ThingWorx

Creating Widgets for ThingWorx

NOTE: this document presumes you're familiar with HTML, JavaScript and jquery.

Creating a new widget for ThingWorx is incredibly straightforward and powerful. Any capability available from JavaScript and HTML can be encapsulated and created as a widget within ThingWorx. Every widget in ThingWorx's Mashup environment was built with the same SDK we deliver to you.

For purposes of demonstration we'll create a simple "Hello World" widget that accepts a bindable "DisplayText" and adds that text to "Hello World".

Creating a widget requires to you have 4 files:

- *helloWorldWidget*.ide.js
- *helloWorldWidget*.ide.css
- *helloWorldWidget*.runtime.js
- *helloWorldWidget*.runtime.css

Adding a new widget requires you to have it available within the Mashup Builder. This is done by packaging the widget and importing is as an extension package, please refer to the ThingWorx Extensibility document for information on facilitating.

Builder Code

The ide.js has a few simple methods you must implement. The widgets are declarative. There are 3 functions where you declare your widget's properties, services and events. For example, to develop a widget with a bindable string property "DisplayText" that is used to update your HTML… this is all the javascript that's required to declare those properties:

```
TW.IDE.Widgets.helloWorldWidget = function () {

    this.widgetProperties = function () {

        return {

            'name': 'Hello World Widget',

            'description': 'Demonstration widget',

            'category': ['Common'],

            'properties': {

                'DisplayText': {

                    'baseType': 'STRING',

                    'defaultValue': 'whatever default you want',

                    'isBindingTarget': true

                }

                   // add any additional properties here

            }

        };

    };

       ...

};
```

Runtime Code

To handle this widget at runtime, you need to have some methods available to:

- render the HTML you'd like at runtime
- set up any bindings you need to after rendering the HTML
- handle property updates

The runtime code must be in the runtime.js file. Here is the runtime code necessary to handle what we've described above:

```
TW.Runtime.Widgets.helloWorldWidget = function () {

    var valueElem;

    this.renderHtml = function () {

            // return any HTML you want rendered for your widget

            // If you want it to change depending on properties that the user

            // has set, you can use this.getProperty(propertyName).  In

            // this example, we'll just return static HTML

            return

                    '<div class="widget-content widget-helloWorldWidget">' +

                            '<span class="whatever">' +

                                    'Hello World' +

                            '</span>' +

                            '<span class="whatever">' +

                            '</span>' +

                    "</div>';

    };


    this.afterRender = function () {

            // NOTE: this.jqElement is the jquery reference to your html dom element

            //       that was returned in renderHtml()


            // get a reference to the value element

            valueElem = this.jqElement.find('.whatever');
```



```
            // udpate that DOM element based on the property value that the user set
            // in the mashup builder
            valueElem.text(this.getProperty('DisplayText'));
    };


    // this is called on your widget anytime bound data changes
    this.updateProperty = function (updatePropertyInfo) {
        // TargetProperty tells you which of your bound properties changed
        if (updatePropertyInfo.TargetProperty === 'DisplayText') {
            valueElem.text(updatePropertyInfo.SinglePropertyValue);
            this.setProperty('DisplayText', updatePropertyInfo.SinglePropertyValue);
        }
    };
};
```

Additional Features

Of course, this is a very simple example … other functionality you can incorporate into your widgets:

- **Services** that can be bound to events (e.g. Click of a button, Selected Rows Changed or Service Completed)
- **Events** that can be bound to various services (invoke a service, navigate to a mashup, etc.)
- **Properties** can be bound out as well (this example only showed properties being bound in)

Within your widget code at runtime, you can access the full power of javascript and HTML … virtually anything that can be done using HTML and JavaScript is available within your widget.

Summary

That's it. Now your widget will be available in the Mashup Builder and you can use it in your mashups. Using the ThingWorx Extensibility Toolkit you can make your widget available to any project you work on.

There are additional documents with all the details for

- ThingWorx Widget SDK - Composer
- ThingWorx Widget SDK - Runtime
- ThingWorx Extensibility

ThingWorx

Widget API - Composer/Builder

Widget Lifecycle in the Composer

- 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 - Composer

- 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 Composer 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 Composer 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 Composer 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 Composer 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 Composer, 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 Composer 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 Composer 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 Composer 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 Composer 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 Composer 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] [Composer only - not at runtime]
    - called before any property is updated within the Composer, 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] [Composer only - not at runtime]
    - called after any property is updated within the Composer
    - return true to have the widget re-rendered in the Composer
- **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 Composer 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 Composer 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 Composer 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
});
```

ThingWorx

Widget API - Runtime

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 **jqElementIdlo**
  -
- **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 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 Composer 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 Composer 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);
}
```