Widget API - Runtime

Adding a widget to the project - Runtime

- Modify IDE\Runtime\Index.html and add a line something like

```
<script type="text/javascript" src="../Common/thingworx/widgets/YourName/YourName.runtime.js"></script>
```

- Modify IDE\Squeal\js\thingworx.runtime.squeal.v2.js and add a line something like

```
.script('../Common/thingworx/widgets/YourName/YourName.runtime.js')
```

- Modify IDE\Runtime\css\widgets.css and add a line something like the following: (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.runtime.css');
```

Widget Lifecycle in the Runtime

- Whenever a mashup that contains one of your widgets, you will first be called with **runtimeProperties**, which is optional and only contains a few definitions.
- The property values that were saved in the mashup definition will be loaded into your object without your code being called in any way
- After your widget is loaded but before it's put on to the screen, the runtime will call **renderHtml** where you return the HTML for your object. The runtime will render that HTML into the appropriate place in the DOM
- Immediately after that HTML is added to the DOM, you will be called with **afterRender**. This is the time to do the various jquery bindings (if you need any). It is only at this point that you can reference the actual DOM elements and you should only do this using code such as

```
var widgetElement = this.domElement; // note that this is a jquery object
```

  This is important because the runtime actually changes your DOM element ID and you should never rely on any id other than the id returned from this.domElementId)
- If you have defined an event that can be bound, whenever that event happens you should call

```
var widgetElement = this.domElement;
widgetElement.triggerHandler('Clicked'); // change 'Clicked' to be whatever your event name
                                         is that you defined in your runtimeProperties that people bind to
```

- If you have any properties bound as data targets, you will be called with **updateProperty**. You are expected to update the DOM directly if the changed property affects the DOM - which is likely, otherwise why would the data be bound :)

- If you have properties that are defined as data sources and they're bound you can be called

with **getProperty_*{propertyName}***() … if you don't define this function, the runtime will simply get the value from the property bag.

API Provided for your Widget - Runtime

- Calls/properties that runtime provides for a widget
  - this.**domElementId**
    - 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)
  - this.**updateSelection**(propertyName,selectedRowIndices)
    - call this anytime your widget changes selected rows on data bound to a certain propertyName … e.g. in a callback you have for an event like onSelectStateChanged you'd call this API and the system will update any other widgets relying on selected rows

Callbacks from the Runtime to your Widget

- **runtimeProperties**() [optional]
  - returns a JSON structure defining the properties of this widget
  - optional properties
    - **isContainer** - true or false (default to false); controls whether an instance of this widget can be a container for other widget instances
    - **needsDataLoadingAndError** - true or false (defaults to false) - set to true if you want your widget to display the standard 25% opacity when no data has been received and turn red when there is an error retrieving data
    - **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
- **renderHtml**() [required]
  - returns HTML fragment that the runtime 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 **jqElementIdIo**
  -
- **afterRender**() [optional]
  - called after we insert your html fragment into the dom
  - use this.domElementId to find the DOM element ID
  - use this.**jqElement** to use the jQuery reference to this dom element
- **beforeDestroy**() [optional but highly recommended]
  - this is called anytime your widget is unloaded, this is the spot to...
    - unbind any bindings
    - clear any data set with .data()
    - destroy any 3rd party libraries or plugins, call their destructors, etc.
    - free any memory you allocated or are holding on to in closures, by setting the variables to *null*
    - you do not need to destroy the DOM elements inside your widget, they will be destroyed

for you by the runtime

- **handleSelectionUpdate**(propertyName, selectedRows, selectedRowIndices)
  - called whenever selectedRows has been modified by the data source you're bound to on that (PropertyName. selectedRows is an array of the actual data and selectedRowIndices is an array of the indices of the selected rows
- **serviceInvoked**(serviceName) **[2.1+]**
  - serviceInvoked() is called whenever a service you defined is triggered.
- **updateProperty**(updatePropertyInfo)
  - updatePropertyInfo is an object with the following JSON structure

{
DataShape: *metadata for the rows returned*,
ActualDataRows: *actual Data Rows*
SourceProperty: *SourceProperty*
TargetProperty: *TargetProperty*
RawSinglePropertyValue: *value of SourceProperty in the first row of ActualDataRows*,
SinglePropertyValue: *value of SourceProperty in the first row of ActualDataRows converted to the defined baseType of the target property [not implemented yet]*,
SelectedRowIndices: *an array of selected row indices.*
SourceDetail: *either "AllData" or "SelectedRows" - for some widgets, it's important tthiso know which of these situations you're in*

}

For each data binding, your widget's updateProperty() will be called each time the source data is changed. You need to check updatePropertyInfo.TargetProperty to determine what aspect of your widget should be updated. Here is an example from thingworx.widget.image.js ...

```
this.updateProperty = function (updatePropertyInfo) {
    // get the img inside our widget in the DOM
    var widgetElement = $('#' + this.domElementId + ' img');

    // if we're bound to a field in selected rows and there are no selected rows,
    //  we'd overwrite the default value if we didn't check here
    if (updatePropertyInfo.RawSinglePropertyValue !== undefined) {

        // see which TargetProperty is updated
        if (updatePropertyInfo.TargetProperty === 'sourceurl') {
            // SourceUrl updated - update the <img src=
            this.setProperty('sourceurl', updatePropertyInfo.SinglePropertyValue);
            widgetElement.attr("src", updatePropertyInfo.SinglePropertyValue);

        } else if (updatePropertyInfo.TargetProperty === 'alternatetext') {
            // AlternateText updated - update the <img alt=
            this.setProperty('alternatetext', updatePropertyInfo.SinglePropertyValue);
            widgetElement.attr("alt", updatePropertyInfo.SinglePropertyValue);
        }
    }
```

```
};
```

Note that we set a local copy of the property in our widget object as well so that if that property is bound as a data source for a parameter to a service call (or any other binding) - the runtime system can simply get the property from the property bag. Alternatively, we could supply a custom getProperty_{propertyName} method and store the value some other way.

- **getProperty_*{propertyName}***()
  - anytime that the runtime needs a property value, it checks to see if your widget implements a function to override and get the value of that property. This is used when the runtime is pulling data from your widget to populate parameters for a service call.

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... });`

- **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.

- **Formatting** - if you have a property with baseType of STYLEDEFINITION, you can get the style information by calling

```
var formatResult = TW.getStyleFromStyleDefinition(widgetProperties['PropertyName']);
```

If you have a property of baseType of STATEFORMATTING

```
var formatResult = TW.getStyleFromStateFormatting({
      DataRow: row,
      StateFormatting: thisWidget.properties['PropertyName']
```

```
});
```

In both cases formatResult is an object with the following defaults:

```
{
     image: '',
     backgroundColor: '',
     foregroundColor: '',
     fontEmphasisBold: false,
     fontEmphasisItalic: false,
     fontEmphasisUnderline: false,
     displayString: '',
     lineThickness: 1,
     lineStyle: 'solid',
     lineColor: '',
     secondaryBackgroundColor: ''
 };
```

If you need to get information about the states defined, you can call **TW.getStateInformation**(stateFormattingProperty) ...

It returns an array of structures with the following fields:

isDefault [true or false]
stateDefinitionType [‘fixed’, ‘numeric’, or ‘string’] Note: this will be the same for every item in the array
styleDefinition [name of the style … can call `TW.getStyleFromStyleDefinition()` as described above
comparator [only used if not default … either ‘==’, ‘<’ and ‘<=’]
value [value used for the state]

Here is a sample call that I added within tagCloud (but didn't check in)

```
var test = TW.getStateInformation(widget.properties['TagStateStyle']);
TW.log.info('# states: ' + test.length.toString());
for (var i = 0; i < test.length; i++) {
   TW.log.info(' '
          + i.toString()
          + ': isDefault:' + test[i].isDefault
          + ', stateDefinitionType:'  + test[i].stateDefinitionType
          + ', comparator:' + test[i].comparator
          + ', value:' + test[i].value);
}
```