FRONTAL

November 2011

Menu

INTRODUCTION TO FRONTAL

WHAT IS FRONTAL?

Frontal is an ActionScript 3 project capable of create RIA flex-based applications from pure html code; Frontal is in effect an **html2flex** library.

It has the goal to be the bridge between standard html projects and flex projects.

WHY FRONTAL?

Frontal gives a developer the chance to build RIA apllications without the knowledge of a flex applicaton nature.

It's a swc library which could be integrated very easily into your own flex applications and your frontal application could only be a part of the entire project.

The Frontal project is also a portal project; it enables the possibility to build modular application in a very easy way; the documentation you are visiting is an example of a portal application with Frontal.

HOW WORKS FRONTAL?

A developer uses Frontal in the front-end part of his application; the server-side can be realized with any server language available.

The Flex project, which makes use of Frontal, receives and parses html response from the server; the Frontal swc then renders the content turning the html tags into flex sdk components.

A portal realized with Frontal, can be enriched with all the style informations of the flex components; the developer is able to apply this information directly in the html code, populating the values of the style attributes.

The interaction with the content is achieved writing ActionScript code into the html code, so taking advantage of all the potentialities of the flex framework, such as strong interactivity with users, effects, browser independency, high user experiences.

Frontal is compiled using the 3.4.1 Flex sdk.

TOPICS

SET UP

The integration of the swc Frontal library involves very few steps.

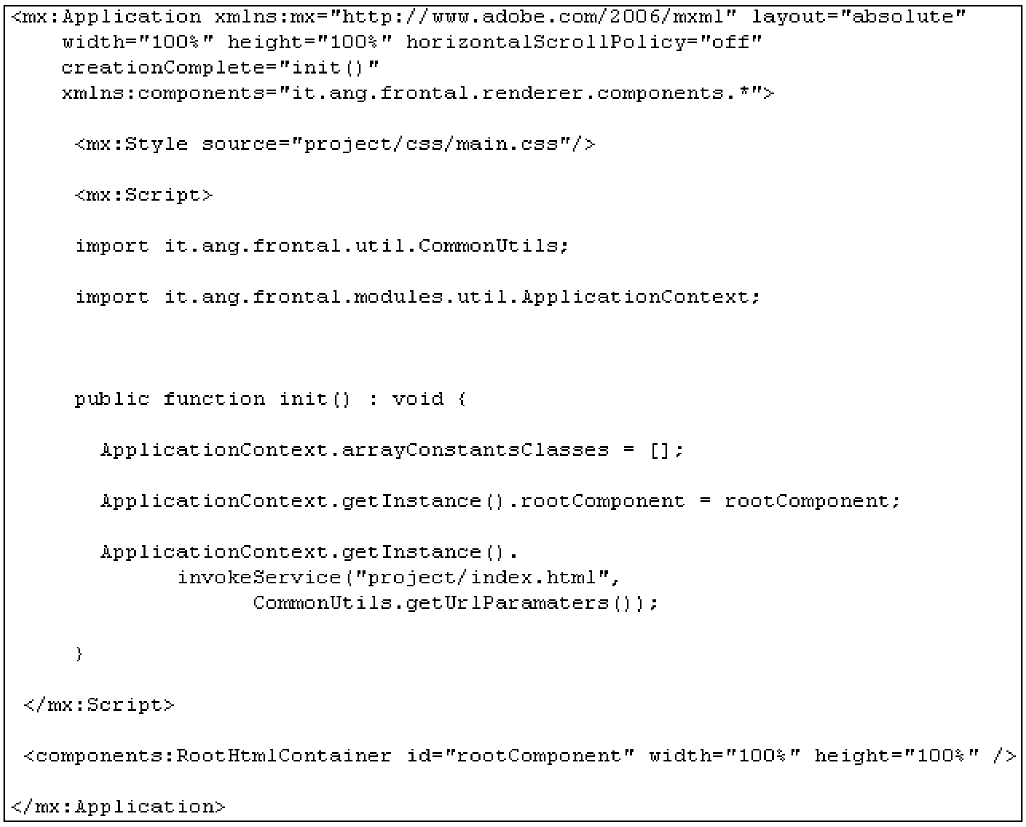
First of all you must include, into your flex project, the swc file (if you are using an ide such as Flex Builder or Flash Builder, you have to access to the Library Path panel of your application), then you have to include few code rows into the application file, as shown in the figure 1.

The visual Frontal component to be introduced is the *RootHtmlContainer* which represents the main container used by Frontal to inject the content; in order to start up the Frontal framework it's necessary to configure the Frontal instance object *ApplicationContext*; in the init function (invoked by the creationComplete event of the application), it's set the rootComponent property of *ApplicationContext* and then invoked the *invokeService* method.

The parameters to pass to the *ApplicationContext invokeService* method are the url of the service which gives back the html content, and eventually the request parameters in the url of the flex application; to retrieve these parameters there is the *getUrlParameters* method of the Frontal *CommonUtils* class which gives back a Dictionary object.

In the init method there is also an other property of the *ApplicationContext* class to be set, that is the *arrayConstantsClasses* property, an array of custom classes by which it is possible to extend the Frontal framework with custom tags; however it is discussed in the Customizing section.

Particulary attention must be paid to the widht and height properties of the Application object and of the *RootHtmlContainer* object, and to the declaration of the css files to include into the project.



```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
    width="100%" height="100%" horizontalScrollPolicy="off"
    creationComplete="init()"
    xmlns:components="it.ang.frontal.renderer.components.*">

     <mx:Style source="project/css/main.css"/>

     <mx:Script>

     import it.ang.frontal.util.CommonUtils;

     import it.ang.frontal.modules.util.ApplicationContext;



     public function init() : void {

       ApplicationContext.arrayConstantsClasses = [];

       ApplicationContext.getInstance().rootComponent = rootComponent;

       ApplicationContext.getInstance().
             invokeService("project/index.html",
                   CommonUtils.getUrlParamaters());

     }

 </mx:Script>

 <components:RootHtmlContainer id="rootComponent" width="100%" height="100%" />

</mx:Application>
```

Figure 1 – Set Up

PORTAL EXTENSION

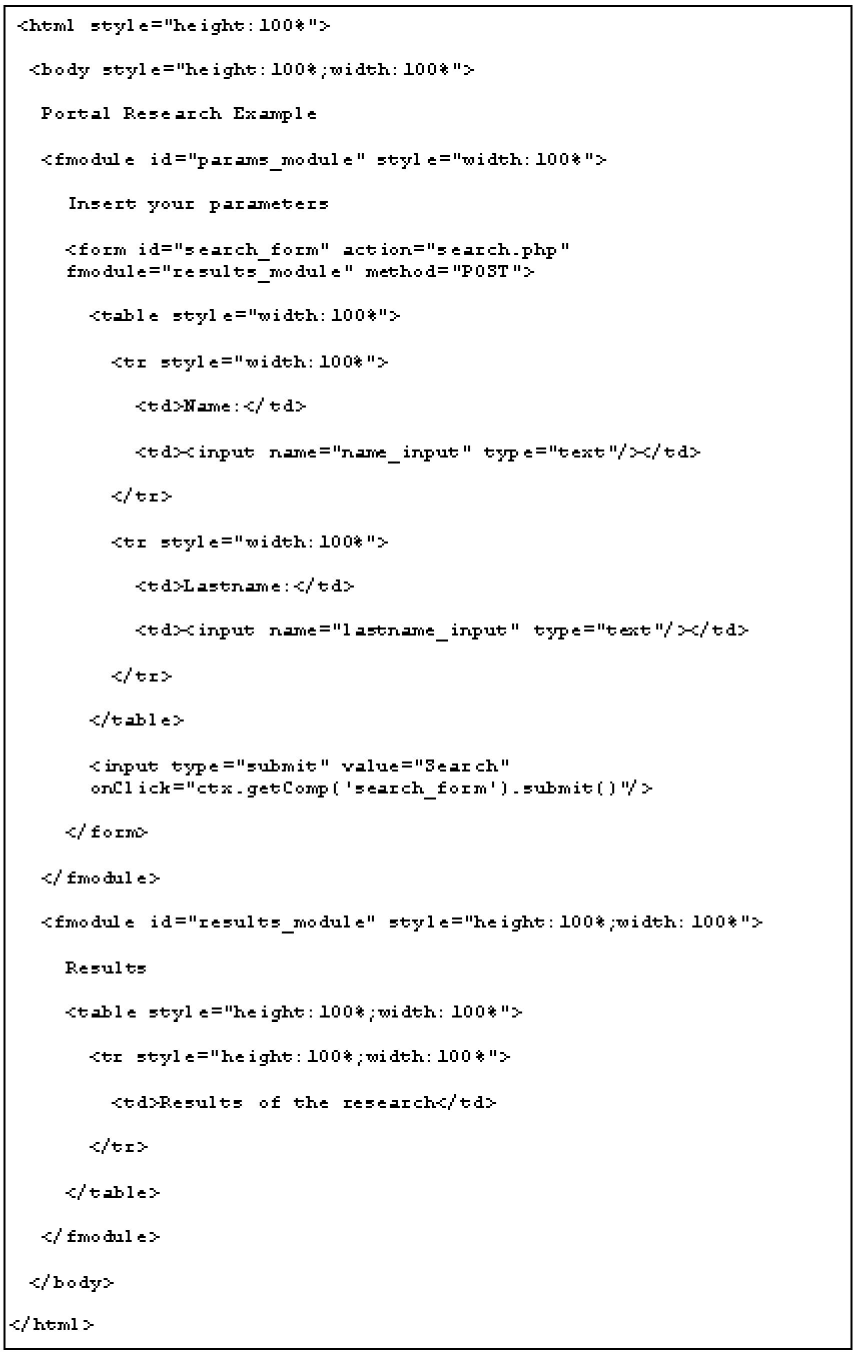
The Frontal library introduces an important behaviour in the development of your application, that is the portal\modular extension.

A Frontal module can be inserted everywhere in your code and it has its own lifecycle and a module context associated. To introduce a Frontal module it's necessary declare a *fmodule* tag with it's id attribute, mandatory in order to refer the module.

From an html point the *fmodule* tag has the same caratheristics as the *div* tag, and so it is a real container and it could be assigned all the attributes and style behaviours of the div tag.

The Frontal module gives the power to be the final point of a server interaction; the developer, both in the *form* and in the *a* tag, can declare what is the Frontal module that receives the server response, and it will be the only part of your application that will be refreshed; this is a clear portlet behaviour. Think about, for example, to have two modules, one for the filling of some particular criteria parameters in order to make a data research, and the other one that visualizes the results; the first one could be wrote in order to have, in the form tag, the module name of the second module, which receives and shows the results of the research.

In the figure 2 there is an example code of the modular extension.



```
<html style="height:100%">

 <body style="height:100%;width:100%">

  Portal Research Example

  <fmodule id="params_module" style="width:100%">

    Insert your parameters

    <form id="search_form" action="search.php"
    fmodule="results_module" method="POST">

      <table style="width:100%">

        <tr style="width:100%">

          <td>Name:</td>

          <td><input name="name_input" type="text"/></td>

        </tr>

        <tr style="width:100%">

          <td>Lastname:</td>

          <td><input name="lastname_input" type="text"/></td>

        </tr>

      </table>

      <input type="submit" value="Search"
      onClick="ctx.getComp('search_form').submit()"/>

    </form>

  </fmodule>

  <fmodule id="results_module" style="height:100%;width:100%">

    Results

    <table style="height:100%;width:100%">

      <tr style="height:100%;width:100%">

        <td>Results of the research</td>

      </tr>

    </table>

  </fmodule>

 </body>

</html>
```

Figure 2 – Portal Extension

Supported Tags

Here is the list of the tags supported from the Frontal library:

- html

It's the root tag. It is treated by Frontal like any other container, because of this it's a best pratice to assigne it the style attribute height, in order to have the desiderated behaviour from Frontal (the attribute width is always setted to 100%).

```
<html>
```

or

```
<html style="height:100%">
```

- body

It's the body html tag. Even if this tag isn't very important for Frontal, because it is treated like any other container, it is a best practice to insert it in order to give a well-formed project structure.

Because the html tag is converted by Frontal to a container, it's a best pratice to assigne it the style attribute height, in order to have the desiderated behaviour from Frontal (the attribute width is always setted to 100%).

```
<body>
```

or

```
<body style="height:100%">
```

- div

It's the most important and used tag.

It's converted by Frontal to a Flex Container Object; in order to have the desiderated behaviour from Frontal, it's a best pratice to assigne it the style dimension attributes.

This tag accepts every html style attribute, and it could be integrated with the flex associated object attributes.

<div>

or

<div style="width:200px;height:100%;overflow:hidden">

or

<div useHandCursor="true" buttonMode="true">

- p

It's the html p tag.

It's converted by Frontal to a Flex Container Object; in order to have the desiderated behaviour from Frontal, it's a best pratice to assigne it the style dimension attributes.

This tag accepts every html style attribute, and it could be integrated with the flex associated object attributes.

<p>

or

<p style="width:200px;height:100%;overflow:hidden">

or

<p useHandCursor="true" buttonMode="true">

- font

It's the html font tag.

- img

It's the img html tag for image visualization.

It's converted by Frontal to an Image Object.

- span

It's the span html tag.

It's converted by Frontal to a Container Object.

- a

It's the a html tag.

This tag gives the chance to navigate into the application based on its own attributes:

target: it's the target html attribute; both if it is setted to blank or '#' and it is absent, the swf is not refreshed and the users remains to interact with the current content;

href: it's the href html attribute and represents the url to which forward the request;

fmodule: it's a Frontal attribute of the a tag; in combination with the href attribute, it drives the response of the request to the indicated module, giving a portal behaviour.

- table

It's the table html tag.

It's converted by Frontal to a Grid Object.

It's able to manage tabulation properties, reading the 'align' and 'cellspacing' attributes.

- tr

It's the tr html tag.

It's converted by Frontal to a GridRow Object.

It's able to manage tabulation properties, reading the 'align' and 'valign' attributes.

- th

It's the tr html tag.

It's converted by Frontal to a GridItem Object with the style attributes 'fontWeight' setted to 'bold' and 'horizontalAlign' setted to 'center'.

- td

It's the td html tag.

It's converted by Frontal to a GridItem Object.

It's able to manage tabulation properties, reading the 'align', 'valign', 'colspan' and 'rowspan' attributes.

- input

It's the input html tag.

Frontal creats the associated gui object based on the 'type' attribute value:

button\submit: Button Object;

checkbox: CheckBox Object;

radio: RadioButton Object (in this case there is the attribute 'groupName' in order to define the radio button group);

hidden: Frontal creates a TextInput Object with the 'includeInLayout' and 'visible' properities setted fo false;

password: Frontal creates a TextInput Object with the 'displayAsPassword' property setted to true;

text: TextInput Object.

- form

It's the form html tag.

It has the same behaviour as a classical html form, except for the presence of the 'fmodule' attribute; this attribute drives the response of the request to the indicated module, giving a portal behaviour.

In order to turn on the form it's necessary to invoke explicity the 'submit()' method; this method can accept also two parameters which are the submit button name and the submit button value.

- select

It's the select html tag.

It's converted by Frontal to a ComboBox Object; in combination with the option tag, it can be populated with data.

- ul

It's the ul html tag.

It's converted by Frontal to a Container Object; it's able to generate an unordered list of items.

- li

It's the li html tag.

It's converted by Frontal to a Container Object; it represents a list item of an unordered list.

- b

It's the b html tag.

It's able to render bold text.

- strong

It's the strong html tag.

It's able to render bold text.

- br

It's the br html tag.

It's able to insert a single line break.

- button

It's the button html tag.

It's converted by Frontal to a Button Object.

- center

It's the center html tag.

It's converted by Frontal to a Container Object with the flex style property 'htmlHorizontalAlign' setted to center.

- i

It's the i html tag.

It's able to render italic text.

- h1 - h2 - h3 - h4 - h5 - h6

They are the h html tags.

These tags are used to define headings.

h1: heading 1;

h2: heading 2;

h3: heading 3;

h4: heading 4;

h5: heading 5;

h6: heading 6.

- small

It's the small html tag.

It's able to render as smaller text.

- big

It's the big html tag.

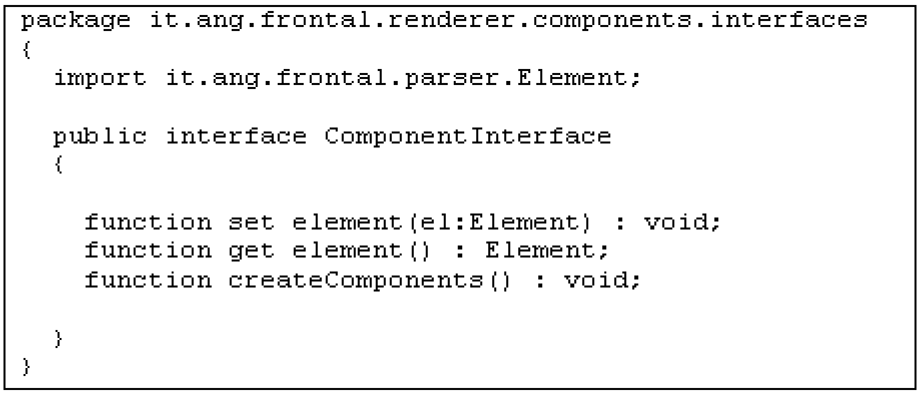
It's able to render as bigger text.

Custom Tags

Customizing tags is an entry point in the Frontal project to extend its functionalities and the objects reusing.

Creating custom tags requires a few steps.

First of all it's necessary to create, into your flex project, the Flex Component that will handle your custom tag; to achieve this your component must implement the Frontal interface *ComponentInterface*.



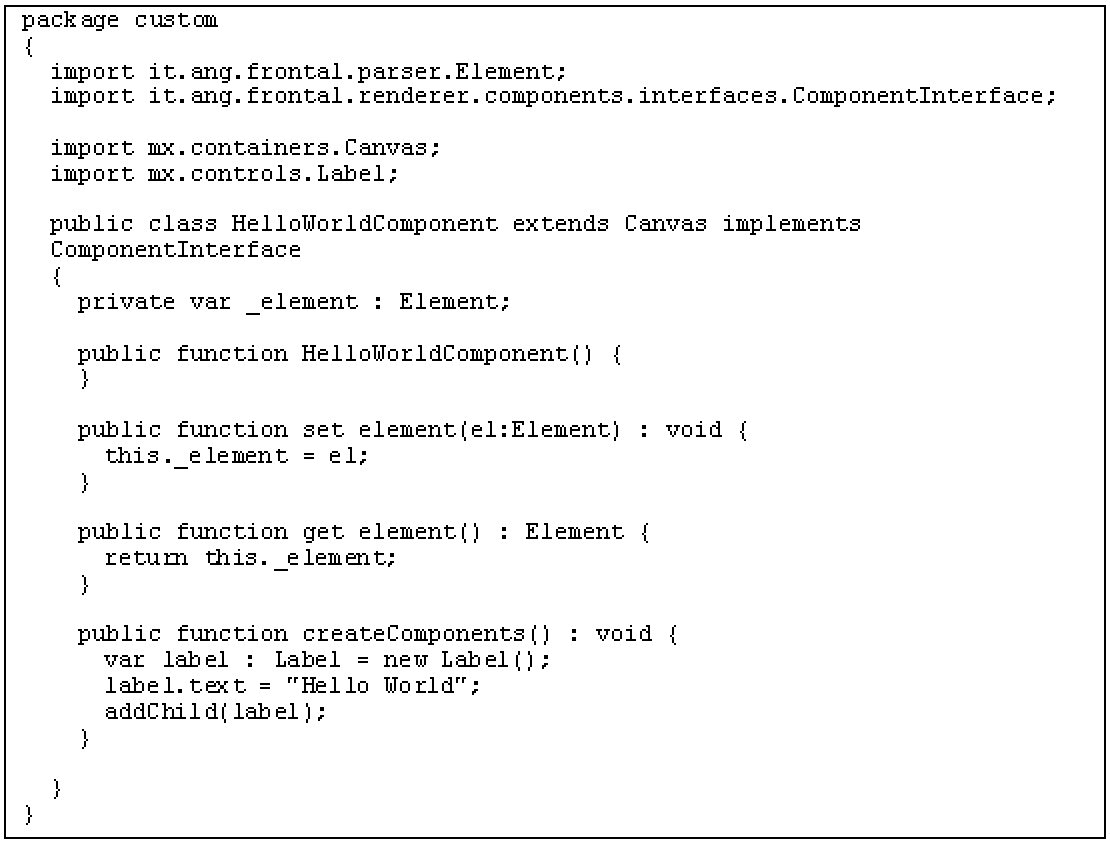
```
package it.ang.frontal.renderer.components.interfaces
{
  import it.ang.frontal.parser.Element;

  public interface ComponentInterface
  {

    function set element(el:Element) : void;
    function get element() : Element;
    function createComponents() : void;

  }
}
```

Figure 3 – ComponentInterface

The methods set and get are used to let Frontal pass to your component the Element Object; this object represents the complete tag element with, eventualy, its nested children.

The *createComponents* method is invoked by Frontal to let to your component manage the content of your tag.

In figure 4 there is an example of component.

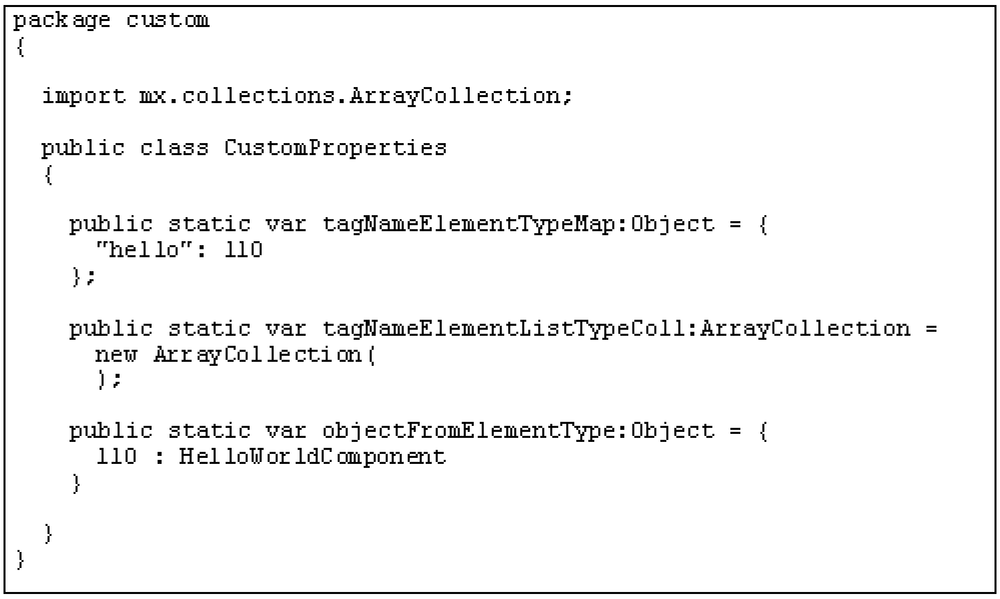


```
package custom
{
  import it.ang.frontal.parser.Element;
  import it.ang.frontal.renderer.components.interfaces.ComponentInterface;

  import mx.containers.Canvas;
  import mx.controls.Label;

  public class HelloWorldComponent extends Canvas implements
  ComponentInterface
  {
    private var _element : Element;

    public function HelloWorldComponent() {
    }

    public function set element(el:Element) : void {
      this._element = el;
    }

    public function get element() : Element {
      return this._element;
    }

    public function createComponents() : void {
      var label : Label = new Label();
      label.text = "Hello World";
      addChild(label);
    }

  }
}
```

Figure 4 – HelloWorldComponent

The *HelloWorldComponent* is a Canvas and it renders a Label Object with text 'Hello World'. This text could be, for example, passed also from the html content via an attribute field; in the *createComponents* method you would have had this attribute available in the *attributeMap* property of the Element Object.

The next step is to configure the Frontal framework; that has to be done setting the static ApplicationContext property *arrayConstantsClasses* with a your class; this class must have three static property as in figure 5.



```
package custom
{

  import mx.collections.ArrayCollection;

  public class CustomProperties
  {

    public static var tagNameElementTypeMap:Object = {
      "hello": 110
    };

    public static var tagNameElementListTypeColl:ArrayCollection =
      new ArrayCollection(
      );

    public static var objectFromElementType:Object = {
      110 : HelloWorldComponent
    }

  }
}
```

Figure 5 – CustomProperties

The *tagNameElementTypeMap* property is a map which links the name of the custom tag with a value important for the Frontal framework; these values have to be greater than 109, because the smaller ones are used by Frontal.

The *tagNameElementListTypeColl* property is a list of values which represent the corrisponding tags in the *tagNameElementTypeMap* map; this list is useful to inform Frontal to not analyze and render any child tag of the custom tag; for example, if the *hello* tag had one or more children, Frontal automatically tries to analyze and render them; if you don't want this behaviour, because it's in the *HelloWorldComponent* that you insert the creation of the children tags, you have to insert the *hello* tag code, that is 110, in this list.

The *objectFromElementType* propery is necessary to correlate the tag, represented by its code, and the Flex class that manages it.

The class with these properties just made, has to be inserted in the *ApplicationContext* as in figure 6.

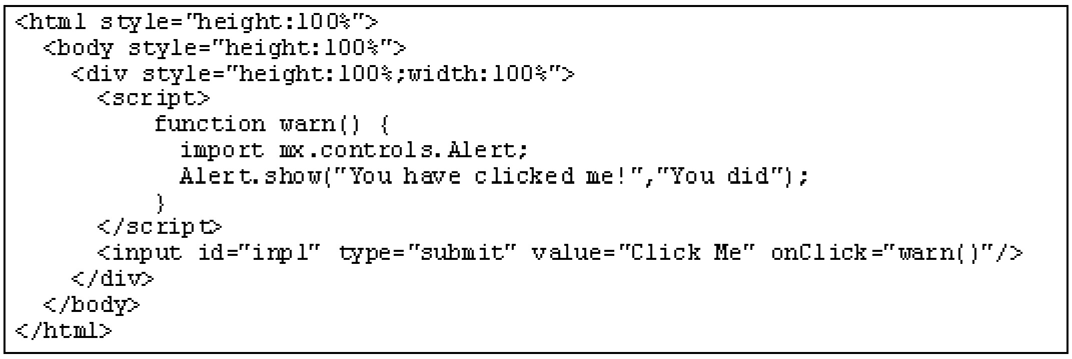
```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
                width="100%" height="100%" horizontalScrollPolicy="off"
                creationComplete="init()"
xmlns:components="it.ang.frontal.renderer.components.*">

        <mx:Style source="project/css/main.css"/>

        <mx:Script>
                <![CDATA[
                        import it.ang.frontal.util.CommonUtils;
                        import it.ang.frontal.modules.util.ApplicationContext;
                        import custom.CustomProperties;

                        public function init() : void {
                                ApplicationContext.arrayConstantsClasses = [CustomProperties];
                                ApplicationContext.getInstance().rootComponent = rootComponent;
                                ApplicationContext.getInstance().invokeService("project/index.html",
CommonUtils.getUrlParameters());
                        }

                ]]>
        </mx:Script>

        <components:RootHtmlContainer id="rootComponent" width="100%" height="100%" />

</mx:Application>
```

Figure 6 – Application File

ACTIONSCRIPT & EVENTS

In order to let users to interact with your web application, you can develop using action script AS3 as you normally do with Javascript in your html pages.

You can insert your AS3 code both into the script tag and directly in the declaration of the tag, in responding to an event as shown in the example in figure 7.



```
<html style="height:100%">
  <body style="height:100%">
    <div style="height:100%;width:100%">
      <script>
          function warn() {
            import mx.controls.Alert;
            Alert.show("You have clicked me!","You did");
          }
      </script>
      <input id="inp1" type="submit" value="Click Me" onClick="warn()"/>
    </div>
  </body>
</html>
```

Figure 7 – ActionScript integration

Into your AS code you can import all the classes of the Flex Framework you want to use; further Frontal gives you the interaction with two Frontal classes, the *ApplicationContext* class and the *ModuleContext* class.

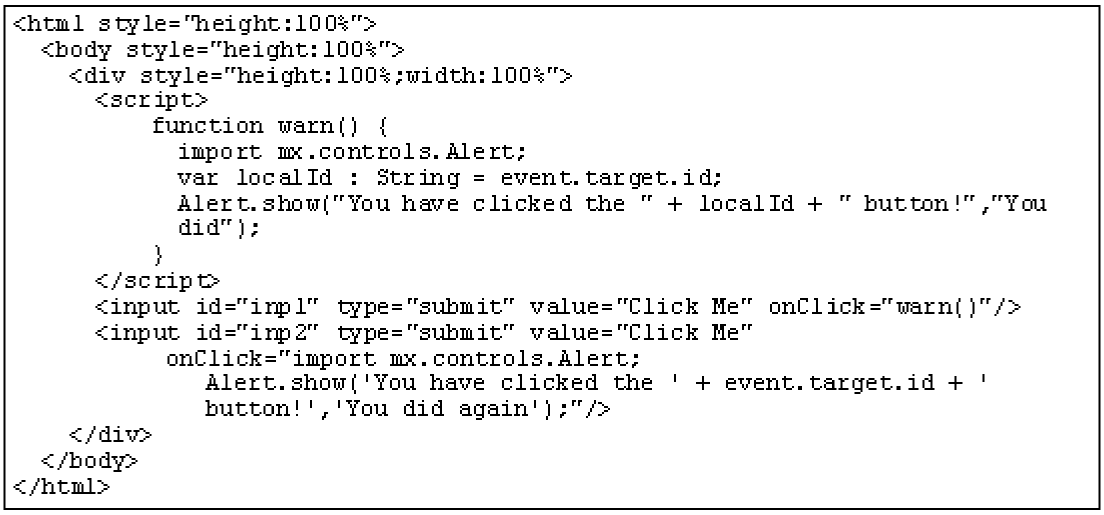
- ***ApplicationContext***: referenced with *appCtx*, you can access to every component in the application with the *getComp* method passing the id attribute value of the component; in the ApplicationContext class there is, also, the *local* property; this property is an object you can utilize for store and read variables.
- ***ModuleContext***: referenced with *ctx*, you can access to every component in the current module with the *getComp* method passing the id attribute value of the component.

Here is the list of the event you can integrate into your html content:

- **onClick**: dispatched on the mouse click on a component and turned into a Flex MouseEvent.
- **onChange**: dispatched on the selection of an item in a list component, or when the value of an input field changes; it is turned into a Flex Event.
- **itemRollOver**: dispatched when the mouse pointer rolls onto a Menu item and turned into a Flex MenuEvent.
- **creationComplete**: dispatched when the component has finished its construction, property processing, measuring, layout, and drawing; turned into a Flex FlexEvent.
- **onLoad**: the same as the creationComplete event.

- **dataChange**: dispatched when the data property changes; this is a property of the Flex Framework components important when these components are used as renderers; turned into a Flex FlexEvent.
- **onMouseOver**: dispatched when the user moves a pointing device over a component; turned into a Flex MouseEvent.
- **onMouseOut**: dispatched when the user moves a pointing device away from a component; turned into a Flex MouseEvent.
- **onMouseDown**: dispatched when a user presses the pointing device button over a component; turned into a Flex MouseEvent.
- **onMouseUp**: dispatched when a user releases the pointing device button over a component; turned into a Flex MouseEvent.
- **onKeyPress**: dispatched when the user presses a key; turned into a Flex KeyboardEvent.
- **onFocus**: dispatched after a display object gains focus; turned into a Flex FocusEvent.
- **onBlur**: dispatched after a display object loses focus; turned into a Flex FocusEvent.
- **onDblClick**: dispatched when a user presses and releases the main button of a pointing device twice in rapid succession over the same component; turned into a Flex MouseEvent.

In the event listener function or, directly in the event attribute in the declaration of the tag, Frontal passes also the event object so it can be accessed by your AS code, as shown in figure 8.



```
<html style="height:100%">
  <body style="height:100%">
    <div style="height:100%;width:100%">
      <script>
          function warn() {
            import mx.controls.Alert;
            var localId : String = event.target.id;
            Alert.show("You have clicked the " + localId + " button!","You
            did");
          }
      </script>
      <input id="inp1" type="submit" value="Click Me" onClick="warn()"/>
      <input id="inp2" type="submit" value="Click Me"
           onClick="import mx.controls.Alert;
              Alert.show('You have clicked the ' + event.target.id + '
              button!','You did again');"/>
    </div>
  </body>
</html>
```

Figure 8 – Events

EFFECTS

Flex provides the opportunity to enrich applications by introducing visual effects with ease.

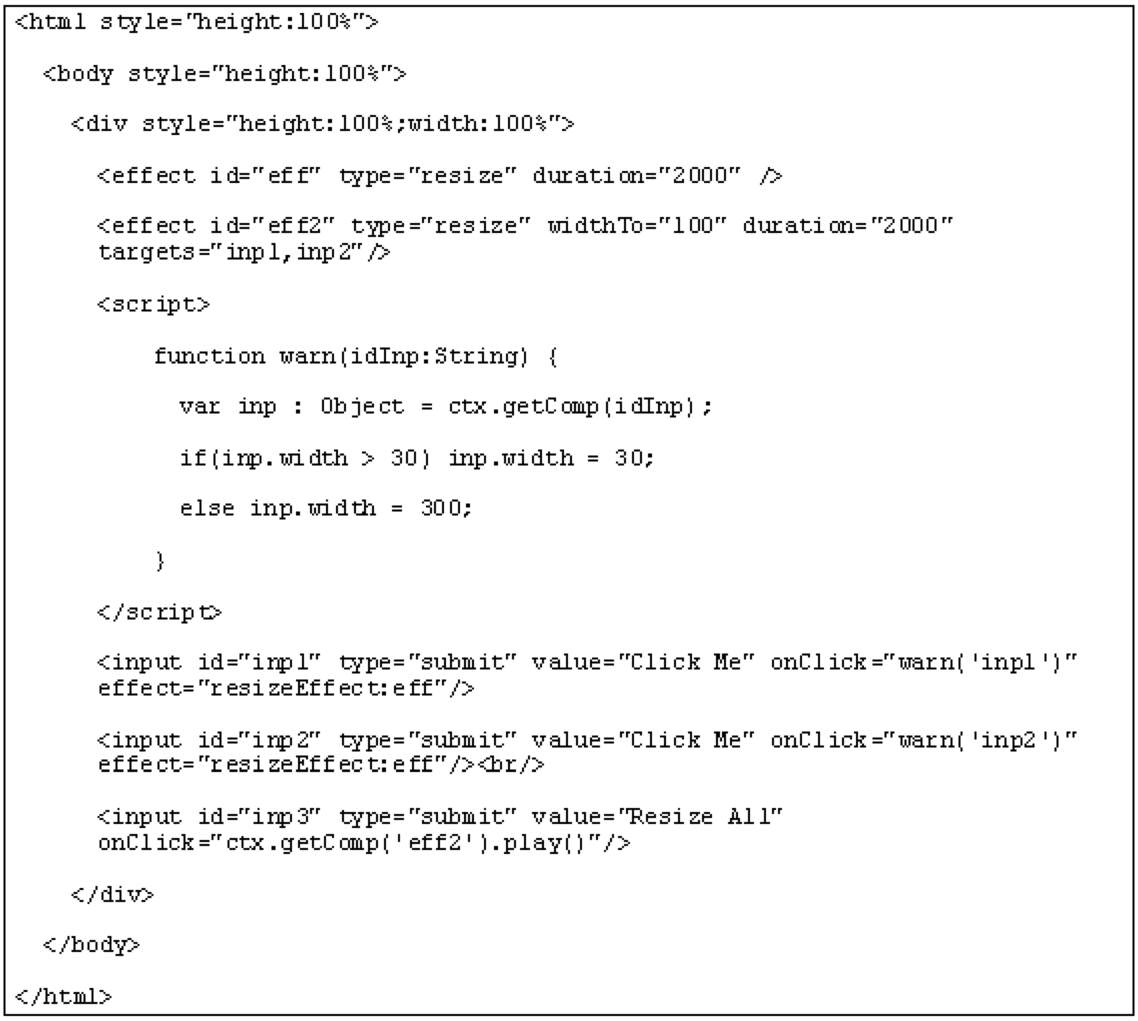
Frontal embraces this opportunity by declaring an own tag to be inserted in the html content, and this tag is called *effect* and its main attribute is the *type* attribute; this attribute specifies the type of effect to be delivered.

Here is the list of the possible effects to introduce coded by the attribute *type*:

- **wipeLeft** : WipeLeft
- **wipeRight** : WipeRight
- **wipeDown** : WipeDown
- **wipeUp** : WipeUp
- **resize** : Resize
- **move** : Move
- **zoom** : Zoom
- **blur** : Blur
- **fade** : Fade
- **glow** : Glow
- **iris** : Iris

- **rotate** : Rotate
- **tween** : Tween
- **dissolve** : Dissolve

In the *effect* tag it is possible to insert all the attributes which can be assigned to the particular effect created; the effect created can be accessed also with the AS code using the *ApplicationContext* or the *ModuleContext* with the *getComp* method and then applying the methods *play*, *playAll*, *stop*, or directly accessing to the Flex effect instance with the property *effect*.



```
<html style="height:100%">

  <body style="height:100%">

    <div style="height:100%;width:100%">

      <effect id="eff" type="resize" duration="2000" />

      <effect id="eff2" type="resize" widthTo="100" duration="2000"
      targets="inp1,inp2"/>

      <script>

          function warn(idInp:String) {

            var inp : Object = ctx.getComp(idInp);

            if(inp.width > 30) inp.width = 30;

            else inp.width = 300;

          }

      </script>

      <input id="inp1" type="submit" value="Click Me" onClick="warn('inp1')"
      effect="resizeEffect:eff"/>

      <input id="inp2" type="submit" value="Click Me" onClick="warn('inp2')"
      effect="resizeEffect:eff"/><br/>

      <input id="inp3" type="submit" value="Resize All"
      onClick="ctx.getComp('eff2').play()"/>

    </div>

  </body>

</html>
```

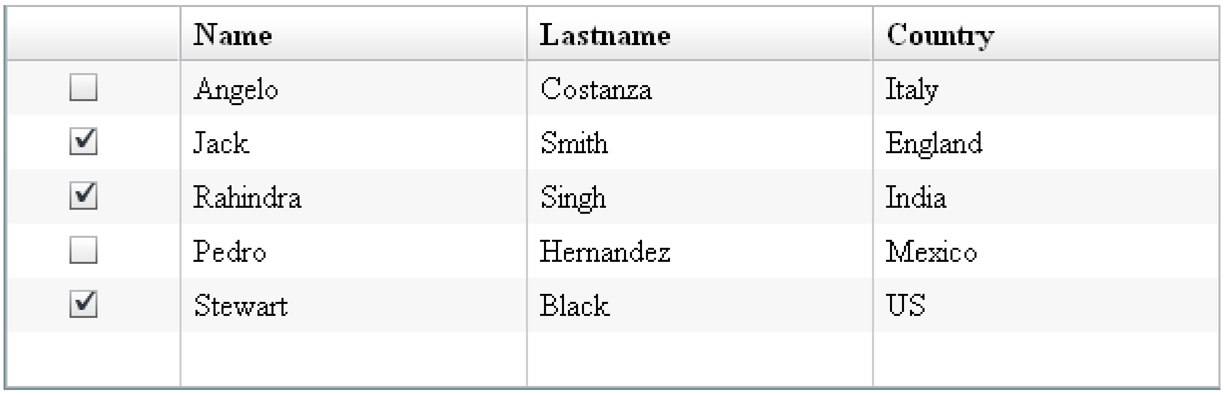
Figure 9 – Effects

SAMPLES

DATAGRID

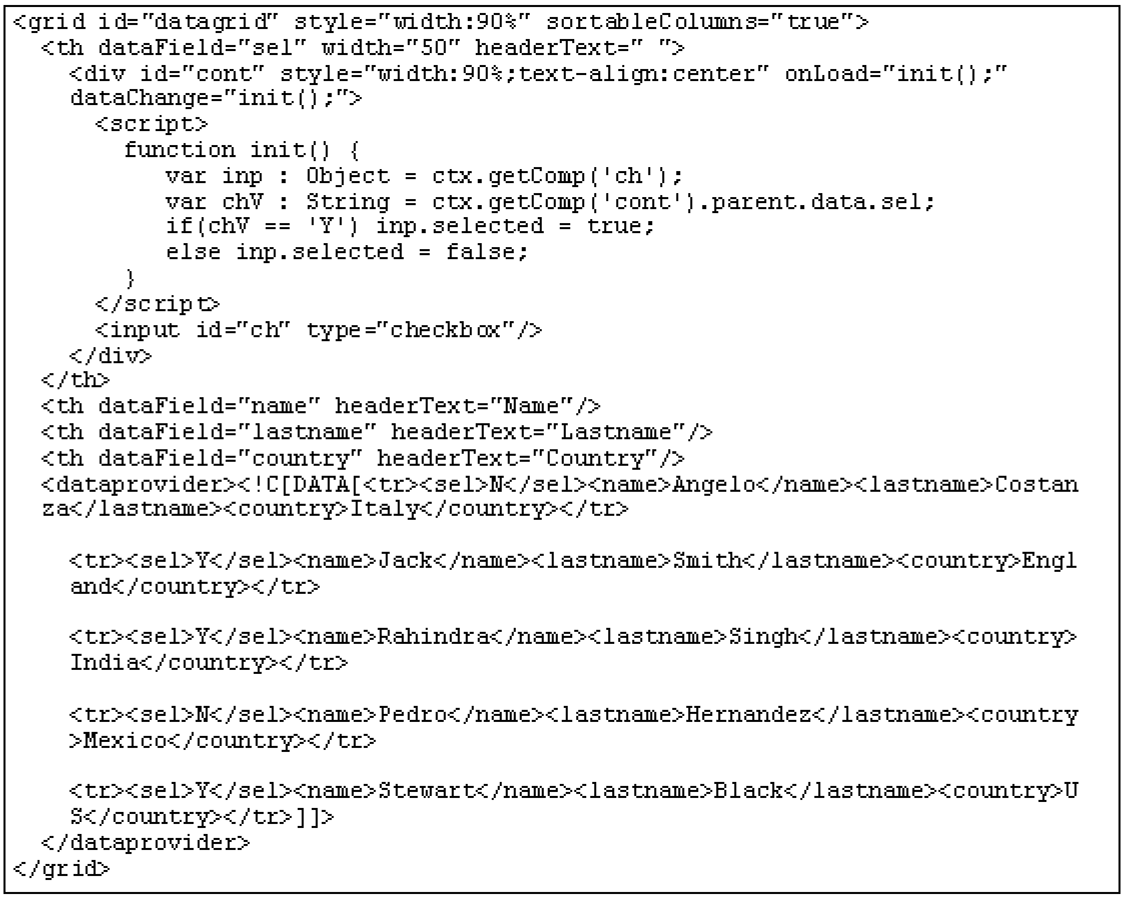
In this example is shown the integration, already available in Frontal, of the Flex component DataGrid; in the example in figure 10 there is the visualization of the DataGrid with the html source and the Custom Frontal Class DataGridContainer in figure 11 and 12.

The tag inserted is grid and it is converted to an instance of the *DataGridContainer* class; in the configuration mechanism the *grid* tag is inserted also into the *tagNameElementListTypeColl* property, which instructs Frontal to no treat the children elements of the *grid* tag, because it will be the *DataGridContainer* to do this job.



	Name	Lastname	Country
☐	Angelo	Costanza	Italy
☑	Jack	Smith	England
☑	Rahindra	Singh	India
☐	Pedro	Hernandez	Mexico
☑	Stewart	Black	US

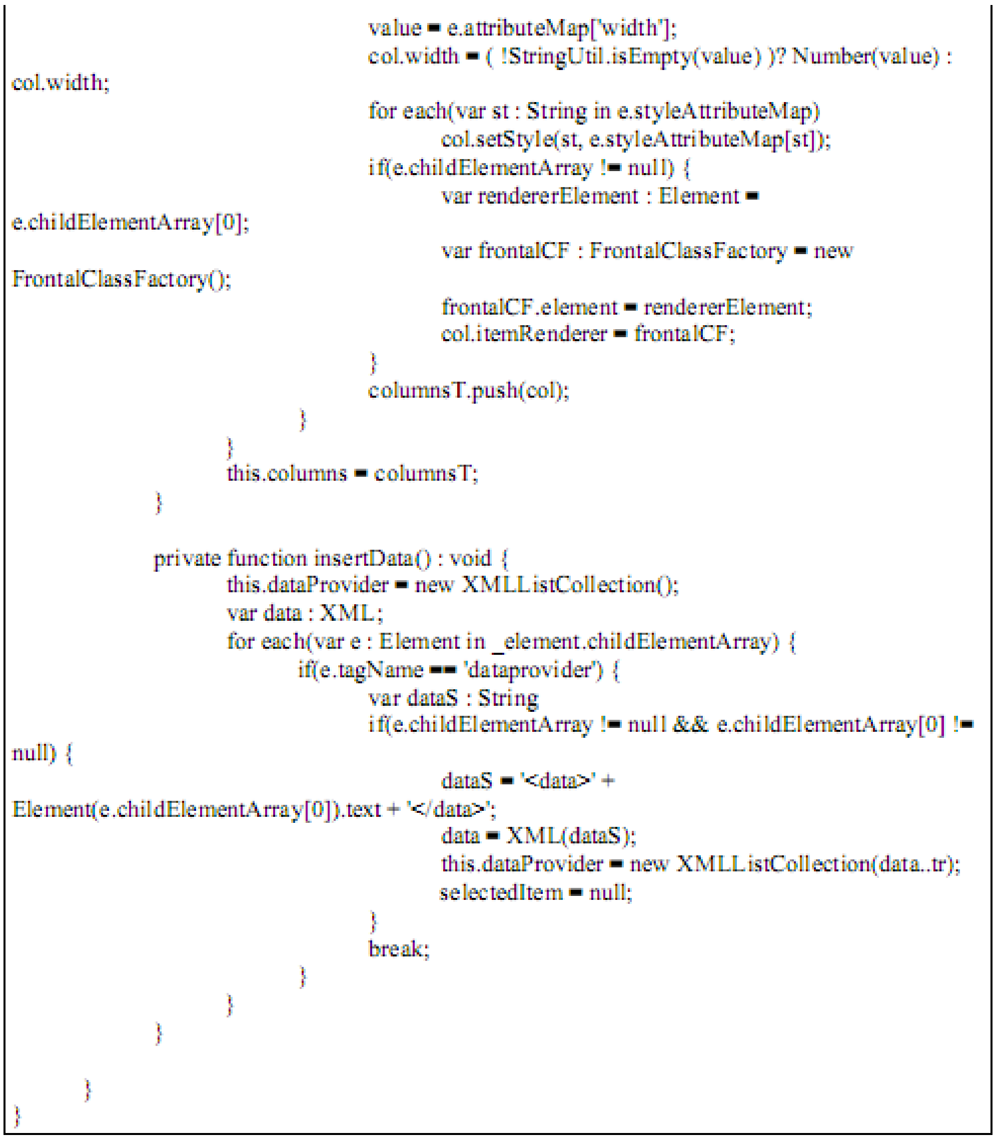
Figure 10 - DataGrid



```
<grid id="datagrid" style="width:90%" sortableColumns="true">
  <th dataField="sel" width="50" headerText=" ">
    <div id="cont" style="width:90%;text-align:center" onLoad="init();"
    dataChange="init();">
      <script>
        function init() {
           var inp : Object = ctx.getComp('ch');
           var chV : String = ctx.getComp('cont').parent.data.sel;
           if(chV == 'Y') inp.selected = true;
           else inp.selected = false;
        }
      </script>
      <input id="ch" type="checkbox"/>
    </div>
  </th>
  <th dataField="name" headerText="Name"/>
  <th dataField="lastname" headerText="Lastname"/>
  <th dataField="country" headerText="Country"/>
  <dataprovider><!C[DATA[<tr><sel>N</sel><name>Angelo</name><lastname>Costan
  za</lastname><country>Italy</country></tr>

    <tr><sel>Y</sel><name>Jack</name><lastname>Smith</lastname><country>Engl
    and</country></tr>

    <tr><sel>Y</sel><name>Rahindra</name><lastname>Singh</lastname><country>
    India</country></tr>

    <tr><sel>N</sel><name>Pedro</name><lastname>Hernandez</lastname><country
    >Mexico</country></tr>

    <tr><sel>Y</sel><name>Stewart</name><lastname>Black</lastname><country>U
    S</country></tr>]]>
  </dataprovider>
</grid>
```

Figure 11 – Grid Html Source

```
package it.ang.frontal.renderer.components
{
        import it.ang.frontal.parser.Element;
        import it.ang.frontal.renderer.components.interfaces.ComponentInterface;
        import it.ang.frontal.util.Constants;
        import it.ang.frontal.util.FrontalClassFactory;
        import it.ang.frontal.util.StringUtil;

        import mx.collections.XMLListCollection;
        import mx.controls.DataGrid;
        import mx.controls.dataGridClasses.DataGridColumn;

        public class DataGridContainer extends DataGrid implements ComponentInterface
        {

                private var _element : Element;

                public function DataGridContainer()
                {
                        super();
                }

                public function set element(el:Element) : void {
                        this._element = el;
                }

                public function get element() : Element {
                        return this._element;
                }

                public function createComponents() : void {
                        createCols();
                        insertData();
                }

                private function createCols() : void {
                        var columnsT : Array = new Array();
                        var value : String;
                        for each(var e : Element in _element.childElementArray) {
                                if(e.elementType == Constants.elementTypeTH) {
                                        value = e.attributeMap['visible'];
                                        if ( !StringUtil.isEmpty(value) ) {
                                                if(value == 'false')
                                                        continue;
                                        }
                                        var col : DataGridColumn = new DataGridColumn();
                                        value = e.attributeMap['dataField'];
                                        col.dataField = ( !StringUtil.isEmpty(value) )? value : 'label';
                                        value = e.attributeMap['headerText'];
                                        col.headerText = ( !StringUtil.isEmpty(value) )? value :
'Header';
```



```
                                        value = e.attributeMap['width'];
                                        col.width = ( !StringUtil.isEmpty(value) )? Number(value) :
col.width;
                                        for each(var st : String in e.styleAttributeMap)
                                             col.setStyle(st, e.styleAttributeMap[st]);
                                        if(e.childElementArray != null) {
                                             var rendererElement : Element =
e.childElementArray[0];
                                             var frontalCF : FrontalClassFactory = new
FrontalClassFactory();
                                             frontalCF.element = rendererElement;
                                             col.itemRenderer = frontalCF;
                                        }
                                        columnsT.push(col);
                                   }
                              }
                              this.columns = columnsT;
                         }

                         private function insertData() : void {
                              this.dataProvider = new XMLListCollection();
                              var data : XML;
                              for each(var e : Element in _element.childElementArray) {
                                   if(e.tagName == 'dataprovider') {
                                        var dataS : String
                                        if(e.childElementArray != null && e.childElementArray[0] !=
null) {
                                             dataS = '<data>' +
Element(e.childElementArray[0]).text + '</data>';
                                             data = XML(dataS);
                                             this.dataProvider = new XMLListCollection(data..tr);
                                             selectedItem = null;
                                        }
                                        break;
                                   }
                              }
                         }

          }
}
```

Figure 12 – DataGridContainer

As you can see from the source of the *DataGridContainer* Class, the creation of the columns and the reading of the data to visualize is responsability of the class, which is a DataGrid.

In the *createCols* method there is also the use of another Frontal class, that is the *FrontalClassFactory* class; this class extends the Flex ClassFactory and is able to generate components from a generic Element object; this is very useful in cases of item renderer objects.

List

In this example is shown the integration, already available in Frontal, of the Flex component TileList; in the example below there is the visualization of the TileList with the html source and the Custom Frontal Class *ListContainer*.

The tag inserted is *list* and it is converted to an instance of the *ListContainer* class; in the configuration mechanism the *list* tag is inserted also into the *tagNameElementListTypeColl* property, which instructs Frontal to no treat the children elements of the *list* tag, because it will be the ListContainer to do this job.

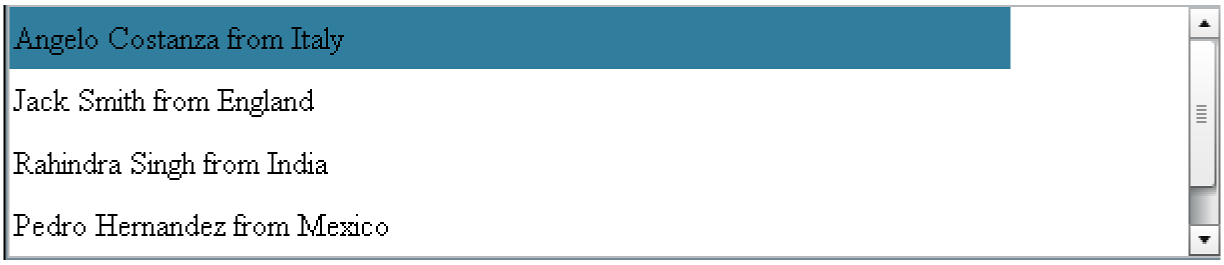

Angelo Costanza from Italy
Jack Smith from England
Rahindra Singh from India
Pedro Hernandez from Mexico

Figure 13 – List

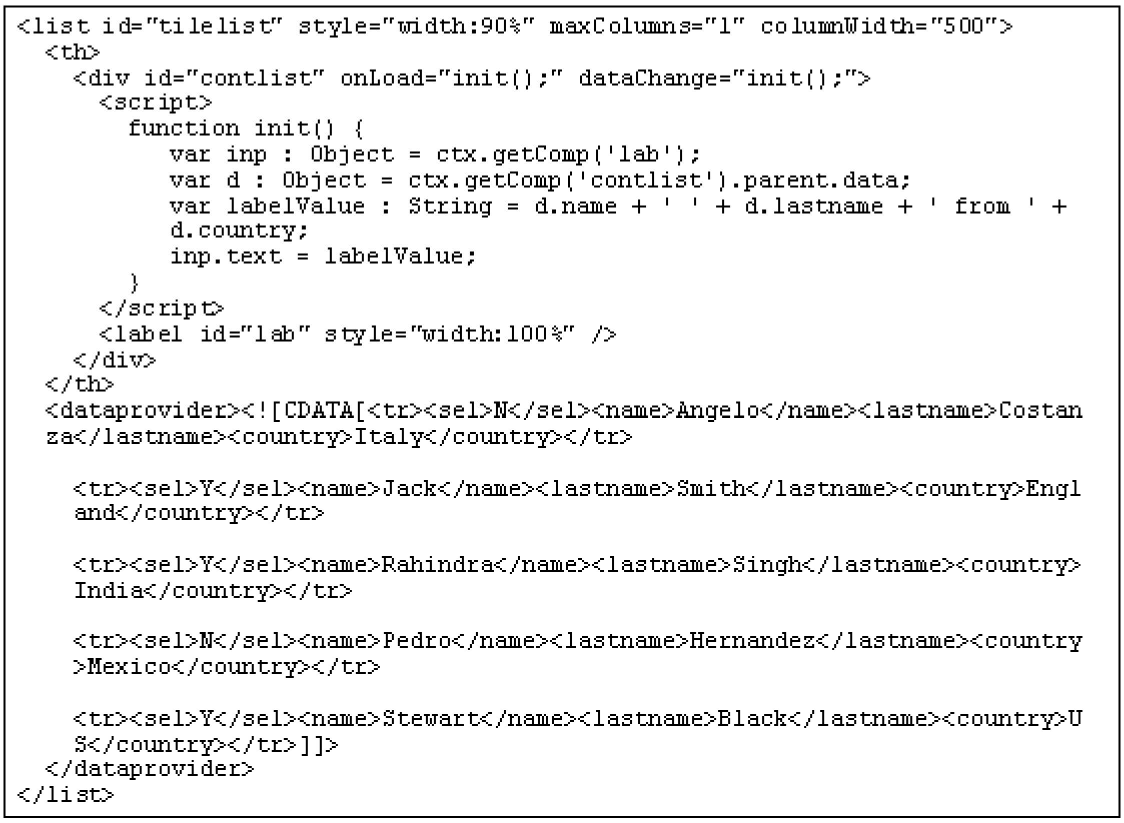

```
<list id="tilelist" style="width:90%" maxColumns="1" columnWidth="500">
  <th>
    <div id="contlist" onLoad="init();" dataChange="init();">
      <script>
        function init() {
           var inp : Object = ctx.getComp('lab');
           var d : Object = ctx.getComp('contlist').parent.data;
           var labelValue : String = d.name + ' ' + d.lastname + ' from ' +
           d.country;
           inp.text = labelValue;
        }
      </script>
      <label id="lab" style="width:100%" />
    </div>
  </th>
  <dataprovider><![CDATA[<tr><sel>N</sel><name>Angelo</name><lastname>Costan
  za</lastname><country>Italy</country></tr>

    <tr><sel>Y</sel><name>Jack</name><lastname>Smith</lastname><country>Engl
    and</country></tr>

    <tr><sel>Y</sel><name>Rahindra</name><lastname>Singh</lastname><country>
    India</country></tr>

    <tr><sel>N</sel><name>Pedro</name><lastname>Hernandez</lastname><country
    >Mexico</country></tr>

    <tr><sel>Y</sel><name>Stewart</name><lastname>Black</lastname><country>U
    S</country></tr>]]>
  </dataprovider>
</list>
```

Figure 14 – List Html Source

```
package it.ang.frontal.renderer.components
{
        import it.ang.frontal.parser.Element;
        import it.ang.frontal.renderer.components.interfaces.ComponentInterface;
        import it.ang.frontal.util.Constants;
        import it.ang.frontal.util.FrontalClassFactory;

        import mx.collections.XMLListCollection;
        import mx.controls.TileList;

        public class ListContainer extends TileList implements ComponentInterface
        {
                private var _element : Element;

                public function set element(el:Element) : void {
                        this._element = el;
                }

                public function get element() : Element {
                        return this._element;
                }

                public function ListContainer() {
                        super();
                }

                public function createComponents() : void {
                        for each(var e : Element in _element.childElementArray) {
                                if(e.elementType == Constants.elementTypeTH) {
                                        if(e.childElementArray != null) {
                                                var rendererElement : Element = e.childElementArray[0];
                                                var frontalCF : FrontalClassFactory = new FrontalClassFactory();
                                                frontalCF.element = rendererElement;
                                                this.itemRenderer = frontalCF;
                                        }
                                        break;
                                }
                        }
                        insertData();
                }

                private function insertData() : void {
                        this.dataProvider = new XMLListCollection();
                        var data : XML;
                        for each(var e : Element in _element.childElementArray) {
                                if(e.tagName == "dataprovider") {
                                        var dataS : String
                                        if(e.childElementArray != null && e.childElementArray[0] != null) {
                                                dataS = '<data>' + Element(e.childElementArray[0]).text +
'</data>';

                                                data = XML(dataS);
                                                this.dataProvider = new XMLListCollection(data..tr);
                                                selectedItem = null;
                                        }
                                        break;
                                }
                        }
                }

        }
}
```

Figure 15 – ListContainer

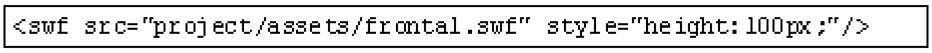
As you can see from the source of the *ListContainer* Class, the creation of the probable item renderer and the reading of the data to visualize is responsability of the class, which is a TileList.

LABEL

n the Frontal project it is present also the Flex component Label, for a pure sense of convenience; it is associated to the html tag *label* and its property text is populated by the html attribute *value*.

SWF

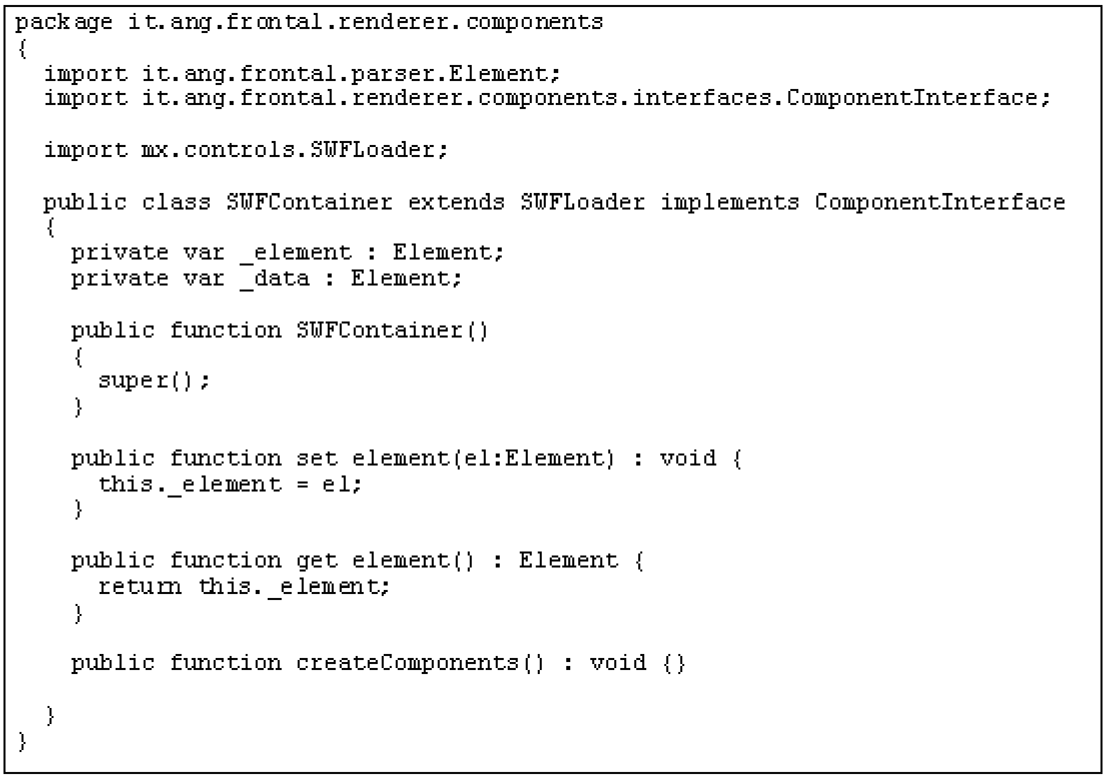
In the Frontal project it has been configured a particular tag named *swf*'; this is converted by Frontal to a *SWFContainer*, which extends the Flex SWFLoader Object, able to load the swf passed in the *src* attribute; in the tag it is possible also declare the style attributes as width and height.



```
<swf src="project/assets/frontal.swf" style="height:100px;"/>
```

Figure 16 – Swf Html Source

Figure 17 – Swf loaded



```
package it.ang.frontal.renderer.components
{
  import it.ang.frontal.parser.Element;
  import it.ang.frontal.renderer.components.interfaces.ComponentInterface;

  import mx.controls.SWFLoader;

  public class SWFContainer extends SWFLoader implements ComponentInterface
  {
    private var _element : Element;
    private var _data : Element;

    public function SWFContainer()
    {
      super();
    }

    public function set element(el:Element) : void {
      this._element = el;
    }

    public function get element() : Element {
      return this._element;
    }

    public function createComponents() : void {}

  }
}
```

Figure 18 – SWFContainer

Best Practices

Here are some best practices in order to use Frontal with all its potentialities.

First of all, in the declaration of the tags, it is usefull defining the style properties width and height; this leads to a better graphical result mostly when are used container tags such as div and table.

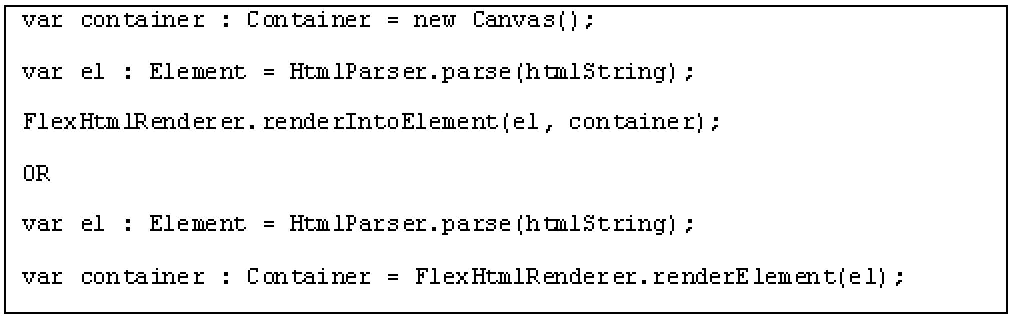
The second advice is to insert text into a container, for example span and div; these containers must have defined the width style property, and it's usefull inserting the text into a CDATA section in order to inform Frontal to ignore the parsing of the text.

Renderer

In the Frontal project it is possible to interact directly with the components responsable of the html parsing and of the generation of the Flex objects.

The class which parses the html code is the *HtmlParser* class; it has the method parse with the input parameter represented by the html code; this method delivers an *Element* instance built upon the html code, populated with all the nested *Element* children.

The class which gives birth to all the Flex objects is the *FlexHtmlRenderer* class; this class has two important static methods that are *renderElement* and *renderIntoElement*; the first one accepts as input an *Element* object returning the UIComponent generated, the second one accepts the as first input the Element object an then the UIComponent in which it is desired to insert the components generated by Frontal.



```
var container : Container = new Canvas();

var el : Element = HtmlParser.parse(htmlString);

FlexHtmlRenderer.renderIntoElement(el, container);

OR

var el : Element = HtmlParser.parse(htmlString);

var container : Container = FlexHtmlRenderer.renderElement(el);
```

Figure 19 – Frontal Classes

NEXT WORKS

The Frontal project is compiled using the 3.4.1 Flex sdk; next works will include the development under the 4 sdk version, facing with the Spark components.

Other works will be directed forward the integration of Frontal with cms projects as Joomla!.

About the Author

Angelo Costanza

Born in Bari, Italy.

Degree in Computer Engineering.

Working as Software Engineer in Milan.

mail: angcostanza@gmail.com

On-Line References

http://frontal.isapp.it

http://www.isapp.it