Widget API - IDE/Builder

Adding a widget to the project - Builder

- Modify IDE\Builder\Index.html and add a line something like
  <script type="text/javascript" src="../Common/thingworx/widgets/*YourName*/*YourName*.ide.js"></script>

- Modify IDE\Builder\css\widgets.css and add a line something like the following line: (be sure not to exceed 31 lines in this file or IE will not load it properly. If we've hit 31 start another widget*{n}*.css)

  @import url('../../Common/thingworx/widgets/*YourName*/*YourName*.ide.css');

Widget Lifecycle in the IDE

- discovered (because of js being loaded into Index.html) and added to the widget toolbar/palette
  - **widgetProperties** is called to get information about each widget (e.g. display name and description)
  - **widgetEvents** is called to get information about the events each widget exposes
  - **widgetServices** is called to get information about the serviceseach widget exposes
- created (e.g. dragged onto a mashup panel)
  - **afterLoad** is called after your object is loaded and properties have been restored from the file, but before your object has been rendered
- appended to the workspace DOM element
  - **renderHtml** is called to get a HTML fragment that will get inserted into the mashup DOM element
  - **afterRender** is called after the HTML fragment representing the widget has been inserted into the mashup DOM element and a usable element ID has been assigned to the DOM element holding the widget content, the DOM element is now ready to be manipulated
- updated (e.g. resized, or updated via the widget property window)
  - **beforeSetProperty** is called before any property is updated
  - **afterSetProperty** is called after any property is updated
- destroyed (i.e. when it's deleted from the mashup)
  - **beforeDestroy** is called right before the widget's DOM element gets removed and the widget is detached from its parent widget and dellocated; this is the place to perform any clean-up of resources (e.g. plugins, event handlers) acquired

throughout the lifetime of the widget

API Provided for your Widget - IDE

- Calls/properties that runtime provides for a widget
    - this.**jqElementId**
        - this is the DOM element ID of your object after renderHtml
    - this.**jqElement**
        - this is the jquery element
    - this.**getProperty**(name)
    - this.**setProperty**(name,value)
        - Note: in the IDE every call to this will call afterSetProperty() if it's defined in your widget
- this**.updatedProperties**()
    - If you change the properties of your object at any point, please call this**.updatedProperties**() to let the Builder know that it needs to update the widget properties window, the connections window, etc.
- this.**getInfotableMetadataForProperty**(propertyName)
    - if you need the infotable metadata for a property you have bound, you can get it by calling this API … it returns undefined if you're not bound.
- this.**resetPropertyToDefaultValue**(propertyName)
    - this resets the named property back to whatever it's default value is.
- this.**removeBindingsFromPropertyAsTarget**(propertyName)
    - this removes any target data bindings from this propertyName … use this only when the user has initiated an action that invalidates that property.
- this.**removeBindingsFromPropertyAsSource**(propertyName)
    - this removes any source data bindings from this propertyName … use this only when the user has initiated an action that invalidates that property.
- this.**isPropertyBoundAsTarget**(propertyName)
    - this returns whether the property has been bound as a target. Most useful when trying to validate if a property has been either set or bound, for example the blog widgets validate() function:

```
    this.validate = function () {
        var result = [];
        var blogNameConfigured = this.getProperty('Blog');
        if (blogNameConfigured === '' || blogNameConfigured ===
undefined) {
            if (!this.isPropertyBoundAsTarget('Blog')) {
                result.push({ severity: 'warning',
                    message: 'Blog is not configured or bound for
             {target-id}' });
            }
        }
        return result;
    }
```

- this.**isPropertyBoundAsSource**(propertyName)
    - this returns whether the property has been bound as a source. Most useful when trying to validate if a property has been bound to a target, for example, the checkbox widgets validate() function:

```
this.validate = function () {
    var result = [];
    if (!this.isPropertyBoundAsSource('State')
         && !this.isPropertyBoundAsTarget('State')) {

        result.push({ severity: 'warning',
         message: 'State for {target-id} is not bound to any target'
        });
    }

    return result;
}
```

Callbacks from IDE to your Widget

- **widgetProperties**() [required]
    - returns a JSON structure defining the properties of this widget
    - required properties
        - **name** - the user-friendly widget name, as shown in the widget toolbar
    - optional properties
        - **description** - a description of the widget; used for tooltip
        - **iconImage** - file name of the widget icon image
        - **category** - an array of strings for specifying one or more categories that the widget belongs to (i.e. Common, Charts, Data, Containers, Components); enables the user to filter widgets by type/category
        - **isResizable** - true or false (default to true)
        - **defaultBindingTargetProperty** - name of the property to use as data/event binding target
        - **borderWidth** - if your widget provides a border, set this to the width of the border. This helps ensure pixel-perfect WYSIWG between builder and runtime.
            - If you set a border of 1px on the “widget-content” element at design time, you are effectively making that widget 2px taller and 2px wider (1px to each side). To account for this descrepancy, setting the borderWidth property will make the design-time widget the exact same number of pixels smaller. Effectively, this places the border “inside” the widget that you have created and making the width & height in the widget properties accurate.

- **isContainer** - true or false (default to false); controls whether an instance of this widget can be a container for other widget instances
- **customEditor** - name of the custom editor dialog to use for entering/editing the widget's configuration. If you put xxx, the system presumes you have created TW.IDE.Dialogs.xxx that conforms to the Mashup Widget Custom Dialog API (described in a separate document). We could support the ability to specify an array here as well where each entry would create an additional tab in the widget configuration dialog. For 1.1 we'll limit this to a string and only one custom configuration.
- **customEditorMenuText**: the text that will appear on the flyout menu for your widget as well as the hover text over the configure widget properties button. For example: 'Configure Grid Columns'.
- **allowPositioning** - optional, true or false (default to true)
- **supportsLabel** - optional, true or false (default to false); if true, the widget will expose a 'Label' property whose value will be used to create a text label that sits next to the widget in the IDE and runtime
- **properties** - a collection of property (attribute) objects; each property object can have...
    - property name :
        - **description** - a description of the widget; used for tooltip
        - **baseType**- the system base type name; in addition, if the **baseType** value is 'FIELDNAME', the widget property window will display a dropdown list that allows the user to pick from a list of fields available in the INFOTABLE bound to the **sourcePropertyName** value, based on the **baseTypeRestriction** specified; for an example, see the *TagCloud* widget implementation. Other special baseTypes:
            - STATEDEFINITION just picks a StateDefinition
            - STYLEDEFINITION just picks a StyleDefinition
            - RENDERERWITHSTATE will show a dialog and allow you to select a renderer and formatting that goes along with it. Note: you can set a default Style by putting the string with the default style name in the 'defaultValue'. Also, note that anytime your binding changes, you should reset this to the default value as in the code below:

```
    this.afterAddBindingSource = function (bindingInfo) {
        if (bindingInfo['targetProperty'] === 'Data') {

this.resetPropertyToDefaultValue('ValueFormat');
        }
    };
```

        - STATEFORMATTING will show a dialog and allow you to pick either fixed style or a state-based style. Note: you can set a default Style by putting the string with the default style name in the ‘defaultValue’. Also, note that anytime your binding changes, you should reset this to the default value as in the code above for RENDERERWITHSTATE.

        - VOCABULARYNAME will just pick a DataTags vocabulary at the moment
    - **mustImplement** - if the baseType is THINGNAME, and you specify “mustImplement” the IDE will restrict to popups implementing the specified EntityType and EntityName [by calling QueryImplementingThings against said EntityType and EntityName]

```
'baseType': 'THINGNAME',
'mustImplement' : {
   'EntityType' : 'ThingShapes',
   'EntityName' : 'Blog'
},
```
    - **baseTypeInfotableProperty** - if baseType is RENDERERWITHFORMAT, baseTypeInfotableProperty specifies which property’s infotable is used for configuration
    - **sourcePropertyName** - when the property’s baseType is ‘FIELDNAME’, this attribute is used to determine which INFOTABLE’s fields are to be used to populate the FIELDNAME dropdown list; for an example, see the *TagCloud* widget implementation
    - **baseTypeRestriction** - when specified, this value is used to restrict the fields available in the FIELDNAME dropdown list; for an example, see the *TagCloud* widget implementation
    - **tagType** - if the baseType is ‘TAGS’ this can be ‘DataTags’ or ‘ModelTags’ … defaults to DataTags
    - **defaultValue** - default undefined; used only for ‘property’ type
    - **isBindingSource** - true or false; allows the property to be a data binding source, default to false
    - **isBindingTarget** - true or false; allows the property to be a data binding target, default to false
    - **isEditable** - true or false; controls whether the property can be edited in the IDE, default to true

        - **isVisible** - true or false; controls whether the property is visible in the properties window, default to true
        - **selectOptions** - an array of value / (display) text structures
            - Example: [{ value: ‘optionValue1’, text: ‘optionText1’}, { value: ‘optionValue2’, text: ‘optionText2’}]
        - **warnIfNotBoundAsSource** - true or false; if true, then the property will be checked by the IDE for whether it’s bound and generate a to-do item when it’s not
        - **warnIfNotBoundAsTarget** - true or false; if true, then the property will be checked by the IDE for whether it’s bound and generate a to-do item when it’s not
- **afterLoad**() [optional]
    - called after your object is loaded and properties have been restored from the file, but before your object has been rendered
- **renderHtml**() [required]
    - returns HTML fragment that the IDE will place in the screen; the widget’s content container (e.g. div) must have a ‘widget-content’ class specified, after this container element is appended to the DOM, it becomes accessible via **jqElement** and its DOM element id will be available in **jqElementId**
- **widgetEvents**() [optional]
    - a collection of events; each event can have...
        - property name:
            - **warnIfNotBound** - true or false; if true, then the property will be checked by the IDE for whether it’s bound and generate a to-do item when it’s not
- **widgetServices**() [optional] **[2.1+]**
    - a collection of services; each service can have...
        - property name:
            - **warnIfNotBound** - true or false; if true, then the property will be checked by the IDE for whether it’s bound and generate a to-do item when it’s not
- **afterRender**() [optional]
    - called after we insert your html fragment into the dom
- **beforeDestroy**() [optional]
    - called right before the widget’s DOM element gets removed and the widget is detached from its parent widget and dellocated; this is the place to perform any clean-up of resources (e.g. plugins, event handlers) acquired throughout the lifetime of the widget
- **beforeSetProperty**(name,value) [optional] [IDE only - not at runtime]
    - called before any property is updated within the IDE, this is a good place to perform any validation on the new property value before it is committed;
    - if a message string is returned, then the message will be displayed to the user, and the new property value will not be committed

- **afterSetProperty**(name,value) [optional] [IDE only - not at runtime]
    - called after any property is updated within the IDE
    - return true to have the widget re-rendered in the IDE
- **afterAddBindingSource**(bindingInfo) [optional]
    - whenever data is bound to your widget, you will called back with this (if you implement it … it's optional)
    - The only field in bindingInfo is targetProperty which is the propertyName that was just bound
- **validate**() [optional]
    - called when the IDE refreshes its to-do list;
    - the call must return an array of result object with *severity* (optional and not implemented) and *message* (required) properties;
    - the message text may contain one or more pre-defined tokens, such as {target-id}, which will get replaced with a hyperlink that allows the user to navigate/select the specific widget that generated the message
    - for example:

```
this.validate = function () {
    var result = [];
    var srcUrl = this.getProperty('SourceURL');
    if (srcUrl === '' || srcUrl === undefined) {
        result.push({ severity: 'warning', message: 'SourceURL is not defined
        for {target-id}' });
    }
    return result;
}
```

Tips

- Use **this.jqElement** to limit your element selections, this will reduce the chance of introducing unwanted behaviors in the the application when there might be duplicate IDs and/or classes in the DOM.
    - Don't do...
        - $('.add-btn').click(function(e) { ...do something... });
    - Do...
        - this.jqElement.find('.add-btn').click(function(e) { ...do something... });
- **widget.properties** - only store actual properties that you want saved and loaded with your object. For example, do not use this area to store a reference to a 3rd-party

component … the Builder literally stores the entire widget.properties as a JSON structure in the mashup and both the Builder and Runtime load this JSON structure.

- **Logging** - we recommend that you use the following methods to log in the Widget IDE and Runtime environment:
    - TW.log.trace(message[, message2, ...][, exception])
    - TW.log.debug(message[, message2, ...][, exception])
    - TW.log.info(message[, message2, ...][, exception])
    - TW.log.warn(message[, message2, ...][, exception])
    - TW.log.error(message[, message2, ...][, exception])
    - TW.log.fatal(message[, message2, ...][, exception])

  You can view the log messages in the Mashup IDE by opening the log window via the Help>Log menu item; in the mashup runtime, you can now click on the "Show Log" button on the top left corner of the page to show log window. If the browser you use supports console.log(), then the messages will also appear in the debugger console.
- **Clicks** - if your widget is not able to be selected, your UI may be “swallowing” clicks. To counteract this, include this line at the end of your code in afterRender():

```
thisWidget.jqElement.find('*').unbind('click');
thisWidget.jqElement.find('*').bind('click', function (e) {
        thisWidget.jqElement.click();
        e.stopPropagation();  // this stops the event from bubbling
                              // up the DOM tree
});
```