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"/>`

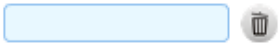
Data : Ce type de liaison permet à la classe d'habillage d'accéder à des propriétés publiques du composant hôte.

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.

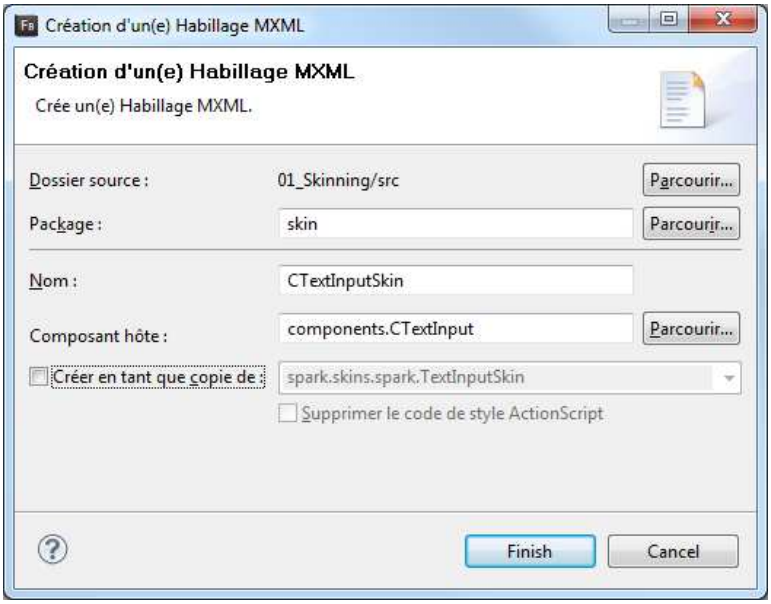
Ensuite, il est nécessaire de traiter le click sur le bouton. Pour ce faire, ajouter les méthodes suivantes :

```
// 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 = "";
        }
    }
}
```

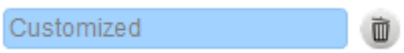
skin.CTextInputSkinCSS.mxml

```
<?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

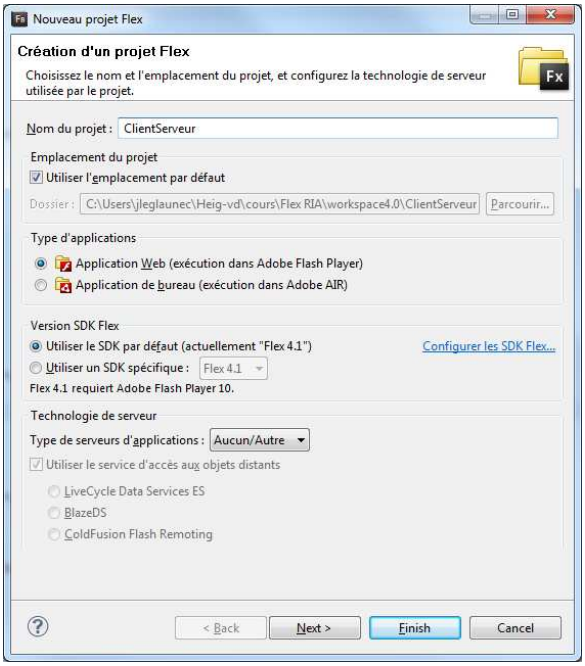
Pour commencer, le fichier **users.xml** doit être déposé dans le dossier ***{HTDOCS} /clientserveur*** du serveur et accessible par une URL semblable à ***{SERVER URL} /clientserveur/users.xml***.

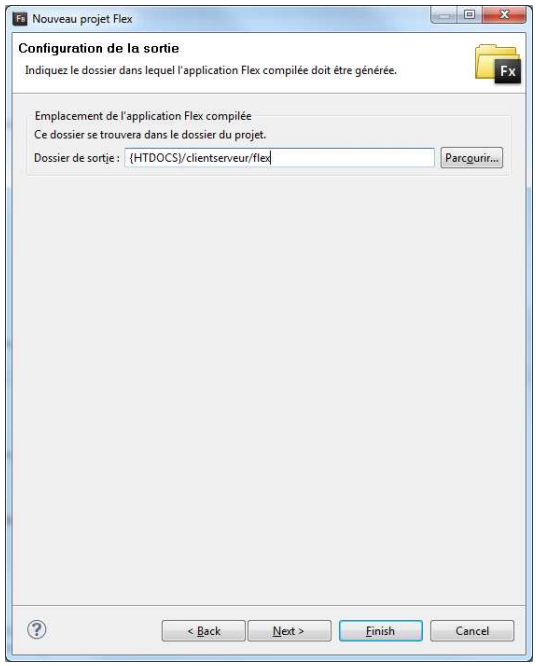
users.xml

```
<?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

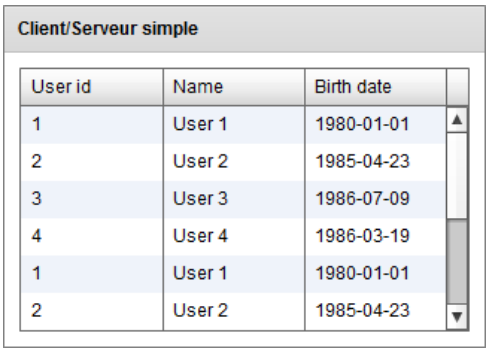
```
<mx:DataGrid
    id="dgdUsers"
    dataProvider="{remoteXmlFile.user}"
    width="100%">
    <mx:columns>
        <mx:DataGridColumn
            dataField="@id"
            headerText="User id"/>
        <mx:DataGridColumn
            dataField="name"
            headerText="Name"/>
        <mx:DataGridColumn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