Xap CodeTour

Tour of XAP Source Code

Getting Started

This tour assumes you are familiar with the architecture overview on the XAP web page

Codebase Layout

XAP code is located in the *codebase/src* folder of SVN. That in turn has three entries:
**dojo* - The latest dojo codebase that works with XAP. Dojo is a DHTML library that supports basic language functions like package declarations and object-oriented programming as well as a widget library.
**google* - The google dom/xpath implementation, modified with some bug fixes and clean up of variable scoping.
**xap* - Original code written by xap developers.

Namespace Handlers and Page Processing

Namespace handlers are a key component of the XAP architecture. A namespace handler is a piece of code that recognizes a certain namespace and does some processing based on those tags. When a XAL page is processed the following steps take place:

- The page is turned into an XML document by *xml/ParserFactory*
- Each immediate child of the root tag of the document is fed to an appropriate namespace handler for that namespace.
- The namespace handler performs some processing based on the XML structure of the tags in that namespace.

These namespace handlers include:

- *xml/xmodify/XmodifyNamespaceHandler.js* - Recognizes the xmodify namespace which is used to modify XML or HTML documents including the main UI document.
- *mco/McoNamespaceHandler.js* - Recognizes the mco namespace which is used to declare javascript code objects that will be used by the application.
- *macro/MacroNamespaceHandler.js* - Recognizes the macro namespace which is used to declare macro functions that will be used by the application.

In addition there are namespaces that provide data support, initial creation of the UI without the need for xmodify, etc.

Instantiating and Modifying the UI

XAP maintains a client-side UI document, which is an XML document that is similar to the HTML document saved by a web browser. Whenever this XML UI document is modified those modifications result in on-screen changes.

XML Structure Events

The XML documents used by XAP fire events to listeners when changes are made to elements within the document. Whenever a child is added or removed, or an attribute is changed, interested listeners will receive notifcation that the change took place.

Plugin Tag Mapping Files

The default tag mapping file is located in *taghandling/plugin.xml*. This file specifies how XML tags in the UI namespace map to classes that implement those tags.

UI Creation Workflow

An instance of *taghandling/PluginDocumentHandler.js* listens to the XML UI document events. Whenever a new tag is added to the UI document (using the DOM APIs or through the use of xmodify)this class will instantiate a **bridge class** based on the contents of the *plugin.xml* file. This bridge class is then responsible for modifying the HTML document to make appropriate onscreen changes.

Bridge Classes and Peer Objects

A bridge class is a class that connects an XML element to the HTML onscreen representation, typically though a peer class.

Consider the case of <window>. When the user adds a <window> tag to the XML document they expect a window to appear onscreen. The Dojo widget library offers a window implementation we can use. What is needed is a translation layer that can inspect the <window> tag and configure a Dojo window widget based on that tag.

A bridge class serves as this translation layer, and has two primary responsibilities:
*When an XML element is added to the UI document or changed in some way, the bridge class must create and maintain the widget implementation.
*When some event happens in the underlying widget, the bridge class must inform the framework of that event and potentially modify the XML DOM to reflect widget changes.

In this case consider a <textField> tag. If the user uses the DOM API to change the **text** attribute of the tag, we expect the onscreen text to change. In addition if the user types into the text field and hits enter we expect the **text** attribute of the XML element to reflect the new text, and we might also expect some event to fire informing us that the editing is finished.

Translating XML Changes to Peer Modifications

A typical bridge class listens for changes on the XML element it is responsible for, using the change notification events described previously. When an attribute on an XML element is changed dynamically the bridge class is notified and makes the appropriate calls to update the widget.

Responding to Widget Events

Just as a bridge class listens for XML events, it also listens for events on the underlying widget. When events occur on that widget the bridge class writes back into the XML element if needed, and fires events as well. How the bridge listens to the widget events depends on the widget itself. Standard dojo widgets can have listeners to specific methods added via **dojo.event.connect()**

Bridge Base Classes

There are a number of base classes to make bridge development easier.

- *tagHandling/AbstractTagImpl.js* - This is the base class for all bridges. It includes all the basic initialization logic, handling of attributes and children, and exposes helper methods to fire events and perform other common tasks.
- *bridges/basic/AbstractWidgetBridge.js* - This is the base class for all bridges that map to UI widgets. It handles basic attributes like **x**, **y** and **color** in a reasonable default fashion.
- *bridges/dojo/DojoWidgetBridge.js* - This is the base class for widgets based on the Dojo framework.

TextFieldBridge - An Example Dojo Widget Bridge

bridges/dojo/TextFieldBridge.js is a good example of a medium complexity bridge class. It does the following:
*Supports basic common attributes like position and coloring.
Supports text-specific attributes like *text,**maxLength** and **editable**
*Fires events and updates the XML DOM when the text in the onscreen widget is changed.

Supporting common attributes comes for free by extending DojoWidgetBridge. Supporting new attributes is a two step process:
Override *getNewAllowedAttributes() to supply a list of attributes specific to this widget that we recognize.
For each attribute, write a method in the form of *setTextAttribute() or **setMaxLengthAttribute** where the middle word(s) of the function name is the attribute. You can see that in the TextFieldBridge **setTextAttribute()** takes the argument, which is the new value of the **text** attribute, and passes it to a function on the widget itself that does the work. **setMaxLengthAttribute()** also calls a function on the widget but does some extra work to truncate the existing text if it is currently too long.

Supporting events is done in TextFieldBridge by using **dojo.event.connect()** to listen for keyup, textchange and mouseout on the underlying widget. When the underlying widget receives a keyup the **onTextChange()** method in TextFieldBridge is called. This method checks to make sure the text has actually changed and if so updates the text in the XML DOM using **writeBackAttribute()** and fires an event using **fireEvent()**, two methods provided by the base class.