspond au niveau de zoom à utiliser

BasicOsm_Controls_Center.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"
               minWidth="955" minHeight="600"
               xmlns:os="http://openscales.org"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="100%"
        height="100%"
        center="6.64263,46.79094"
        zoom="15">
        <os:Mapnik
            name="Mapnik"
            proxy="http://openscales.org/proxy.php?url="/>

        <!-- Mouse controls -->
        <os:DragHandler/>
        <os:WheelHandler/>

        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>

    </os:Map>

    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            [Bindable] private var map:Map = null;
            private function initMap():void
            {
                map = fxmap.map;
            }
        ]]>
    </fx:Script>

</s:Application>
```

8.3.1 Résultat

La carte est maintenant centrée sur Yverdon.

8.4 Composant WMS

Cet exemple met en œuvre l'utilisation du composant WMS qui permet de charger une couche depuis un serveur cartographique conforme OGC WMS. Pour ce faire, nous allons faire appel à un tel service conforme du SITN (Système d'Information du Territoire Neuchâtelois).

Pour en savoir plus sur WMS (Web Map Service):

```
http://fr.wikipedia.org/wiki/Web_Map_Service
```

Service WMS ImageOne2006 du SITN :

```
http://www.ne.ch/neat/site/jsp/rubrique/rubrique.jsp?StyleType=bleu&DocId
=33442
```

Le composant WMS permet de préciser l'url d'un tel service, la couche à charger, ainsi que le format d'image demandé :

```
<os:WMS
    name="SITN WMS - Ortho layer"
    url="http://sitn.ne.ch/ogc-sitn-open/wms"
    layers="ortho"
    format="image/jpeg"
    proxy="http://localhost/proxy.php?url="/>
```

BasicWms.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:os="http://openscales.org"
               minWidth="955" minHeight="600">

    <os:Map
        id="fxmap"
        width="100%"
        height="100%">

<os:WMS
    name="SITN WMS - Ortho layer"
    url="http://sitn.ne.ch/ogc-sitn-open/wms"
    layers="ortho"
    format="image/jpeg"
    proxy="http://localhost/proxy.php?url="/>

        <os:Extent
            west="6.30" south="46.80"
            east="7.13" north="47.20"
            projection="EPSG:4326" />

        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>

        <os:DragHandler/>
        <os:WheelHandler/>

    </os:Map>
</s:Application>
```

L'utilisation d'un tel service requiert de préciser la zone cartographique demandée et ce par la définition d'une enveloppe géographique (Extent, avec les valeurs west, south, east, north).

8.4.1 Résultat

L'application affiche une orthophoto du canton de Neuchâtel :

8.5 CurrentExtent

L'intérêt d'un tel framework est de pouvoir définir des interactions cartographiques personnalisées. Ainsi il est possible de gérer des événements spécifiques (MapEvent, LayerEvent).

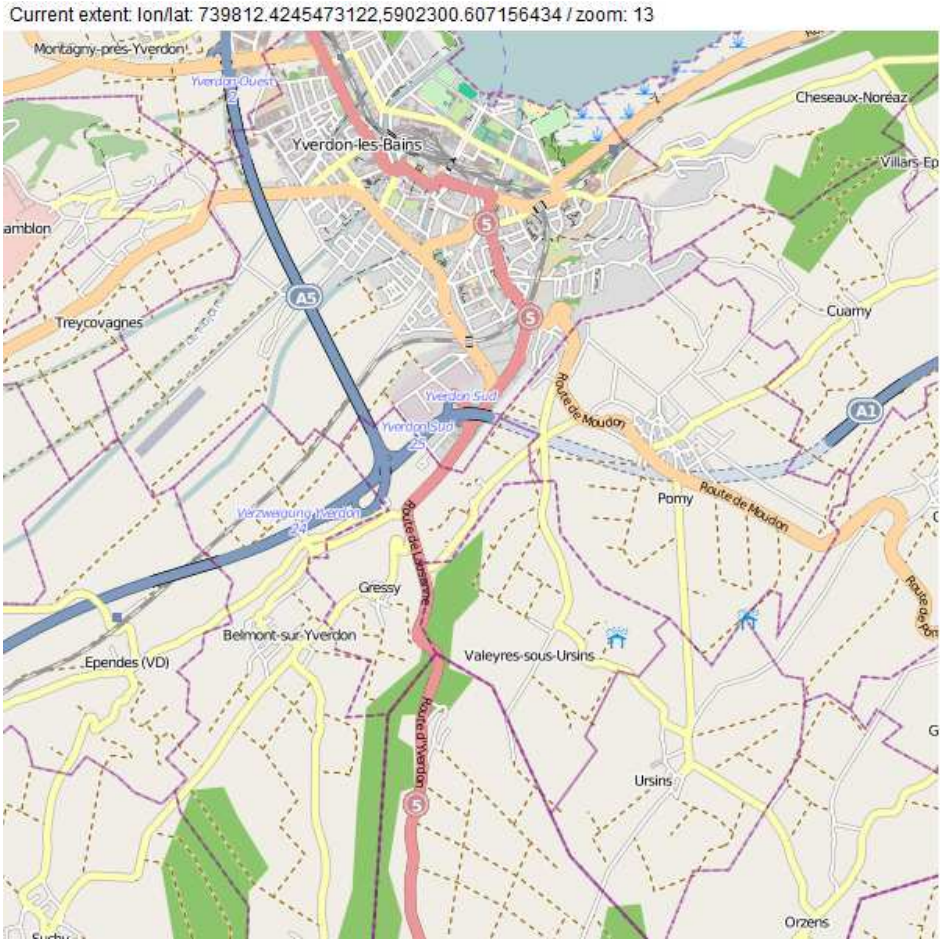
Le code ci-dessous ajoute un listener sur un MapEvent.MOVE_END (fin de déplacement de carte). Il s'agit à chaque déplacement d'afficher les nouvelles caractéristiques de la vue cartographique (map.center et map.zoom).

CurrentExtent.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:os="http://openscales.org"
               minWidth="955" minHeight="600"
               creationComplete="initMap()">

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            import org.openscales.core.events.MapEvent;

            [Bindable] private var map:Map = null;
            [Bindable] private var currentExtent:String;

            private function initMap():void {
                map = fxmap.map;
                getCurrentExtent();
                map.addEventListener(MapEvent.MOVE_END,
                                     getCurrentExtent);
            }

            private function getCurrentExtent(event:MapEvent = null):void
            {
                currentExtent = "lon/lat: "+map.center.lon+","
                               +map.center.lat+" / zoom: "+map.zoom;
            }

        ]]>
    </fx:Script>

    <os:Map id="fxmap"
            width="600" height="600"
            y="20"
            zoom="9" center="6.8,46.9">
        <os:Mapnik name="base"/>
        <os:MousePosition x="10" y="{fxmap.height-20}"
                          displayProjection="EPSG:900913"/>
        <os:DragHandler/>
        <os:WheelHandler/>
    </os:Map>

    <s:Label x="5" y="5" text="Current extent: {currentExtent}"/>

</s:Application>
```

8.5.1 Résultat

Le centre de la map ainsi que le niveau de zoom sont affichés après chaque déplacement.



8.6 Exercice

Mettre en place une application cartographique qui enregistre les différentes vues durant la navigation. Les caractéristiques des vues sont historisées dans une DataGrid. L'utilisateur peut revenir à tout moment à une vue précédente de l'historique par sélection dans la DataGrid.

9 Superpositions (overlay)

9.1 Basic WMS overlay

Cet exemple met en œuvre l'utilisation d'une couche WMS affichée par-dessus la couche de base Mapnik. Pour ce faire, nous allons faire appel au service WMS du serveur OGO.

- Url du service : `http://ogo.heig-vd.ch/geoserver/wms`
- Couche : `ogo:g4districts98`

Etant donné que la couche de base Mapnik utilise une projection EPSG:900913, il faut spécifier explicitement que l'on désire obtenir la couche WMS dans cette même projection. Cette couche s'ajoute donc à la carte comme suit :

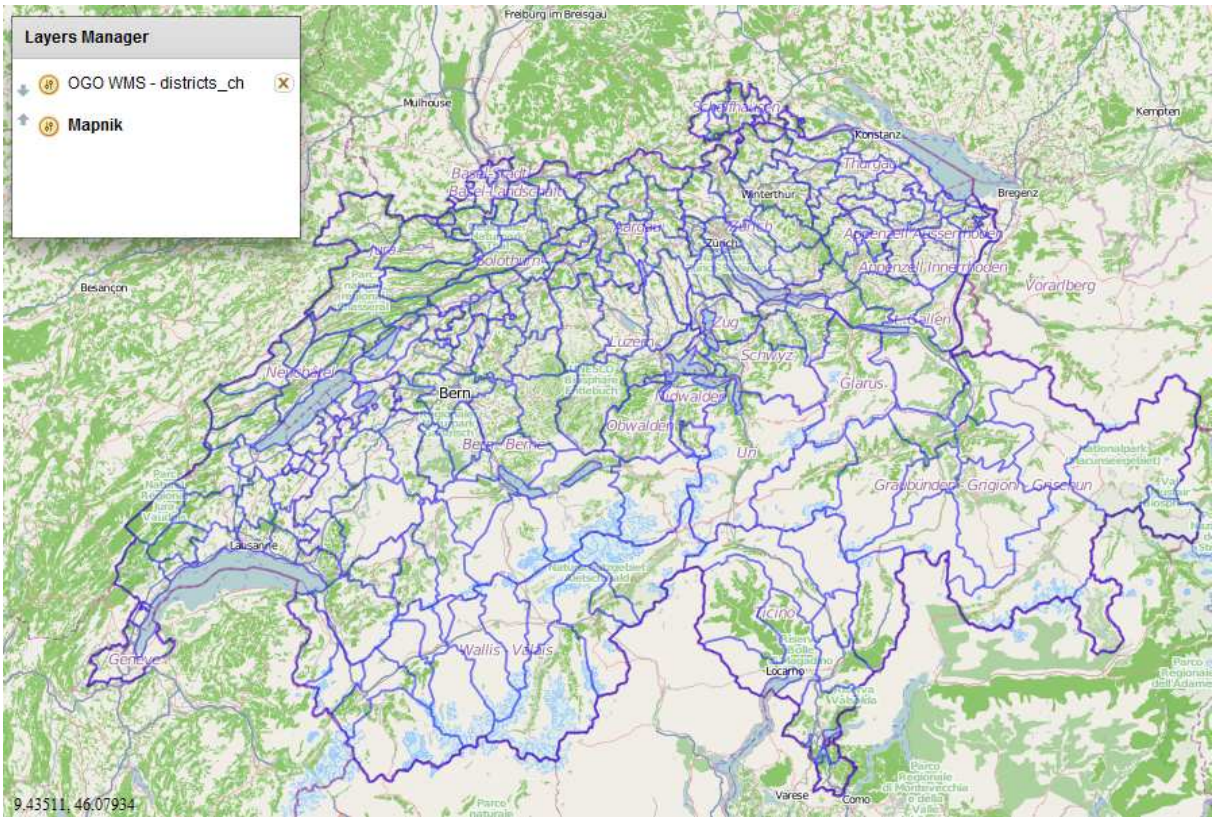
```
<os:WMS
    name="OGO WMS - districts_ch" alpha="0.5"
    url="http://ogo.heig-vd.ch/geoserver/wms"
    layers="ogo:districts_ch" projection="EPSG:900913"
    transparent="true" format="image/png"  styles="line"/>
```

Wms_Overlay.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:os="http://openscales.org"
               minWidth="955" minHeight="600">

    <os:Map
        id="fxmap"
        width="100%"
        height="100%"
        center="8.32,46.9"
        zoom="8">

        <os:Mapnik
            name="Mapnik"
            proxy="http://openscales.org/proxy.php?url="/>

        <os:WMS
            name="OGO WMS - districts_ch" alpha="0.5"
            url="http://ogo.heig-vd.ch/geoserver/wms"
            layers="ogo:districts_ch" projection="EPSG:900913"
            transparent="true" format="image/png" styles="line"/>

        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>

        <os:DragHandler/>
        <os:WheelHandler/>

        <os:ControlPanel
            title="Layers Manager"
            x="10" y="10"
```

```
            width="215">
            <os:LayerManager/>
        </os:ControlPanel>
    </os:Map>
</s:Application>
```

9.1.1 Résultat



L'exemple `Wms_Overlay_Mapnik_Sitn_Ogo.mxml` présent dans les sources met en œuvre l'utilisation de trois couches provenants de différents services.

9.2 Superposition d'objets géographiques – point

Il est possible d'ajouter des objets géographiques points, lignes et polygones à notre carte (on parle de Features). Tout objet est caractérisé par une géométrie et des attributs.

9.2.1 Création de la couche d'objets

Pour ce faire, il est nécessaire de créer une nouvelle couche :

```
var featureLayer:FeatureLayer = new FeatureLayer("MyFeatures");
featureLayer.projection = new ProjProjection("EPSG:4326");
featureLayer.style = Style.getDefaultPointStyle();
```

L'ajout d'un point peut se faire de plusieurs manières:

- L'ajout d'un `PointFeature`, dont la symbologie est celle définie par défaut sur la couche (`Style.getDefaultPointStyle`).

```
var point:PointFeature = PointFeature.createPointFeature(
                                    new Location(6.65591, 46.78566));
featureLayer.addFeature(point);
```

- L'ajout d'un `Marker`, dont la symbologie est celle définie par défaut pour les marqueurs.

```
var marker:Marker = new Marker(
                new org.openscales.geometry.Point(6.64282, 46.79092));
featureLayer.addFeature(marker);
```

- L'ajout d'un `CustomMarker`, dont la symbologie peut être personnalisée.

```
var marker_2:PointFeature =
CustomMarker.createUrlBasedMarker("http://ogo.heig-
                vd.ch/ria/images/add_placemark.png",
                new Location(6.64922, 46.78637),
                {featureName:"Custom marker"});
featureLayer.addFeature(marker_2);
```

PointsAndMarkers_Overlay.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:os="http://openscales.org"
               minWidth="955" minHeight="600"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="800"
        height="600"
        zoom="15"
        center="6.64282, 46.79092"
        x="100"
        y="100">
        <os:Mapnik
            name="base"
            proxy="http://openscales.org/proxy.php?url="/>
        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>
        <os:DragHandler/>
        <os:WheelHandler/>
    </os:Map>

    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            import org.openscales.core.feature.CustomMarker;
            import org.openscales.core.feature.Marker;
            import org.openscales.core.feature.PointFeature;
            import org.openscales.core.layer.FeatureLayer;
            import org.openscales.core.style.Style;
            import org.openscales.geometry.Point;
            import org.openscales.geometry.basetypes.Location;
            import org.openscales.proj4as.ProjProjection;

            [Bindable] private var map:Map = null;

            private function initMap():void{
                map = fxmap.map;

                var featureLayer:FeatureLayer = new
                            FeatureLayer("PointsFeatures");
                featureLayer.projection = new
                            ProjProjection("EPSG:4326");
                featureLayer.style = Style.getDefaultPointStyle();
                map.addLayer(featureLayer);

                // Ajout du premier PointFeature
                var point_1:PointFeature =
                      PointFeature.createPointFeature(
                                  new Location(6.65591, 46.78566),
                                  {featureName:"Point 1"});
                featureLayer.addFeature(point_1);
```

```
                // Ajout du deuxième PointFeature
                var point_2:PointFeature =
                                PointFeature.createPointFeature(
                                        new Location(6.64046, 46.79113),
                                        {featureName:"Point 2"});
                featureLayer.addFeature(point_2);

                // Ajout d'un marqueur
                var marker_1:Marker = new Marker(
                new org.openscales.geometry.Point(6.64282, 46.79092),
                                {featureName:"Default marker"});
                featureLayer.addFeature(marker_1);

                // Ajout d'un marqueur personnalisé
                var marker_2:PointFeature =
                        CustomMarker.createUrlBasedMarker("http://ogo.heig-
vd.ch/ria/images/add_placemark.png",
                                new Location(6.64922, 46.78637),
                                {featureName:"Custom marker"});
                featureLayer.addFeature(marker_2);
            }
        ]]>
    </fx:Script>
</s:Application>
```

9.2.2 Résultat

9.3 Superposition d'objets géographiques – lignes

Il est possible d'ajouter des lignes à notre carte. Pour ce faire, il est nécessaire de créer une nouvelle couche :

```
var featureLayer:FeatureLayer = new FeatureLayer("LineFeatures");
featureLayer.projection = new ProjProjection("EPSG:4326");
featureLayer.style = Style.getDefaultLineStyle();
```

Les coordonnées de la ligne doivent être composées dans un vecteur de la manière suivante :

```
var vector_1:Vector.<Number> = new Vector.<Number>;
vector_1.push(6.65591);       //P1[x]
vector_1.push(46.78566);      //P1[y]
vector_1.push(6.64922);       //P2[x]
vector_1.push(46.78637);      //P2[y]
vector_1.push(6.64046);       //P3[x]
vector_1.push(46.79113);      //P3[y]
vector_1.push(6.64282);       //P4[x]
vector_1.push(46.79092);      //P4[y]
```

La création d'un objet de type ligne (`LineStringFeature`) se fait au travers d'une géométrie (`LineString`) à partir du vecteur créé ci-dessus.

```
var line_1:LineString = new LineString(vector_1);
var lineFeature:LineStringFeature = new LineStringFeature(line_1,
                                                {featureName:"Line 1"});
featureLayer.addFeature(lineFeature);
```

LineString_Overlay.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:os="http://openscales.org"
               minWidth="955" minHeight="600"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="800"
        height="600"
        zoom="15"
        center="6.64282, 46.79092"
        x="100"
        y="100">
        <os:Mapnik
            name="base"/>
        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>
        <os:DragHandler/>
        <os:WheelHandler/>
    </os:Map>
```

```
    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            import org.openscales.core.feature.LineStringFeature;
            import org.openscales.core.layer.FeatureLayer;
            import org.openscales.core.popup.Anchored;
            import org.openscales.core.style.Style;
            import org.openscales.geometry.LineString;
            import org.openscales.proj4as.ProjProjection;

            [Bindable] private var map:Map = null;

            private var currentPopup:Anchored;

            private function initMap():void{
                map = fxmap.map;
                var featureLayer:FeatureLayer = new
                                    FeatureLayer("MyFeatures");
                featureLayer.projection = new
                                    ProjProjection("EPSG:4326");
                featureLayer.style = Style.getDefaultLineStyle();

                // Création du vecteur de la LineString
                var vector_1:Vector.<Number>=new Vector.<Number>;
                vector_1.push(6.65591);    //P1[x]
                vector_1.push(46.78566);   //P1[y]
                vector_1.push(6.64922);    //P2[x]
                vector_1.push(46.78637);   //P2[y]
                vector_1.push(6.64282);    //P2[x]
                vector_1.push(46.79092);   //P2[y]
                vector_1.push(6.64046);    //P3[x]
                vector_1.push(46.79113);   //P4[y]

                // Création de la LineString
                var line_1:LineString = new LineString(vector_1);
                var lineFeature:LineStringFeature = new
                       LineStringFeature(line_1, {featureName:"Line 1"});
                featureLayer.addFeature(lineFeature);

                map.addLayer(featureLayer);
            }
        ]]>
    </fx:Script>
</s:Application>
```

9.3.1 Résultat

9.4 Superposition d'objets géographiques – polygones

Il est possible d'ajouter des polygones à notre carte. Pour ce faire, il est nécessaire de créer une nouvelle couche :

```
var featureLayer:FeatureLayer = new FeatureLayer("PolygonFeatures");
featureLayer.projection = new ProjProjection("EPSG:4326");
featureLayer.style = Style.getDefaultSurfaceStyle();
```

Les coordonnées du polygone sont à entrer dans un vecteur de la manière suivante :

```
var vector_1:Vector.<Number> = new Vector.<Number>;
vector_1.push(6.65591);       //P1[x]
vector_1.push(46.78566);      //P1[y]
vector_1.push(6.64922);       //P2[x]
vector_1.push(46.78637);      //P2[y]
vector_1.push(6.64046);       //P3[x]
vector_1.push(46.79113);      //P3[y]
vector_1.push(6.64282);       //P4[x]
vector_1.push(46.79092);      //P4[y]
```

La création d'un objet de type polygone (`PolygonFeature`) se fait au travers d'un vecteur de géométries (`LinearRing`) à partir du vecteur créé ci-dessus. Une géométrie `LinearRing` est une `LineString` spéciale qui est automatiquement « fermée ».

```
var ring_1:LinearRing = new LinearRing(vector_1);
var geom:Vector.<Geometry> = new Vector.<Geometry>;
geom.push(ring_1);
var polygon:Polygon = new Polygon(geom);
var polygonFeature:PolygonFeature = new PolygonFeature(polygon,
                                          {featureName:"MyPolygon"});
featureLayer.addFeature(polygonFeature);
```

Polygon_Overlay.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:os="http://openscales.org"
               minWidth="955" minHeight="600"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="800"
        height="600"
        zoom="15"
        center="6.64282, 46.79092"
        x="100"
        y="100">
        <os:Mapnik
            name="base"/>
        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>
```

```
        <os:DragHandler/>
        <os:WheelHandler/>
    </os:Map>

    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            import org.openscales.core.feature.PolygonFeature;
            import org.openscales.core.layer.FeatureLayer;
            import org.openscales.core.popup.Anchored;
            import org.openscales.core.style.Style;
            import org.openscales.geometry.Geometry;
            import org.openscales.geometry.LinearRing;
            import org.openscales.geometry.Polygon;
            import org.openscales.proj4as.ProjProjection;

            [Bindable] private var map:Map = null;

            private var currentPopup:Anchored;

            private function initMap():void
            {
                map = fxmap.map;

                var featureLayer:FeatureLayer = new
                            FeatureLayer("PolygonFeatures");
                featureLayer.projection = new
                            ProjProjection("EPSG:4326");
                featureLayer.style = Style.getDefaultSurfaceStyle();
                map.addLayer(featureLayer);

                // Création du vecteur du LinearRing
                var vector_1:Vector.<Number>=new Vector.<Number>;
                vector_1.push(6.65591);    //P1[x]
                vector_1.push(46.78566);   //P1[y]
                vector_1.push(6.64922);    //P2[x]
                vector_1.push(46.78637);   //P2[y]
                vector_1.push(6.64046);    //P3[x]
                vector_1.push(46.79113);   //P3[y]
                vector_1.push(6.64282);    //P4[x]
                vector_1.push(46.79092);   //P4[y]

                // Création du Polygon
                var ring_1:LinearRing = new LinearRing(vector_1);
                var geom:Vector.<Geometry> = new Vector.<Geometry>;
                geom.push(ring_1);
                var polygon:Polygon = new Polygon(geom);
                var polygonFeature:PolygonFeature = new
                       PolygonFeature(polygon, {featureName:"MyPolygon"});
                featureLayer.addFeature(polygonFeature);
            }
        ]]>
    </fx:Script>

</s:Application>
```

9.4.1 Résultat

9.5 Interaction avec les objets

9.5.1 On Click

Nous allons à présent ajouter une info bulle lors du clic sur un objet. Pour ce faire, il est nécessaire d'ajouter un EventListener « CLICK » sur la couche de features s'occupant d'appeler la méthode showPopup( ).

```
featureLayer.addEventListener(MouseEvent.CLICK, showPopup);
```

```
private function showPopup(e:FeatureEvent):void
{
    // Supprimer l'éventuel popup ouvert
    if (currentPopup != null)
    {
        fxmap.map.removePopup(currentPopup);
        currentPopup = null;
    }

    // Ajoute le nouveau popup
    var feature : Feature = e.feature;
    currentPopup = new Anchored();
    currentPopup.htmlText = feature.data.featureName;
    currentPopup.size = new Size(measureText(currentPopup.htmlText).width
                                                            + 35, 30);
    currentPopup.feature = feature;
    fxmap.map.addPopup(currentPopup, true);
}
```

La notation `feature.data` permet d'accéder aux attributs de la feature. En effet, les attributs sont rattachés à un objet géographique grâce à l'utilisation du second paramètre des constructeurs createPointFeature, PolygonFeature, LineStringFeature. Par exemple:

```
PointFeature.createPointFeature(…,{featureName:"Point 1"});
```

On reconnait ici la syntaxe de déclaration d'un objet litéral pour définir les attributs de géométrie.

Les sources sont disponibles dans le fichier `PointsAndMarkers_Overlay_click.mxml`

9.5.2 On Over

Il est possible d'ajouter l'info bulle lorsque la souris est placée sur un objet. Pour ce faire, ajouter les EventListener suivants :

```
featureLayer.addEventListener(MouseEvent.MOUSE_OVER, addPopup);
featureLayer.addEventListener(MouseEvent.MOUSE_OUT, removePopup);
```

```
private function addPopup(e:FeatureEvent):void
{
    // Ajoute le nouveau popup
    var feature : Feature = e.feature;
    currentPopup = new Anchored();
    currentPopup.htmlText = feature.data.featureName;
    currentPopup.size = new Size(measureText(currentPopup.htmlText).width
                                                            + 35, 30);
    currentPopup.closeBox = false;
    currentPopup.feature = feature;
    fxmap.map.addPopup(currentPopup, true);
}
```

```
private function removePopup(e:FeatureEvent):void
{
    // Supprimer l'éventuel popup ouvert
    if (currentPopup != null)
    {
        fxmap.map.removePopup(currentPopup);
        currentPopup = null;
    }
}
```

Il en va de même pour l'interaction sur des lignes et des polygones.

Les sources sont disponibles dans le fichier `PointsAndMarkers_Overlay_over.mxml`

9.6 KML Features

Jusqu'à présent, nous avons créé des features « manuellement ». Il est possible de récupérer une couche de features provenant d'un fichier KML. Ce standard OGC est basé sur le formalisme XML et permet de décrire des objets géographiques ainsi que leur symbologie.

Pour en savoir plus sur le format KML (Keyhole Markup Language):

```
http://en.wikipedia.org/wiki/Keyhole_Markup_Language
```

Dans notre exemple, nous allons charger deux couches de features KML :

Limite administrative de la Haute-Savoie

```
<os:KML name="Haute-Savoie" url="http://ogo.heig-vd.ch/ria/data/74.kml"/>
```

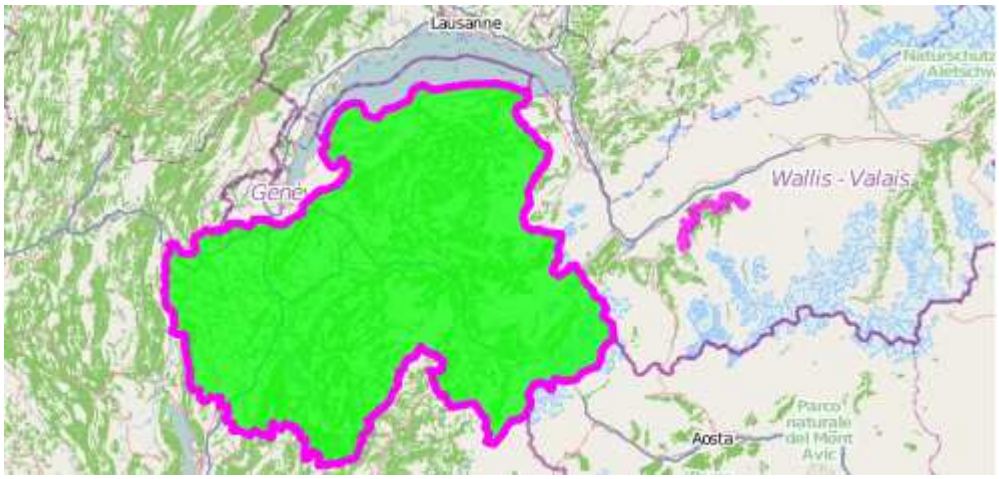
Trace GPS du Grand-Raid

```
<os:KML name="Grand Raid" url="http://ogo.heig-vd.ch/ria/data/utgtrack-
819.kml"/>
```

Kml_Overlay.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:os="http://openscales.org"
               minWidth="955" minHeight="600">
    <s:VGroup>
        <os:Map id="fxmap"
                width="500"
                height="600"
                x="60"
                y="50"
                zoom="9" center="6.8,46">
            <os:Mapnik name="base"/>
            <os:KML
                name="Haute-Savoie"
                url="http://ogo.heig-vd.ch/ria/data/74.kml"/>
            <os:KML
                name="Grand Raid"
                url="http://ogo.heig-vd.ch/ria/data/utgtrack-819.kml"/>
            <os:MousePosition
                x="10" y="{fxmap.height-20}"
                displayProjection="EPSG:4326"/>
            <os:DragHandler/>
            <os:WheelHandler/>
            <os:ControlPanel
                title="Layers Manager"
                x="510" y="10" width="215">
                <os:LayerManager/>
            </os:ControlPanel>
        </os:Map>
    </s:VGroup>
</s:Application>
```

9.6.1 Résultat



9.7 Exercice

Mettre en place une application cartographique qui localise des boulangeries suisses. Les données proviennent d'un flux GML `http://ogo.heig-vd.ch/ria/data/boulangeries.xml`. L'objectif est d'afficher un point pour chacune des boulangeries du flux GML. De plus, on désire afficher le nom de la boulangerie dans un pop-up lorsque l'utilisateur clique sur un point.

Pour en savoir plus sur le format GML (Geography Markup Language):

```
http://en.wikipedia.org/wiki/Geography_Markup_Language
```

Parsing XML e4x avec namespaces

```
http://www.darronschall.com/weblog/2006/04/using-xml-namespaces-with-e4x-and-actionscript-3.cfm
```

10 Styling de features

Jusqu'à présent, nous avons utilisé le rendu par defaut des objets géographiques :

- `Style.getDefaultPointStyle();`
- `Style.getDefaultLineStyle();`
- `Style.getDefaultSurfaceStyle();`

Or il est possible de personnaliser les styles qui gérent la représentation cartographique des objets géographiques.

Voyons comment personnaliser les points d'un objet géographique à l'aide du PointSymbolizer associé à `WellknownMarker` et `DisplayObjectMarker`.

10.1 WellknownMarker

Tout comme dans le chapitre précédent, il est tout d'abord nécessaire d'ajouter une nouvelle couche de Features à la carte. Cependant, on ne va pas utiliser le style par défaut, mais un style personnalisé.

```
featureLayer = new FeatureLayer("Features");
featureLayer.projection = new ProjProjection("EPSG:4326");
featureLayer.style = createStyle();
```

Puis nous ajoutons bien évidemment un certain nombre de points à cette couche de manière à pouvoir tester notre style personnalisé.

```
var point_1:PointFeature = PointFeature.createPointFeature(new
                Location(6.65591, 46.78566), {featureName:"Point 1"});
featureLayer.addFeature(point_1);

…
```

C'est la création du style personnalisé qui nous intéresse plus particulièrement dans cet exemple (méthode `createStyle`) :

```
private function createStyle():Style
{
    var marker:WellKnownMarker = new
                    WellKnownMarker(WellKnownMarker.WKN_TRIANGLE);
    marker.fill = new SolidFill(0xFF0000, .6);
    marker.stroke = new Stroke(0x233321, 2);
    marker.size = 24;

    var symbolizers:Vector.<Symbolizer> = new Vector.<Symbolizer>;
    symbolizers.push(new PointSymbolizer(marker));

    var rule:Rule = new Rule();
    rule.symbolizers = symbolizers;

    var rules:Vector.<Rule> = new Vector.<Rule>;
    rules.push(rule);

    var style:Style = new Style();
```

```
        style.rules = rules;

        return style;
    }
```

Le `PointSymbolizer` est associé à un marqueur, ici le `WellknownMarker`. Celui-ci permet d'utiliser des marqueurs simples tels que (cercle, croix, carré, étoile ou triangle) tout en précisant des paramètres (couleur de fond, couleur de bordure). Il est aussi nécessaire d'indiquer la taille du symbole.

Le style retourné par notre fonction possède une seule règle, avec uniquement le `PointSymbolizer` décrit ci-dessus. (c.f 10.3 pour une utilisation avancée des règles)

Symbolizer_WellknownMarker.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:os="http://openscales.org"
               minWidth="955" minHeight="600"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="800"
        height="600"
        zoom="15"
        center="6.64282, 46.79092"
        x="100"
        y="100">
        <os:Mapnik
            name="base"/>
        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>
        <os:DragHandler/>
        <os:ClickHandler/>
        <os:WheelHandler/>
    </os:Map>

    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[

            import org.openscales.core.Map;
            import org.openscales.core.feature.PointFeature;
            import org.openscales.core.layer.FeatureLayer;
            import org.openscales.core.popup.Anchored;
            import org.openscales.core.style.Rule;
            import org.openscales.core.style.Style;
            import org.openscales.core.style.fill.SolidFill;
            import org.openscales.core.style.marker.WellKnownMarker;
            import org.openscales.core.style.stroke.Stroke;
            import org.openscales.core.style.symbolizer.PointSymbolizer;
            import org.openscales.core.style.symbolizer.Symbolizer;
            import org.openscales.geometry.basetypes.Location;
```

```
import org.openscales.proj4as.ProjProjection;

[Bindable] private var map:Map = null;

private var featureLayer:FeatureLayer;

private var currentPopup:Anchored;

private function initMap():void
{
    map = fxmap.map;
    featureLayer = new FeatureLayer("Features");
    featureLayer.projection = new
                        ProjProjection("EPSG:4326");
    featureLayer.style = createStyle();

    // Ajout du premier PointFeature
    var point_1:PointFeature =
                PointFeature.createPointFeature(
                      new Location(6.65591, 46.78566),
                      {featureName:"Point 1"});
    featureLayer.addFeature(point_1);

    // Ajout du deuxième PointFeature
    var point_2:PointFeature =
                PointFeature.createPointFeature(
                      new Location(6.64046, 46.79113),
                      {featureName:"Point 2"});
    featureLayer.addFeature(point_2);

    // Ajout du troisième PointFeature
    var point_3:PointFeature =
                PointFeature.createPointFeature(
                new Location(6.65046, 46.78113),
                {featureName:"Point 3"});
    featureLayer.addFeature(point_3);

    // Ajout du quatrième PointFeature
    var point_4:PointFeature =
                PointFeature.createPointFeature(
                new Location(6.64046, 46.77113),
                {featureName:"Point 4"});
    featureLayer.addFeature(point_4);

    map.addLayer(featureLayer);
}

// Création d'un "custom point symbolizer"
private function createStyle():Style
{
    // Création du style
    var marker:WellKnownMarker = new
          WellKnownMarker(WellKnownMarker.WKN_TRIANGLE);
    marker.fill = new SolidFill(0xFF0000, .6);
    marker.stroke = new Stroke(0x233321, 2);
    marker.size = 24;

    var symbolizers:Vector.<Symbolizer> = new
                        Vector.<Symbolizer>;
    symbolizers.push(new PointSymbolizer(marker));

    var rule:Rule = new Rule();
    rule.symbolizers = symbolizers;

    var rules:Vector.<Rule> = new Vector.<Rule>;
```

```
                rules.push(rule);

                var style:Style = new Style();
                style.rules = rules;

                return style;
            }

        ]]>
    </fx:Script>
</s:Application>
```

10.1.1 Résultat

10.2 DisplayObjectMarker

Dans le chapitre précédent, nous avons utilisé des marqueurs prédéfinis dont on peut ajuster les aspects graphiques (couleur, épaisseur). Il existe un second type de marqueur qui rend possible d'utiliser une image comme marqueur, c'est le rôle du `DisplayObjectMarker`.

Tout comme le chapitre précédent, il est nécessaire d'ajouter une nouvelle couche de Features à la carte. Cependant, on ne va pas utiliser le style par défaut, mais un style personnalisé.

```
featureLayer = new FeatureLayer("Features");
featureLayer.projection = new ProjProjection("EPSG:4326");
featureLayer.style = createStyle();
```

Puis nous ajoutons bien évidemment un certain nombre de points à cette couche de manière à pouvoir tester notre style personnalisé.

```
var point_1:PointFeature = PointFeature.createPointFeature(new
                Location(6.65591, 46.78566), {featureName:"Point 1"});
featureLayer.addFeature(point_1);

…
```

La création du style personnalisé basé sur `DisplayObjectMarker` nous intéresse plus particulièrement ici (méthode `createStyle`) :

```
private function createStyle():Style
{
    [Embed(source="/tutorial/styling/assets/add.png")] var _image:Class;
    var marker:DisplayObjectMarker = new DisplayObjectMarker(_image);

    var symbolizer:PointSymbolizer = new PointSymbolizer(marker);

    var symbolizers:Vector.<Symbolizer> = new Vector.<Symbolizer>;
    symbolizers.push(symbolizer);

    var rule:Rule = new Rule();
    rule.symbolizers = symbolizers;

    var rules:Vector.<Rule> = new Vector.<Rule>;
    rules.push(rule);

    var style:Style = new Style();
    style.rules = rules;

    return style;
}
```

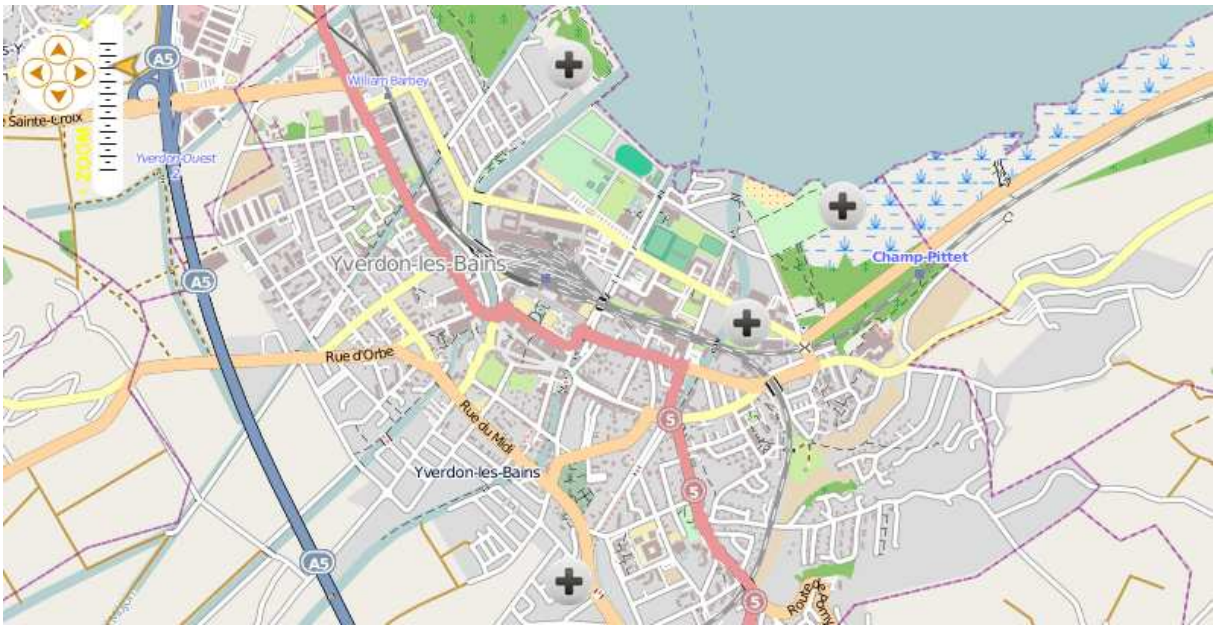
Le `PointSymbolizer` repose sur l'objet graphique `DisplayObjectMarker` qui permet d'utiliser des marqueurs personnalisés basés sur des images.

Symbolizer_DisplayObjectMarker.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:os="http://openscales.org"
               minWidth="955" minHeight="600"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="800"
        height="600"
        zoom="15"
        center="6.64282, 46.79092"
        x="100"
        y="100">
        <os:Mapnik
            name="base"/>
        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>
        <os:DragHandler/>
        <os:ClickHandler/>
        <os:WheelHandler/>
    </os:Map>

    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            import org.openscales.core.feature.PointFeature;
            import org.openscales.core.layer.FeatureLayer;
            import org.openscales.core.popup.Anchored;
            import org.openscales.core.style.Rule;
            import org.openscales.core.style.Style;
            import org.openscales.core.style.fill.SolidFill;
            import org.openscales.core.style.marker.DisplayObjectMarker;
            import org.openscales.core.style.stroke.Stroke;
            import org.openscales.core.style.symbolizer.PointSymbolizer;
            import org.openscales.core.style.symbolizer.Symbolizer;
            import org.openscales.geometry.basetypes.Location;
            import org.openscales.proj4as.ProjProjection;

            [Bindable] private var map:Map = null;

            private var featureLayer:FeatureLayer;

            private var currentPopup:Anchored;

            private function initMap():void
            {
                map = fxmap.map;
                featureLayer = new FeatureLayer("MyFeatures");
                featureLayer.projection = new
                                    ProjProjection("EPSG:4326");
                featureLayer.generateResolutions(19);
                featureLayer.style = createStyle();

                // Ajout du premier PointFeature
```

```
                var point_1:PointFeature =
                        PointFeature.createPointFeature(
                                new Location(6.65591, 46.78566),
                                {featureName:"Point 1"});
                featureLayer.addFeature(point_1);

                // Ajout du deuxième PointFeature
                var point_2:PointFeature =
                        PointFeature.createPointFeature(
                                new Location(6.64046, 46.79113),
                                {featureName:"Point 2"});
                featureLayer.addFeature(point_2);

                // Ajout du troisième PointFeature
                var point_3:PointFeature =
                        PointFeature.createPointFeature(
                                new Location(6.65046, 46.78113),
                                {featureName:"Point 3"});
                featureLayer.addFeature(point_3);

                // Ajout du quatrième PointFeature
                var point_4:PointFeature =
                        PointFeature.createPointFeature(new
                                Location(6.64046, 46.77113),
                                {featureName:"Point 4"});
                featureLayer.addFeature(point_4);

                map.addLayer(featureLayer);
            }

            // Création d'un "custom point symbolizer"
            private function createStyle():Style
            {
                [Embed(source="/tutorial/styling/assets/add.png")]
                var _image:Class;
                var marker:DisplayObjectMarker = new
                                        DisplayObjectMarker(_image);

                var symbolizer:PointSymbolizer = new
                                        PointSymbolizer(marker);

                var symbolizers:Vector.<Symbolizer> =
                                                new Vector.<Symbolizer>;
                symbolizers.push(symbolizer);

                var rule:Rule = new Rule();
                rule.symbolizers = symbolizers;

                var rules:Vector.<Rule> = new Vector.<Rule>;
                rules.push(rule);

                var style:Style = new Style();
                style.rules = rules;

                return style;
            }

        ]]>
    </fx:Script>

</s:Application>
```

10.2.1 Résultat



10.3 Filtres

Dans les deux chapitres précédents nous avons utilisé une seule règle de symbolisation, celle-ci s'appliquant à tous les objets géographiques de la couche. Néanmoins, il est possible d'appliquer un filtre à une règle permettant ainsi que la symbolisation ne s'applique qu'à une partie des objets géographiques, c'est-à-dire ceux qui « rentrent » dans le filtre.

Un filtre évalue les caractéristiques de l'objet géographique, soit sa géométrie, soit ses attributs. Voyons ci-dessous un exemple qui applique un filtrage sur un attribut.

Tout comme le chapitre précédent, il est nécessaire d'ajouter une nouvelle couche de Features à la carte. Cependant, on ne va pas utiliser le style par défaut, mais un style personnalisé.

```
featureLayer = new FeatureLayer("Features");
featureLayer.projection = new ProjProjection("EPSG:4326");
featureLayer.style = createStyle();
```

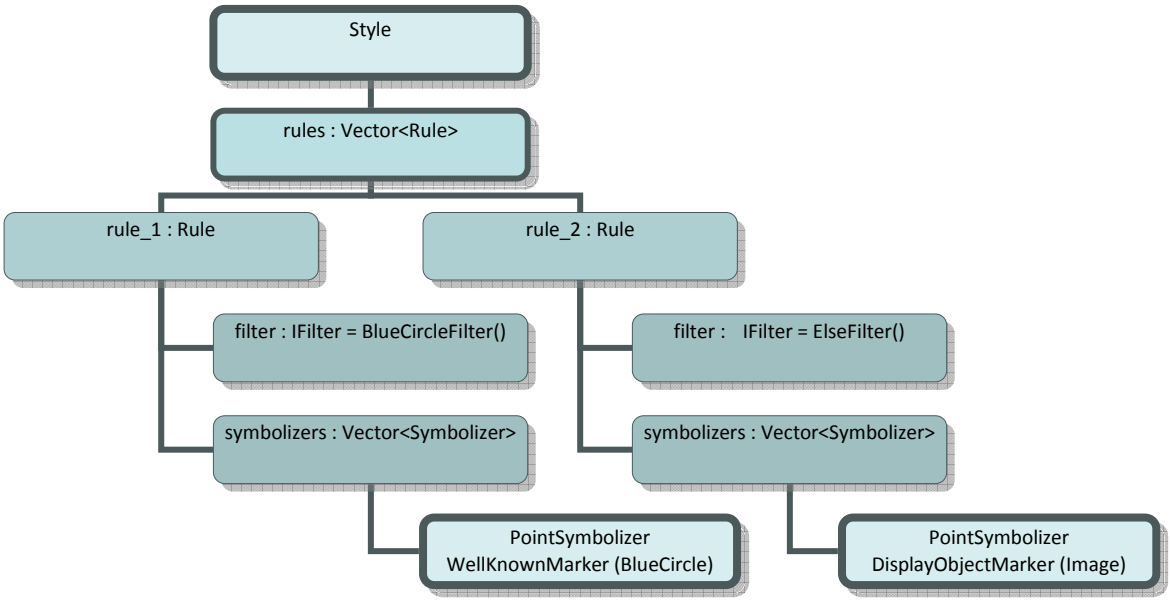
Puis nous ajoutons bien évidemment un certain nombre de points à cette couche de manière à pouvoir tester notre style personnalisé.

```
var point_1:PointFeature = PointFeature.createPointFeature(
                                    new Location(6.65591, 46.78566),
                                    {bluecircle:"true"});
featureLayer.addFeature(point_1);
…
```

Nous utilisons un attribut `bluecircle` définit à « true » ou « false » permettant de modifier la manière dont va être représenté l'objet. C'est cette valeur qui va être évaluée par le filtre.

En effet, le style est composé de deux règles (la première représentant des features sous forme d'un cercle bleu, la deuxième utilisant une image personnalisée).

C'est ici qu'intervient la notion de filtre permettant de déterminer quelle est la règle à appliquer pour représenter un objet. Dans notre cas, le choix de la règle se base sur la valeur d'un attribut (`bluecircle`).




```
// Création d'un "custom point symbolizer"
private function createStyle():Style
{
    // Paramètres du point
    var blue_fill:SolidFill = new SolidFill(0x0000FF, .6);
    var stroke:Stroke = new Stroke(0x233321, 2)

    //
    // Points en bleu
    //
    var marker_1:WellKnownMarker = new
                                WellKnownMarker(WellKnownMarker.WKN_CIRCLE);
    marker_1.fill = blue_fill;
    marker_1.stroke = stroke;
    marker_1.size = 24;

    var symbolizer_1:PointSymbolizer = new PointSymbolizer(marker_1);

    var symbolizers_1:Vector.<Symbolizer> = new Vector.<Symbolizer>;
    symbolizers_1.push(symbolizer_1);

    var rule_1:Rule = new Rule();
    rule_1.symbolizers = symbolizers_1;
    rule_1.filter = new BlueCircleFilter();
```

```
        //
        // Custom Image (Default)
        //
        [Embed(source="/tutorial/styling/assets/add.png")] var _image:Class;
        var marker_2:DisplayObjectMarker = new DisplayObjectMarker(_image);

        var symbolizer_2:PointSymbolizer = new PointSymbolizer(marker_2);

        var symbolizers_2:Vector.<Symbolizer> = new Vector.<Symbolizer>;
        symbolizers_2.push(symbolizer_2);

        var rule_2:Rule = new Rule();
        rule_2.symbolizers = symbolizers_2;
        rule_2.filter = new ElseFilter();

        //
        // Ajout des règles
        //
        var rules:Vector.<Rule> = new Vector.<Rule>;
        rules.push(rule_1);
        rules.push(rule_2);

        var style:Style = new Style();
        style.rules = rules;

        return style;
    }
```

La classe `BlueCircleFilter` nous intéresse plus particulièrement. En effet, celle-ci implémente l'interface `IFilter` et doit donc posséder une méthode `matches(features:Feature):Boolean`.

BlueCircleFilter.as

```
package tutorial.styling.filters
{
    import org.openscales.core.feature.Feature;
    import org.openscales.core.filter.IFilter;

    public class BlueCircleFilter implements IFilter
    {
        public function matches(feature:Feature) : Boolean
        {
            return feature.data.bluecircle == "true";
        }
    }
}
```

Dans notre cas, cette méthode est utilisée pour filtrer les objets qui rentrent dans la première règle de symbolisation contrôlant si l'objet possède l'attribut bluecircle à « true ».

Aussi, on introduit une seconde règle avec un filtre spécifique ElseFilter permettant de définir la symbolisation à appliquer à tous les objets qui n'entrent dans aucun autre filtre.

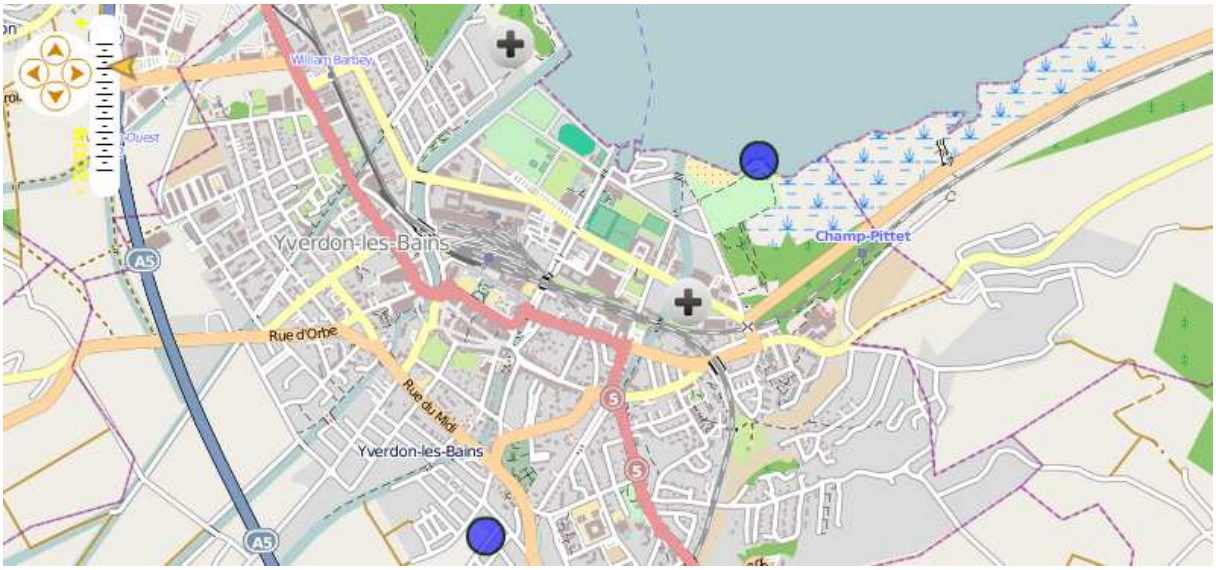
Symbolizer_Filters.mxml

```
<?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:os="http://openscales.org"
               minWidth="955" minHeight="600"
               creationComplete="initMap()">
    <os:Map
        id="fxmap"
        width="800"
        height="600"
        zoom="15"
        center="6.64282, 46.79092"
        x="100"
        y="100">
        <os:Mapnik
            name="base"/>
        <os:MousePosition
            x="10"
            y="{fxmap.height-20}"
            displayProjection="EPSG:4326"/>
        <os:DragHandler/>
        <os:ClickHandler/>
        <os:WheelHandler/>
    </os:Map>

    <os:PanZoom
        map="{map}"
        x="{fxmap.x+10}"
        y="{fxmap.y+10}"/>

    <fx:Script>
        <![CDATA[
            import org.openscales.core.Map;
            import org.openscales.core.feature.PointFeature;
            import org.openscales.core.filter.ElseFilter;
            import org.openscales.core.layer.FeatureLayer;
            import org.openscales.core.popup.Anchored;
            import org.openscales.core.style.Rule;
            import org.openscales.core.style.Style;
            import org.openscales.core.style.fill.SolidFill;
            import org.openscales.core.style.marker.DisplayObjectMarker;
            import org.openscales.core.style.marker.WellKnownMarker;
            import org.openscales.core.style.stroke.Stroke;
            import org.openscales.core.style.symbolizer.PointSymbolizer;
            import org.openscales.core.style.symbolizer.Symbolizer;
            import org.openscales.geometry.basetypes.Location;
            import org.openscales.proj4as.ProjProjection;

            import tutorial.styling.filters.BlueCircleFilter;

            [Bindable] private var map:Map = null;

            private var featureLayer:FeatureLayer;

            private var currentPopup:Anchored;

            private function initMap():void
            {
                map = fxmap.map;
                featureLayer = new FeatureLayer("MyFeatures");
                featureLayer.projection = new
                                    ProjProjection("EPSG:4326");
```

```
            featureLayer.generateResolutions(19);
            featureLayer.style = createStyle();

            // Ajout du premier PointFeature
            var point_1:PointFeature =
                        PointFeature.createPointFeature(
                              new Location(6.65591, 46.78566),
                              {bluecircle:"true"});
            featureLayer.addFeature(point_1);

            // Ajout du deuxième PointFeature
            var point_2:PointFeature =
                        PointFeature.createPointFeature(
                              new Location(6.64046, 46.79113),
                              {bluecircle:"false"});
            featureLayer.addFeature(point_2);

            // Ajout du troisième PointFeature
            var point_3:PointFeature =
                        PointFeature.createPointFeature(
                              new Location(6.65046, 46.78113),
                              {bluecircle:"false"});
            featureLayer.addFeature(point_3);

            // Ajout du quatrième PointFeature
            var point_4:PointFeature =
                        PointFeature.createPointFeature(
                              new Location(6.64046, 46.77113),
                              {bluecircle:"true"});
            featureLayer.addFeature(point_4);

            map.addLayer(featureLayer);
      }

      // Création d'un "custom point symbolizer"
      private function createStyle():Style
      {
            // Paramètres du point
            var blue_fill:SolidFill = new SolidFill(0x0000FF, .6);
            var stroke:Stroke = new Stroke(0x233321, 2)

            //
            // Points en bleu
            //
            var marker_1:WellKnownMarker = new
                        WellKnownMarker(WellKnownMarker.WKN_CIRCLE);
            marker_1.fill = blue_fill;
            marker_1.stroke = stroke;
            marker_1.size = 24;

            var symbolizer_1:PointSymbolizer =
                        new PointSymbolizer(marker_1);

            var symbolizers_1:Vector.<Symbolizer> =
                        new Vector.<Symbolizer>;
            symbolizers_1.push(symbolizer_1);

            var rule_1:Rule = new Rule();
            rule_1.symbolizers = symbolizers_1;
            rule_1.filter = new BlueCircleFilter();

            //
            // Custom Image (Default)
            //
            [Embed(source="/tutorial/styling/assets/add.png")]
```

```
                var _image:Class;
                var marker_2:DisplayObjectMarker =
                                new DisplayObjectMarker(_image);

                var symbolizer_2:PointSymbolizer =
                                new PointSymbolizer(marker_2);

                var symbolizers_2:Vector.<Symbolizer> =
                                new Vector.<Symbolizer>;
                symbolizers_2.push(symbolizer_2);

                var rule_2:Rule = new Rule();
                rule_2.symbolizers = symbolizers_2;
                rule_2.filter = new ElseFilter();

                //
                // Ajout des règles
                //
                var rules:Vector.<Rule> = new Vector.<Rule>;
                rules.push(rule_1);
                rules.push(rule_2);

                var style:Style = new Style();
                style.rules = rules;

                return style;
            }

        ]]>
    </fx:Script>
</s:Application>
```

10.3.1 Résulat

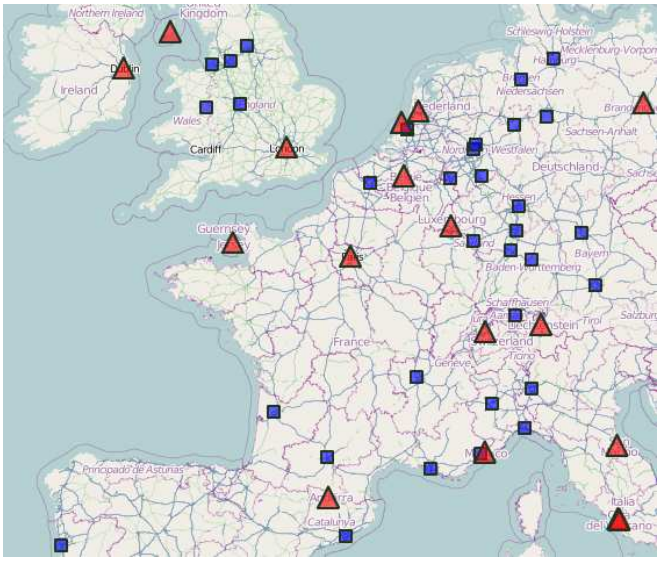


Les exemples présents dans les sources (`UseLineSymbolizer.mxml` et `UsePolygonSymbolizer.mxml`) illustrent l'utilisation des symbolizers `LineSymbolizer` et `PolygonSymbolizer`.

10.4 Exercices

Pour ces deux exercices, vous devez charger une couche d'objets géographique points à partir du flux GeoJSON suivant : http://ogo.heig-vd.ch/ria/data/cities.json

Appliquez un style personnalisé qui permet de représenter les villes comme ci-dessous : les capitales sont des triangles rouges et les autres villes des carrés bleus (remarque : analysez le flux JSON, il contient une propriété CAPIT_0_1 bien utile).



Représentez les villes de sorte qu'une symbologie différente leur soit appliqué selon appartenance à un des cadrants (remarque : il y a quatre cadrants, Nord-Ouest, Nord-Est, Sud-Est, Sud-Ouest).