WidgetDeveloperManual

Updated May 18, 2010 by gessedafe

1 Introduction

If you are not familiar with ***Crux*** framework, please consult the User Manual first.

This documentation will show you how to work with custom widgets inside ***Crux***.

1.1 GWT widgets

Crux widgets are GWT widgets. ***Crux*** just offer an aditional engine where those widgets can be plugged.

So, you can plug any custom widgets in ***Crux*** Declarative Engine. If you are not familiar with GWT widgets development, please consult its documentation first.

2 *Crux* Declarative Engine

2.1 Overview

Crux declarative engine uses factories to create the widgets based on informations contained inside the html pages. For a deeper explanation about how this declarative engine works, consult the following section.

To register a custom widget in ***Crux*** declarative engine, you just have to create a factory class for it.

See the following example:

```
@DeclarativeFactory(id="myWidget", library="myLibrary")
public class MyWidgetFactory extends WidgetFactory<MyWidget>
{
   @Override
   public MyWidget instantiateWidget(Element element, String widgetId)
   {
      return new MyWidget();
   }

   @Override
   @TagAttributes({
      @TagAttribute("myWidgetAttribute")
   })
   public void processAttributes(WidgetFactoryContext<MyWidget> context) throws InterfaceConfigException
   {
      super.processAttributes(context);
   }
}
```

Note that the above class uses the annotation `@DeclarativeFactory` and extends the abstract class `WidgetFactory`. These two conditions are necessary for any ***Crux*** widget factory.

Through the `@DeclarativeFactory` annotation, you can specify the name of the library where your widget will be registered and the name that will be associated with your widget itself.

Before we go deeper inside the code shown and see more examples, let's see how you could use the widget registered with ***Crux*** declarative engine:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:mylib="http://www.sysmap.com.br/crux/myLibrary" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <mylib:myWidget id="myWidgetID" myWidgetAttribute="attributeValue"/>
   </body>
</html>
```

Note that a namespace called `http://www.sysmap.com.br/crux/myLibrary` is used on the above page. A `XSD` file defining this namespace is created by the SchemaGenerator.

The example shown assumes that the widget `MyWidget` contains a public String property called `myWidgetAttribute` that will be bound to the tag's attribute `myWidgetAttribute` on `crux.xml` file.

2.2 How The Engine Works

If you look to the source code of any ***Crux*** page on your browser, you will find a lot of `<span>` tags similar to the following one:

```
<span id="myWidgetID" _type="myLibrary_myWidget" _myWidgetAttribute="attributeValue"></span>
```

This HTML page with those tags is generated by the Crux Compiler.

The ***Crux*** EntryPoint will start a factory class called `ScreenFactory`, that will search for all `<span>` tags with the attribute `_type` on the page document. Those tags represent your widgets declarations on the `.crux.xml` file.

`ScreenFactory` uses the attribute `_type` to decide which factory class it wil call to create the desired widget, for each of those `<span>` tags found. Then, `ScreenFactory` replaces the `<span >` tag by the widget created.

Of course, we could use another kind of metadata structure on HTML pages, like javascript arrays, or some kind of strings (like XML) on a javascript block to serve as input for ***Crux*** `ScreenFactory`. However, some widgets on a page contain other HTML blocks as

children. Sometimes, those HTML blocks contain other child widgets... So, the most natural way to handle this is to use the HTML DOM itself as the source of the metadata. It turn the implementation very easy and the page loads much faster.

For aditional information about this declarative approach, consult the ***Crux*** FAQ.

2.3 WidgetFactory

`WidgetFactory` is the basic class for any ***Crux*** widget factory. `ScreenFactory` delegates to those factories the creation of the requested widget, based ont the `<span>` element on the DOM.

`WidgetFactory.createWidget()` method is called for this creation. It executes the following steps:

1. Calls the method `T instantiateWidget(element, widgetId)`
2. Calls the method `void processAttributes(WidgetFactoryContext<T> context)`
3. Calls the method `void processEvents(WidgetFactoryContext<T> context)`
4. Calls the method `void processChildren(WidgetFactoryContext<T> context)`
5. Calls the method `void postProcess(WidgetFactoryContext<T> context)`
6. Return the created widget to `ScreenFactory`, that will properly attach it on screen and replaces its `<span>` metadata tag.

The `instantiateWidget` method is abstract and must be implemented by any Factory subclass. The other methods can be used by the factory to specify and process the information about widget's attributes, events and children. All of those other methods receive a parameter of type `WidgetFactoryContext`. This class has the following public methods:

Method	Description
getWidget	Returns the widget previously instantiated be the method `instantiateWidget`
getElement	Returns the widget metadata element (the `<span>` tag)
getWidgetId	Returns the widget identifier
setAttribute(String key, Object value)	Allows the factory to set some information to use in some other step of the parser, (like during the children processing)
getAttribute(String key)	Returns some attribute previously set by method setAttribute

The next sections describe how to properly overwrite each of those methods to serve your widget needs.

2.3.1 The instantiateWidget method

The `instantiateWidget` method is called to instantiate a new widget object . The `<span>` element is passed as parameter and can be used if the widget to be created needs some attribute in its constructor.If the widget shown on section 2.1 needs some attribute, we could write something like:

```
@DeclarativeFactory(id="myWidget", library="myLibrary")
public class MyWidgetFactory extends WidgetFactory<MyWidget>
{
   @Override
   public MyWidget instantiateWidget(Element element, String widgetId)
   {
      return new MyWidget(element.getAttribute("_theRequiredAttribute"));
   }
  ...
}
```

The purpose of this method is just instantiate the widget. The best place to handle the extraction of attributes, events and children is not here. You must use the other methods, shown bellow, in order to take advantage of all benefits this engine offers you.

2.3.2 Attributes and Events Processing

The `processAttributes` and `processEvents` methods are used to the binding of the `<span>` elements and the widgets created.

2.3.2.1 The processAttributes method

This method has two goals:

1. Turn easier the binding beetwen the widget metadata element (the `<span>` tag) and the widget itself.
2. Inform to ***Crux*** engine which attributes this factory can handle. This information is used to generate a proper XSD file, that can be used to enable autocompletion for developers (See SchemaGenerator).

See the following code:

```
@DeclarativeFactory(id="myWidget", library="myLibrary")
public class MyWidgetFactory extends WidgetFactory<MyWidget>
{

   @Override
   @TagAttributesDeclaration({
      @TagAttributeDeclaration("myWidgetAttribute")
   })
   public void processAttributes(WidgetFactoryContext<MyWidget> context) throws InterfaceConfigException
   {
      super.processAttributes(context);

      MyWidget widget = context.getWidget();
      String attr = element.getAttribute("_myWidgetAttribute");
      if (StringUtils.isNotEmpty(attr))
      {
         widget.setMyWidgetAttribute(attr);
      }
   }
}
```

The first thing done by the above processAttributes method is call `super.processAttributes`. It is important, once all attributes common to all GWT widgets (like `visible, width, height` and others) are handled by `WidgetFactory.processAttributes`.

Other important point to observe is the annotation `@TagAttributesDeclaration` on method. It is used to inform all attributes supported by this widget. This information is used by SchemaGenerator. Note that `WidgetFactory.processAttributes` has this same annotation including more attributes. SchemaGenerator will consider attributes declared on super classes to generate the XSD files.

If you observe the code shown on section 2.1, you will note that the annotation `@TagAttributes` is used in the place of `@TagAttributesDeclaration`. The two annotations are very similar and support the same attributes. However, when you use the `@TagAttributes`, ***Crux*** will generate the code for the binding of those attributes. The code on section 2.1 just declare the attribute `myWidgetAttribute` on the method annotation and does not need to implement all the code shown in this section.

In other words, the above code can bre written as:

```
@DeclarativeFactory(id="myWidget", library="myLibrary")
public class MyWidgetFactory extends WidgetFactory<MyWidget>
{
   @Override
   @TagAttributes({
      @TagAttribute("myWidgetAttribute")
   })
   public void processAttributes(WidgetFactoryContext<MyWidget> context) throws InterfaceConfigException
   {
      super.processAttributes(context);
   }
}
```

The annotation `@TagAttributesDeclaration` must be used when you want to inform ***Crux*** that the attributes exist (because the SchemaGenerator), but you need to parse it programmatically. The code below shows a situation where you can face this need: the widget has an attribute that is required by its constructor. Note that you can use both annotations on a `processAttributes` method:

```
@DeclarativeFactory(id="myWidget", library="myLibrary")
public class MyWidgetFactory extends WidgetFactory<MyWidget>
{
   @Override
   public MyWidget instantiateWidget(Element element, String widgetId)
   {
      return new MyWidget(element.getAttribute("_theRequiredAttribute"));
   }

   @Override
   @TagAttributes({
      @TagAttribute("myWidgetAttribute")
   })
   @TagAttributesDeclaration({
      @TagAttributeDeclaration(value="theRequiredAttribute", required=true)
   })
   public void processAttributes(WidgetFactoryContext<MyWidget> context) throws InterfaceConfigException
   {
      super.processAttributes(context);
   }
}
```

2.3.2.1.1 @TagAttributes Annotation

The `@TagAttributes` has one attribute (value) of type array of `@TagAttribute`. Each `@TagAttribute` is used to inform one attribute of the widget.

`@TagAttribute` accepts the following attributes:

Name	Type	Required	Default	Description
value	String	yes	-	The name of the attribute.
defaultValue	String	no	""	The default value to be used on XSD.
type	Class<?>	no	String	The type of the attribute.
required	boolean	no	false	Inform if the attribute is required.
supportsI18N	boolean	no	false	Inform if the attribute support ***Crux*** declarative i18n.

The supported types for an attribute is:

- any primitive type or primitive wrapper;
- String;
- Date;
- any enum type.

All enum types will be mapped to simpleTypes on the generated XSD.

The parser generated by ***Crux*** will handle all type conversions needed and all declarative i18n messages.

The same properties exist on `@TagAttributesDeclaration` annotation. However, using it, the parsing of the `<span>` element and the binding of the attributes is not done automatically.

2.3.2.2 The processEvents method

This method does almost the same as processAttributes, however, it handles widget events.

Two annotations can be used on `processEvents` method: `@TagEvents` and `@TagEventsDeclaration`.

We can compare these annotations with `@TagAttributes` and `@TagAttributesDeclaration` .

See the following example:

```
@DeclarativeFactory(id="myWidget", library="myLibrary")
public class MyWidgetFactory extends WidgetFactory<MyWidget>
{
```

```
    @Override
    @TagEvents({
        @TagEvent(ClickEvtBind.class)
    })
    @TagEventsDeclaration({
        @TagEventDeclaration("onCustomEvent")
    })
    public void processEvents(WidgetFactoryContext<MyWidget> context) throws InterfaceConfigException
    {
        super.processEvents(context);

        final Event eventCustom = EvtBind.getWidgetEvent(context.getElement(), "onCustomEvent");
        if (eventCustom != null)
        {
            context.getWidget().addCustomEventHandler(new CustomEventHandler()
            {
                public void onCustomEvent(CustomEvent event)
                {
                    Events.callEvent(eventCustom, event);
                }
            });
        }
    }
}
```

2.3.2.2.1 @TagEvents Annotation

The `@TagEvents` has one attribute (value) of type array of `@TagEvent`. Each `@TagEvent` is used to inform one event of the widget.

`@TagEvent` accepts one attribute of type `EvtBinder`. `EvtBinders` are classes that can automatically bind event declarations to widgets.

Crux offers `EvtBinders` for all GWT events and for custom ***Crux*** widgets events. The following table shows the GWT `EvtBinders`:

Event type	Class	Widget type
onBeforeSelection	BeforeSelectionEvtBind	HasBeforeSelectionHandlers<?>
onBlur	BlurEvtBind	HasBlurHandlers
onChange	ChangeEvtBind	HasChangeHandlers
onClick	ClickEvtBind	HasClickHandlers
onFocus	FocusEvtBind	HasFocusHandlers
onHighlight	HighlightEvtBind	HasHighlightHandlers<?>
onKeyDown	KeyDownEvtBind	HasKeyDownHandlers
onKeyPress	KeyPressEvtBind	HasKeyPressHandlers
onKeyUp	KeyUpEvtBind	HasKeyUpHandlers
onError	LoadErrorEvtBind	HasErrorHandlers
onLoad	LoadEvtBind	HasLoadHandlers
onMouseDown	MouseDownEvtBind	HasMouseDownHandlers
onMouseMove	MouseMoveEvtBind	HasMouseMoveHandlers
onMouseOut	MouseOutEvtBind	HasMouseOutHandlers
onMouseOver	MouseOverEvtBind	HasMouseOverHandlers
onMouseUp	MouseUpEvtBind	HasMouseUpHandlers
onMouseWheel	MouseWheelEvtBind	HasMouseWheelHandlers
onOpen	OpenEvtBind	HasOpenHandlers<?>
onScroll	ScrollEvtBind	HasScrollHandlers
onSelection	SelectionEvtBind	HasSelectionHandlers<?>
onShowRange	ShowRangeEvtBind	HasShowRangeHandlers<?>
onChange	ValueChangeEvtBind	HasValueChangeHandlers<?>

The annotation `@TagEventDeclaration` has one attribute of type `String`. This attribute only informs to ***Crux*** the name of the event. The parsing of the `<span>` element and the binding with the widget must be done programmatically, as shown on the previous example.

2.3.2.3 Factories Inheritance

The annotations present on `processAttributes` and `processEvents` methods of Widget factories super classes and implemented interfaces are also considered by ***Crux*** engine.

Let's take a look at the following example.

```
public interface HasNameFactory<T extends HasName>
{
    @TagAttributes({
        @TagAttribute("name")
    })
    void processAttributes(WidgetFactoryContext<T> context) throws InterfaceConfigException;
}
```

```
@DeclarativeFactory(id="myWidget", library="myLibrary")
```

```
public class MyWidgetFactory extends WidgetFactory<MyWidget> implements HasNameFactory<MyWidget>
{
        ...
   @Override
   @TagAttributes({
      @TagAttribute("myWidgetAttribute")
   })
   public void processAttributes(WidgetFactoryContext<MyWidget> context) throws InterfaceConfigException
   {
      super.processAttributes(context);
   }
}
```

That factory shown above will accept and automatically bind the attributes `myWidgetAttribute` and `name` to `MyWidget`.

Crux libraries use the inheritance support to turn easy the development of its factories. GWTWidgets library has interfaces that binds attributes and events for most of GWT basic interfaces for widgets. Just implementing some of those interfaces is enough to provide your factory with the ability of binding various attributes and events. The current existing interfaces are:

Interface	Attributes	Events
HasAllFocusHandlersFactory	-	onFocus, onBlur
HasAllKeyHandlersFactory	-	onKeyUp, onKeyPress, onKeyDown
HasAllMouseHandlersFactory	-	omMouseUp, onMouseDown, onMouseOver, onMouseOut, onMouseMove, onMouseWheel
HasAnimationFactory	animationEnabled	-
HasBeforeSelectionHandlersFactory	-	onBeforeSelection
HasChangeHandlersFactory	-	onChange
HasClickHandlersFactory	-	onClick
HasCloseHandlersFactory	-	onClose
HasDirectionFactory	direction	-
HasHighlightHandlersFactory	-	onHighlight
HasNameFactory	name	-
HasOpenHandlersFactory	-	onOpen
HasScrollHandlersFactory	-	onScroll
HasSelectionHandlersFactory	-	onSelection
HasShowRangeHandlersFactory	-	onShowRange
HasTextFactory	text	-
HasValueChangeHandlersFactory	-	onChange
HasWordWrapFactory	wordWrap	-

2.3.3 Children Processing

2.3.3.1 The processChildren method

2.3.3.2 WidgetChildProcessor

2.3.4 The postProcess method