Custom ( aka Composite ) properties

# What are they?

Custom properties are typically “composed” of real data model properties; hence they are also referred to as composite properties. *(To prevent confusion, from here on in this document they are referred to as composite properties. )*  They do not hold a value, but are derived from one or more real data model property instances. Data model properties are defined in the XML.

# History

They were originally introduced in the GUI to make the presentation layer easier to understand for the end user. Configuration objects are displayed in a grid control. And since properties are bound to the grid control, this makes it much easier to implement this kind of complicated data model.

In a simple example, an IPv4 Address on EmulatedDevice is implemented as a child Ipv4If object. The Ipv4If class has an Address property. The user may not care to know the full underlying data model details when configuring a test from the GUI. The user just wants to configure the IP address on the device.

In a more complex example, the user wants to understand what interfaces are defined on a particular device. For this, we build up a string representation of the underlying interface stack to display in the grid.

# New implementation. ( Why they are being moved to the BLL )

A problem with the GUI implementation is that to perform operations on these properties, those operations must be performed in the GUI. This is not a good solution for scripting automation users, since there would need to be a dual implementation to get the same functionality in the BLL. So, composite properties are being moved to the BLL on a per need basis. **No new composite properties should be introduced in the GUI. Put them in the BLL.**

Things to know about composite properties:

1. They are not serializable. They do not need to be since they are really an alias for other properties.
2. They **do** use memory in the GUI proxy objects. ( unlike the old implementation that used .NET properties. )
3. There is a framework for managing these properties.

# How to create a composite property

1. Define the property in the data model XML with category="composite” Example: <stc:property name="Ipv4Address" type="ip" default="192.85.1.3" category="composite" nullable="true" displayName="IPv4 Address"/>
2. Register composite property handlers with the CompositePropertyManager. Only 1 is absolutely required.
   1. Register the **setter** delegate via CompositePropertyManager ::RegisterSetterDelegate ( Not needed for read only properties. )
   2. Register the **getter** delegate via CompositePropertyManager::RegisterGetterDelegate
   3. Register the **Is Applicable** delegate via CompositePropertyManager:: RegisterIsApplicableDelegate ( Only required for nullable=”true” properties )
3. Register the re-evaluation trigger via CompositePropertyManager::RegisterComposedOfDefinition **( Most common case )**

The composite property framework needs to know what your property is made up of so that when those properties change, the composite property can be updated and that value reflected back in the GUI. Do this by registering a vector of ComposedOfDef\_t which tells the framework what class id, property id, and relationships make up the property. For most cases, this method will work.

For corner cases that require special attention, you may have to handle the trigger yourself. There are examples of both cases as well as examples of set/get/isapplicable delegates in

EmulatedDeviceCompositePropertiesHandler:: RegisterEmulatedDeviceCompositePropertyHandlers in EmulatedDeviceCompositePropertiesHandler.cpp

Note: If there is a common pattern that is identified that arises from a corner case, please consider moving it to the framework.