Data Binding

Description

DataBinding is the Courier framework mechanism to automatically distribute data items to interested consumers. The owner of data is the model component. It is the master in data distribution and sends and receives both data and control messages.

Data Binding Fundamentals

Binding Source

Data is organized in so called binding sources. A binding source is a group of logically related data items. The data items in a binding source can be hierarchically organized to compound data types and also supports lists natively. Data items can be of any native C/C++ type.

Data items might be grouped under the following aspects to binding sources

- **logical dependencies**
  If data items have logical dependencies, these data items shall be put into the same binding source. In the data model of a radio station the station name is logically related to the frequency of the station. Both data items should only be updated once to avoid an inconsistent visualization of data.
- **update frequency**
  data with equal update rates shall be grouped in a binding source. The speed data item should be put into the same binding source as the odometer data item. The update rate of both items is very different and the odometer value would be distributed in an unnecessary high data rate. Always the complete set of data in a binding source is updated.
- **data size**
  Binding source data size shall be balanced. Defining too small binding sources creates overhead in the data binding system (a reference to a Courier::DataItemMsg is stored in every component that receives the message). Defining too large binding sources might imply copying unnecessary data (as all binding source data needs to be copied to the Courier::DataItemMsg, the number of unchanged data that needs to be copied increases with the binding source data size).

A binding source may contain any number of

- Elementary data items which can be
  - scalar items of any C / C++ type

    - lists of elementary or compound items
- Compound data items (complex C / C++ types) which may contain again
    - Compound data items
    - Elementary data items

All data items in binding sources will be updated in an according Courier::DataItemMsg. Thus it is guaranteed that the data contained in a binding source will always be self consistent.

Bindings can only be defined on elementary data items and list items. It is not possible to define a binding on compound data items or single list items.

Binding sources are defined with the Courier XML Data Definition Language (XCDDL). From the data definition, source code is generated by CourierGenerator tool. XCDDL is also used by CGI Studio SceneComposer. Bindings between widget properties and data items can be specified in SceneComposer and will be automatically instantiated by Courier at runtime.

Data Item Keys

Courier::DataItemKey is the central identifier for data items in data binding. The keys are generated by code generation and can be used to address a specific data item in a binding source.

```
// bind SpeedWidget::Speed property to /CarStatus/Speed
ok = ok && CreateBinding(::Generated::ItemKey::CarStatus::SpeedItem,
                         "SpeedWidget",
                         "BindableSpeed",
                         widget);
```

Courier::DataItemKey is the most efficient way to address a data item. Nevertheless there are two additional addressing schemes supported by data binding.

- ASCII string path e.g. /CarStatus/Speed
- Hash values calculated from the fully qualified ASCII string path

Normal application developers will never encounter the need to use one of the alternative addressing schemes. Hash values are used to serialize binding information into the asset library and used internally only.

Data Flow

Data is asynchronously distributed in the system with messages (see Courier::DataItemMsg). Currently this is the only supported data access mechanism. Components receive data in Courier::DataItemMsg messages. The components will hold a reference to the last received Courier::DataItemMsg for each binding source. Thus components can read the latest data set synchronously from the message. If e.g. the view component opens a new View, the data for bound

widget properties can be set immediately from the stored Courier::DataItemMsg messages accordingly.

If the model receives new data, it will send the new set of data with a new Courier::DataItemMsg. The message not only carries modified, new data, but the complete set of the binding source data. For each data item the message carries a flag indicating if the item has been modified or not. For non-list data a single flag is sufficient information to indicate a change in data, for list data multiple types of changes may happen (e.g. adding, removing, modifying existing list items). Therefore additional information is necessary to inform controller and view about the nature of change. Courier::Internal::ListEventMsg is used to send list modification information from model to controller and view. Courier::ListEventNotifier take over the job of sending Courier::Internal::ListEventMsg.

Components (controller / view) can also modify data items sent by the model. Again modified data is sent back in messages to the model. The Courier::UpdateModelMsg messages are responsible to carry modified data items back to the model component. Courier::UpdateModelMsg may carry multiple requests to change or update data for multiple binding sources.

On reception of an UpdateModelMsg the model component inspects the requests and, if the requests are accepted, modifies model data accordingly.

Revision Numbers

All data binding messages carry a revision number of the data. The Courier framework maintains a revision number for each binding source data. Every time a Courier::DataItemMsg is sent by the model, the according binding source revision number will be incremented. Requests from other components towards the model carry the revision number of the data the request has been issued with. If e.g. a widget requests to delete a list item, the model can check if the delete request has been made on the latest version of the data. If the revision number of the request is smaller than the revision number maintained in the model, the request might be ignored because widget and model have an inconsistent data state (see Courier::BindingSourceRevisionStore).

List Handling

DataBinding supports lists. As list data may be too large to send in messages, data binding supports fragmentation of lists. With fragmentation only list fragments of a defined number if list items are sent. Properties that can receive lists must be defined with Courier::ListPropertyType. Following property definition defines a property to receive list types:

```
// property excepting lists of type TString
CdaBindableProperty(MenuEntries, Courier::ListPropertyType<FeatStd::String>
, GetMenuEntries, SetMenuEntries)
CdaDescription("Test for properties bound to lists")
CdaCategory("List Test Properties")
```

```
CdaBindablePropertyEnd()
```

The Courier::ListPropertyType implements the Courier::AsyncListInterface which the widget can use to request data from the list. The interface is - as the name suggests - asynchronous. The widget sends requests towards the list and the list responds to the requests with Courier::ListEvent. To receive list events, the widget has to register an event callback (see Courier::ListEventHandler).

See List Handling for more details.

Type Conversion

To establish a binding between a data item and a widget property, the data item type must match exactly the property type. To overcome that limitation, data binding defines the concept of type converters (see Courier::TypeConverter). Type conversion works both for list and non-list data.

See chapter Type Converter for further information.

Data Binding Cook Book

Make Application Ready for Data Binding

In order to make your application ready for data binding, the following actions must be performed:

Make Widget Properties Data Binding Aware

Widget properties have to be defined with CdaBindableProperty instead of CdaProperty. This will enable the property for data binding. Making a property bindable adds a small overhead to the property definition (approximately 4 bytes RAM per property - shared by all property instances).

If your widget is going to change the property value (e.g. a text input widget that changes the Text property), data binding must be informed about the change by using the PropertyModified method.

Widget properties handling lists are explained in List Handling

Define Data Binding Model with XCDDL

Next is to define the application data model with XCDDL language - see Courier Data Definition Language XCDDL.

Implement the Application Model Component

See Model Component Sample Implementation for further details.

Programmatically Establish Bindings

Bindings between widget properties and data items are normally defined in SceneComposer.

In order to do this, the specification file (XCDL or a XCDL derived description file) must be set. This can be set in SceneComposer by going to *File/Define application specification* and setting the following:

- Specification: XCDL or a XCDL derived description file (e.g. *BindingSources.xcddl* for *CourierSampleApp*)
- Search path: *<cgi_studio_root>/cgi_studio_courier/src*

After these settings are made, the **Binding Property** of the widget must be set according to the requirements of the application.

Nevertheless bindings may also be implemented in the application code. Courier::CreateBinding does the trick.

```
// bind SpeedWidget::Speed property to /CarStatus/Speed
ok = ok && CreateBinding(::Generated::ItemKey::CarStatus::SpeedItem,
                         "SpeedWidget",
                         "BindableSpeed",
                         widget);
```

> Bindings exist in the defined thread context. A binding from a data item to a widget property exists in the context of the view component and the thread the view component is executed in. Therefore programmatic definition of bindings must be done in the view component context only.

Courier Data Definition Language XCDDL

XCDDL fundamentals

XCDDL data definition contains compound data type and binding source definitions. Each distinct data entity in the binding source definition is called a *data item*. Data items can be of any C / C++ type as long as this type obeys certain criteria:

- the type must define a default constructor
- the type must be copyable (copy c-tor and assignment operators with public access)

- data binding has to perform certain type inspection on the data item types. Therefore the type shall not have protected or private inheritance for a base class

The following XCDDL snippet defines a RadioStationData type based on a very simplified radio station model.

```
        <compound name="RadioStationData" uid="{8B61567E-F90E-4828-8EEE-E4FBD82515BD}">
            <comment>
                <![CDATA[]]>
            </comment>
            <item name="Name" type="RadioStationName" readonly="true" uid="{0EB7E841-CA1B-
4B56-A755-0738924DA739}"/>
            <item name="Freq" type="Courier::Float" readonly="false" defaultValue="97.0"
uid="{DDD5EFDA-EAAF-4B89-846D-6E70405C1B9D}"/>
            <item name="HasTMC" type="bool" readonly="true" defaultValue="false"
uid="{8362FDC9-2C25-4986-A21B-3815F1B4344C}"/>
            <item name="Band" type="Band::Enum" validateType="false" readonly="false"
defaultValue="Band::FM" uid="{6889C212-C0CE-4031-909D-9E5C7E07ADFA}"/>
        </compound>
```

The compound RadioStationData type is subsequently used to define a binding source Radio:

The binding source contains a data item for the current station, current TMC messages supplying station, and a list of preset stations.

```
        <bindingSource name="Radio" readonly="true" access="asynchronous" uid="{5361E13F-B66C-
4CC9-97A7-B17CC615C195}">
            <comment>
                <![CDATA[]]>
            </comment>
            <item name="CurrentStation" type="RadioStationData" uid="{4FE2D8D6-E76F-45BE-BD22-
01BC7A1352C9}">
                <comment>
                    <![CDATA[]]>
                </comment>
            </item>
            <item name="CurrentTMCStation" type="RadioStationData" uid="{d8e35b4b-b236-4814-
bc04-4274245174d0}">
                <comment>
                    <![CDATA[]]>
                </comment>
```

```
            </item>
            <list name="Presets" type="RadioStationData" maxCount="6" allowEdit="true"
allowRemove="true" allowAdd="true" uid="{BFCB2576-6DC0-44B5-877A-4E8DF0DD0A70}">
                <comment>
                    <![CDATA[]>
                </comment>
            </list>
        </bindingSource>
```

Please refer to XCDDL schema documentation for further details.

Sample data definition

```
<definition>
    <!--                                                        from namespace /Generated -->
    <include href="../../../../cgi_studio_courier/src/Courier/Platform/Types.xcdl" />
    <generator generate="true" location="DataModel">
        <namespace namespace="Generated"/>
        <header>
            <prolog file="../../../../cgi_studio_courier/CourierCopyright.txt" />
            <include file="FeatStd/Util/String.h"/>
            <include file="DataModel/BindingSourcesTypes.h"/>
      <include file="DataModel/RadioTypes.h"/>
          <include file="FeatStd/Util/StringBuffer.h" />
        </header>
        <source>
          <prolog file="../../../../cgi_studio_courier/CourierCopyright.txt" />
        </source>
    </generator>

    <types>
        <!--
        <compound name="RadioStationData" uid="{8B61567E-F90E-4828-8EEE-E4FBD82515BD}">
            <comment>
                <![CDATA[]>
            </comment>
            <item name="Name" type="RadioStationName" readonly="true" uid="{0EB7E841-CA1B-
4B56-A755-0738924DA739}"/>
            <item name="Freq" type="Courier::Float" readonly="false" defaultValue="97.0"
uid="{DDD5EFDA-EAAF-4B89-846D-6E70405C1B9D}"/>
            <item name="HasTMC" type="bool" readonly="true" defaultValue="false"
```

```
uid="{8362FDC9-2C25-4986-A21B-3815F1B4344C}"/>
            <item name="Band" type="Band::Enum" validateType="false" readonly="false"
defaultValue="Band::FM" uid="{6889C212-C0CE-4031-909D-9E5C7E07ADFA}"/>
        </compound>
        <!--
    </types>

    <bindingSources>
        <!--
        <bindingSource name="Radio" readonly="true" access="asynchronous" uid="{5361E13F-B66C-
4CC9-97A7-B17CC615C195}">
            <comment>
                <![CDATA[]]>
            </comment>
            <item name="CurrentStation" type="RadioStationData" uid="{4FE2D8D6-E76F-45BE-BD22-
01BC7A1352C9}">
                <comment>
                    <![CDATA[]]>
                </comment>
            </item>
            <item name="CurrentTMCStation" type="RadioStationData" uid="{d8e35b4b-b236-4814-
bc04-4274245174d0}">
                <comment>
                    <![CDATA[]]>
                </comment>
            </item>
            <list name="Presets" type="RadioStationData" maxCount="6" allowEdit="true"
allowRemove="true" allowAdd="true" uid="{BFCB2576-6DC0-44B5-877A-4E8DF0DD0A70}">
                <comment>
                    <![CDATA[]]>
                </comment>
            </list>
        </bindingSource>
        <!--

        <bindingSource name="CarStatus" readonly="true" access="asynchronous" uid="{7F571FF5-
F3F8-49DA-977F-745A4FAC1A7C}">
            <comment>
                <![CDATA[]]>
            </comment>
```

```
            <item name="Speed" type="Courier::UInt32" defaultValue="0" readonly="false"
uid="{FBCADE39-6065-4B11-8F4E-D49C64DAD4B7}"/>
            <item name="RPM" type="Courier::UInt32" defaultValue="0" readonly="true"
uid="{9A5D1EE6-DA77-49A9-B303-E958100C5CB2}"/>
        </bindingSource>

        <bindingSource name="TestValues" readonly="false" access="asynchronous"
uid="{03ee17dc-5693-41a7-88f5-6407040b8e3d}">
            <comment>
                <![CDATA[]]>
            </comment>
            <item name="TestValue" type="Courier::Int32" defaultValue="0" readonly="false"
uid="{70664678-C981-42AA-BE66-626834205C73}"/>
            <list name="TestValueList" type="Courier::UInt8" maxCount="10" allowAdd="false"
allowEdit="false" allowRemove="false" uid="{6629b4a5-1618-4fea-a8c0-f1414424d2d4}"/>
        </bindingSource>

        <bindingSource name="SettingsMenu" access="asynchronous" uid="{6a240f30-285e-480c-
9231-4f07e50b11cc}">
            <comment>
                <![CDATA[]]>
            </comment>
            <list name="MenuEntries" maxCount ="10" type ="FeatStd::String" windowSize="3"
uid="{0ccf837b-9769-41b0-b8f8-111a7b272233}"/>
        </bindingSource>

        <messagePool bindingSourceRef="Radio" messagePoolsize="3" />
        <messagePool bindingSourceRef="CarStatus" messagePoolsize="3" />
        <messagePool bindingSourceRef="SettingsMenu" messagePoolsize="8" />
        <messagePool bindingSourceRef="TestValues" messagePoolsize="3" />
    </bindingSources>
</definition>
```

List Handling

Description

Courier::SyncListInterface and Courier::AsyncListInterface are the central access interfaces for lists. Handling lists split up handling lists in the model component and in widgets. The model component supplies list data and emits list events while the bound widget properties consume this data and events and emit list requests towards the model component.

Widget List Handling

Widget properties must be defined as bindable property to enable them for data binding. List properties must be defined with Courier::ListPropertyType.

The following example defines among other properties a list property of type TString.

```
CdaWidgetDef(SampleWidget, Widget)
    CdaDescription("Sample widget")
    CdaReadableName("Sample Widget")

    CdaProperties()
        // bool property that controls render enable / disable of the the attached scene graph r
        CdaBindableProperty(Enabled, bool, GetEnabled, SetEnabled)
        CdaDescription("Status: enabled or not")
        CdaBindablePropertyEnd()

        // property excepting lists of type TString
        CdaBindableProperty
(MenuEntries, Courier::ListPropertyType<FeatStd::String>, GetMenuEntries, SetMenuEntries)
        CdaDescription("Test for properties bound to lists")
        CdaCategory("List Test Properties")
        CdaBindablePropertyEnd()

        // property with reference type FeatStd::String
        CdaBindableProperty(Label, FeatStd::String, GetLabel, SetLabel)
        CdaDescription("Test for properties bound to reference data types (FeatStd::String)")
        CdaBindablePropertyEnd()

        CdaBindableProperty
(ValueList, Courier::ListPropertyType<Courier::UInt32>, GetValueList, SetValueList)
        CdaCategory("List Test Properties")
        CdaDescription("Test for properties bound to lists")
        CdaBindablePropertyEnd()

        CdaBindableProperty
(GenericList, Courier::ListPropertyType<ListItemDataPresenter>, GetGenericList, SetGenericList)
        CdaCategory("List Test Properties")
        CdaDescription("generic list type")
        CdaBindablePropertyEnd()
```

```
CdaBindableProperty(BoolList, Courier::ListPropertyType<bool>, GetBoolList, SetBoolList)
        CdaCategory("List Test Properties")
        CdaDescription("boolean list")
        CdaBindablePropertyEnd()
    CdaPropertiesEnd()
CdaWidgetDefEnd()
```

The getter and setter function of a list property is very similar to any other property setter

```
const Courier::ListPropertyType<FeatStd::String>& GetMenuEntries() const {
    return mMenuEntriesProperty;
}

void SetMenuEntries(const Courier::ListPropertyType<FeatStd::String> &value) {
    mMenuEntriesProperty = value;
    mMenuEntriesProperty.SetEventCallback(this, &SampleWidget::OnMenuEntryPropertyListEvent);
}
```

The only difference to standard setter and getter is that the setter will not only set the property, but also a callback function to receive events that are triggered by the list.

The setter does not invoke FrameworkWidget::PropertyChanged as list objects are read only data items. Only items of the list can be changed by the widget.

List Events

Courier::ListPropertyType exposes a Courier::AsyncListInterface to the widget. The widget cannot synchronously read list elements but has to request list items from the list. The requested items then will be delivered with Courier::ListEvent. Courier::ListEventType::Enum defines other events a list might emit.

The following code shows a sample implementation for handling list events received by a list property.

```
void SampleWidget::OnMenuEntryPropertyListEvent(const Courier::ListEvent &listEvent)
{
    FeatStd::String string;
    const FeatStd::String *itemString = listEvent.GetItem<FeatStd::String>();
    if (itemString != 0) {    // if the list event carries a list item
        string = *itemString;
    }

    COURIER_LOG_INFO("received %s: newIndex=%u, oldIndex=%u, item=%s",
                     GetEventTypeName(listEvent.EventType()),
                     listEvent.NewIndex(),
                     listEvent.
```

```
OldIndex(),
                      string.GetCString());

    bool ok = true;

    switch (listEvent.EventType()) {
        // ignore new fragment events, fragment events come on top of RequestedItem events
        case Courier::ListEventType::NewFragment:
            break;

        // list item delivery bay - the order requested items come in is not specified
        case Courier::ListEventType::RequestedItem:
            SetMenuEntry(listEvent.NewIndex(), string, false);
            break;

        // change event handling
        case Courier::ListEventType::ModifiedItem:
            // change item in local list
            SetMenuEntry(listEvent.NewIndex(), string, false);
            if (!listEvent.HasDataItem()) {
                // if the event does not carry the modified data item, create a request for it
                ok = mMenuEntriesProperty.Request(listEvent.NewIndex(), 1);
            }
            break;

        // add event handling
        case Courier::ListEventType::AddedItem:
            // insert item to local list
            SetMenuEntry(listEvent.NewIndex(), string, true);
            if (!listEvent.HasDataItem()) {
                // if the event does not carry the modified data item, create a request for it
                ok = mMenuEntriesProperty.Request(listEvent.NewIndex(), 1);
            }
            break;

        case Courier::ListEventType::ListCleared:
            mMenuEntries.Clear();
            break;

        case Courier::ListEventType::MovedItem:
            ok = mMenuEntries.Move(listEvent.NewIndex(), listEvent.OldIndex());
            break;

        case Courier::ListEventType::RemovedItem:
            ok = mMenuEntries.RemoveAt(listEvent.NewIndex());
            break;
```

```
        case Courier::ListEventType::ItemCountChanged:
        case Courier::ListEventType::RefreshList:
            ResetMenuEntries();
            ok = mMenuEntriesProperty.Request(0, mMenuEntriesProperty.Count());
            break;

        case Courier::ListEventType::None:
        default:
            COURIER_DEBUG_FAIL();
            break;
    }

    if (!ok) {
        COURIER_LOG_WARN("error processing list event");
    }

    if (listEvent.EventType() != Courier::ListEventType::NewFragment) {
        // we mirrored the complete list if count matches and no empty string
        // is in menu entries. if we achieved this state, log all entries and
        // send notification message (will trigger next test)
        bool complete = mMenuEntries.Count() == mMenuEntriesProperty.Count();
        for (FeatStd::UInt32 i = 0; complete && i < mMenuEntries.Count(); ++i) {
            complete = *mMenuEntries[i].GetCString() != 0;
        }
        if (complete) {
            for (FeatStd::UInt32 i = 0; i < mMenuEntries.Count(); ++i) {
                COURIER_LOG_DEBUG("MenuItem %02u %s", i, mMenuEntries[i].GetCString());
            }

            // state has been updated -> tell the world about it
            // in this sample the message will trigger the execution of the next test step in Sa
            SampleWidgetMenuEntriesSetMsg *msg = COURIER_MESSAGE_NEW
(SampleWidgetMenuEntriesSetMsg)();
            if (msg != 0) {
                (void) msg->Post();
            }
        }
    }
}
```

Model Component List Handling

List Event

Lists can undergo multiple types of changes. List changes are therefore always propagated in two steps

- send the binding source data that contains the modified list

- send one or more notifications that tell about what changes have been applied on the lists

Enforce a refresh of list content

If the list content has changed on a large scale, issuing a refresh event triggers all data clients to reset their state and to reload list contents.

```
bool SampleAppModelComponent::RefreshListTest()
{
    bool ok;

    // initialize the list
    ok = SetupMenuEntries();

    // menu entries are sent in fragments -> copy the modified data to the fragment before sendi
    ok = ok && (*mSettingsMenuData).mMenuEntries.ReadFragment(mSettingsMenu, 0);

    // send data to the view
    ok = ok && mSettingsMenuData.SendUpdate();

    // use a list event notifier to send the list event
    ListEventNotifier settingsMenuNotifier(Generated::ItemKey::SettingsMenu::MenuEntriesItem);

    // trigger widget model refresh - refresh indicates that the widgets bound to the list shall
    // refresh all state related to this list
    ok = ok && settingsMenuNotifier.NotifyRefreshList();

    return ok;
}
```

Adding a list item

If a list item has been added, the change must be propagated to all clients of the list.

```
bool SampleAppModelComponent::AddItemTest()
{
    bool ok;

    ListEventNotifier settingsMenuNotifier(Generated::ItemKey::SettingsMenu::MenuEntriesItem);

    // add a menu item
    ok = mSettingsMenu.InsertAt(0, FeatStd::String("New menu entry"));

    // menu entries are sent in fragments -> copy the modified data to the fragment before sendi
    ok = ok && (*mSettingsMenuData).mMenuEntries.ReadFragment(mSettingsMenu, 0);

    // mark the item as modified - this will cause an automatic increment of the data revision r
    // of the binding source. Alternatively the data revision can also be manually incremented
    // with BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    ok = ok && mSettingsMenuData.MarkItemModified(ItemKey::SettingsMenu::MenuEntriesItem);
```

```
    // Send the updated list. note - we did not set the update bit of SettingsMenu::MenuEntriesI
    // as this would trigger a RefreshList (i.e. the list data item received a new value and thi
    // can only result in a refresh of the list in the widget
    ok = ok && mSettingsMenuData.SendUpdate();

    // *after* sending the data, we tell about the changes in the list. In this case we added ar
    // item at position 0
    ok = ok && settingsMenuNotifier.NotifyItemAdded(0);

    return ok;
}
```

Modifying a list item

If a list item has been modified, the change must be propagated to all clients of the list.

```
bool SampleAppModelComponent::ModifyItemTest()
{
    bool ok;
    ListEventNotifier settingsMenuNotifier(Generated::ItemKey::SettingsMenu::MenuEntriesItem);

    // modifiy the list item at index 0 and send the updated data & copy to fragment
    mSettingsMenu[0] = "modified list entry";

    // update the fragment
    ok = (*mSettingsMenuData).mMenuEntries.ReadFragment(mSettingsMenu, 0);

    // mark the item as modified - this will cause an automatic increment of the data revision r
    // of the binding source. Alternatively the data revision can also be manually incremented
    // with BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    ok = ok && mSettingsMenuData.MarkItemModified(ItemKey::SettingsMenu::MenuEntriesItem);

    ok = ok && mSettingsMenuData.SendUpdate();

    // tell which items have been changed (again - after the data has been updated)
    ok = ok && settingsMenuNotifier.NotifyItemModified(0);

    return ok;
}
```

List item count change

If a list item has been appended at the end of the list, the change must be propagated to all clients of the list.

```
bool SampleAppModelComponent::ChangeItemCountTest()
{
    bool ok;
    ListEventNotifier settingsMenuNotifier(Generated::ItemKey::SettingsMenu::MenuEntriesItem);

    // add items at the *end* of the list, send the modified list, and tell what changed in the
    ok = mSettingsMenu.PushBack(FeatStd::String("new entry"));
```

```
    // not to forget to update the list item count in the fragment too
    (*mSettingsMenuData).mMenuEntries.UpdateListItemCount(mSettingsMenu.Count());

    // mark the item as modified - this will cause an automatic increment of the data revision r
    // of the binding source. Alternatively the data revision can also be manually incremented
    // with BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    ok = ok && mSettingsMenuData.MarkItemModified(ItemKey::SettingsMenu::MenuEntriesItem);

    ok = ok && mSettingsMenuData.SendUpdate();
    ok = ok && settingsMenuNotifier.NotifyItemCountChanged();

    return ok;
}
```

List item has been moved

If a list item has been moved from one position to the other in the list, the change must be propagated to all clients of the list

```
bool SampleAppModelComponent::MoveItemTest()
{
    bool ok;
    ListEventNotifier settingsMenuNotifier(Generated::ItemKey::SettingsMenu::MenuEntriesItem);

    // move item at index 1 to index 0
    ok = mSettingsMenu.Move(0, 1);

    // update the fragment
    ok = ok && (*mSettingsMenuData).mMenuEntries.ReadFragment(mSettingsMenu, 0);


    // mark the item as modified - this will cause an automatic increment of the data revision r
    // of the binding source. Alternatively the data revision can also be manually incremented
    // with BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    ok = ok && mSettingsMenuData.MarkItemModified(ItemKey::SettingsMenu::MenuEntriesItem);

    // send update and notify about the list change
    ok = ok && mSettingsMenuData.SendUpdate();
    ok = ok && settingsMenuNotifier.NotifyItemMoved(0, 1);

    return ok;
}
```

List item has been removed

If a list item has been removed from the list, the change must be propagated to all clients of the list.

```
bool SampleAppModelComponent::RemoveItemTest()
{
    bool ok;
```

```
    ListEventNotifier settingsMenuNotifier(Generated::ItemKey::SettingsMenu::MenuEntriesItem);

    // remove item at index 0
    ok = mSettingsMenu.RemoveAt(1);

    // update the fragment
    ok = ok && (*mSettingsMenuData).mMenuEntries.ReadFragment(mSettingsMenu, 0);


    // mark the item as modified - this will cause an automatic increment of the data revision r
    // of the binding source. Alternatively the data revision can also be manually incremented
    // with BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    ok = ok && mSettingsMenuData.MarkItemModified(ItemKey::SettingsMenu::MenuEntriesItem);

    ok = ok && mSettingsMenuData.SendUpdate();
    ok = ok && settingsMenuNotifier.NotifyItemRemoved(0);

    return ok;
}
```

List has been cleared

If a list has been cleared, the change must be propagated to all clients of the list.

```
bool SampleAppModelComponent::ClearListTest()
{
    bool ok;
    ListEventNotifier settingsMenuNotifier(Generated::ItemKey::SettingsMenu::MenuEntriesItem);

    // clear the list
    mSettingsMenu.Clear();

    // update the fragment
    ok = (*mSettingsMenuData).mMenuEntries.ReadFragment(mSettingsMenu, 0);

    // mark the item as modified - this will cause an automatic increment of the data revision r
    // of the binding source. Alternatively the data revision can also be manually incremented
    // with BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    ok = ok && mSettingsMenuData.MarkItemModified(ItemKey::SettingsMenu::MenuEntriesItem);

    // update and notify the other components
    ok = ok && mSettingsMenuData.SendUpdate();
    ok = ok && settingsMenuNotifier.NotifyListCleared();
    return ok;
}
```

List Request Handling

```
bool SampleAppModelComponent::OnSettingsMenuListRequest(const Request &request)
{
    // handle requests towards SettingsMenuList
```

```
    COURIER_LOG_INFO("request %d, newIndex=%u, oldIndex=%u", request.RequestType(), request.NewI

    // determine if the request is based on an outdated data revision
    // this sample will gracefully ignore change requests on outdated data base. the only reques
    // data is the request for a new list fragment
    // request.DataRevision returns the revision number in which context the request has been dc
    // BindingSourceRevisionStore::GetRevision returns the current revision number of the bindir
    bool outDated = Courier::CompareRevisions
(request.DataRevision(), BindingSourceRevisionStore::GetRevision(request.ItemKey())) < 0;
    if (outDated && request.RequestType() != ListRequestType::NewFragment) {
        COURIER_LOG_DEBUG("received out dated request (%u vs. %u",
                          request.DataRevision(),
                          BindingSourceRevisionStore::GetRevision(request.ItemKey()));
        return false;
    }

    // will be set if the request requires an update to be sent
    bool requiresUpdate = false;
    // will be set if the request caused a change in the data model
    bool listChanged = false;

    // modify list data according to the request type
    switch(request.RequestType()) {
        case ListRequestType::AddItem: {
            // add a new list item at NewIndex
            const FeatStd::String *newItem = request.GetItem<FeatStd::String>();
            requiresUpdate = newItem != 0 && mSettingsMenu.InsertAt(request.NewIndex(), *newItem
            break;
        }

        case ListRequestType::ModifyItem: {
            // existing list item gets modified at index NewIndex
            const FeatStd::String *newItem = request.GetItem<FeatStd::String>();
            if (newItem != 0 && request.NewIndex() < mSettingsMenu.Count()) {
                mSettingsMenu[request.NewIndex()] = *newItem;
                listChanged = true;
            }
            break;
        }

        case ListRequestType::ClearList:
            // clear the list
            mSettingsMenu.Clear();
            listChanged = true;
            break;

        case ListRequestType::MoveItem:
            // move a list item to a new index
            listChanged =  mSettingsMenu.Move(request.NewIndex(), request.OldIndex());
            break;

        case ListRequestType::RemoveItem:
```

```
            // remove a list item
            listChanged = mSettingsMenu.RemoveAt(request.NewIndex());
            break;

        case ListRequestType::NewFragment:
            // explicit request for a new list fragment
            requiresUpdate = true;
            break;

        case ListRequestType::PrefetchItems:
            // nothing to do here - the list is not prefetching
            // this request causes normally the model to read data from external
            // data sources. it prepares near future NewFragment requests. for example
            // a list widget might send prefetch request for next list pages when the user
            // scrolls down.
            // No events are sent back to the view component in response to the prefetch
            // request.
            break;

        case ListRequestType::None:
        default:
            COURIER_DEBUG_FAIL();
            break;
    }

    if (listChanged) {
        // if the list got modified, increase data revision number
        (void) BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    }

    // return true if the request requires updates to be sent
    return requiresUpdate || listChanged;
}
```

Type Converter

To establish a binding between a data item and a widget property, the data item type must match exactly the property type.

To overcome that limitation, data binding defines the concept of type converters (see Courier::TypeConverter). A type converter is nothing more than two functions that convert from type A to B and vice versa. Thus the overhead of a type converter is quite low (approximately 8 bytes RAM per type converter and the code size of the two conversion functions).

```
struct DateTime {
    Courier::UInt mHour;
    Courier::UInt mMinute;
```

```
    Courier::UInt mSecond;

    Courier::UInt mDay;
    Courier::UInt mMonth;
    Courier::UInt mYear;
};

// DateTime --> std::string
bool TypeConvert(std::string &str, const DateTime &dateTime)
{
    Courier::Char buf[50];
    sprintf(buf, "%02u:%02u:%02u %u.%u.%u", dateTime.mHour, dateTime.mMinute,
            dateTime.mSecond, dateTime.mDay, dateTime.mMonth, dateTime.mYear);
    str = buf;
    return true;
}

// std::string --> DateTime
bool TypeConvert(DateTime &dateTime, const std::string &str)
{
    Courier::Int n = sscanf(str.c_str(), "%02u:%02u:%02u %u.%u.%u", &dateTime.mHour,
&dateTime.mMinute,
                            &dateTime.mSecond, &dateTime.mDay, &dateTime.mMonth,
&dateTime.mYear);
    return n == 6;
}

bool RegisterMyTypeConverters()
{
    using Courier::TypeConverter;
    // create a TypeConverter instance to make it known to Courier DataBinding
    static TypeConverter<std::string, DateTime> converter;

    return true;
}
```

Courier ships already with a set of standard type converters. To activate the converters the application must invoke Courier::RegisterStdTypeConverters.

Type converters should be created at system startup time only. The list of type converters is not synchronized.

Reference Types

To avoid multiple instances and copying of large buffers like images, messages can also reference types. A reference type does not store data intrinsically, but keeps only a reference to the data buffer. Needless to say that in a multi threaded configuration (e.g. model component and view component are executed in different threads) the access to the reference data must be synchronized. The Courier SampleApplication implements a reference type for UTF8 encoded strings which might act as a starting point for application specific reference types.

Model Component Sample Implementation

Model Component Implementation

The data model defined in Courier Data Definition Language XCDDL is hosted by CourierSampleApplication in the following class

```
class SampleAppModelComponent : public Courier::ModelComponent {
    public:
        SampleAppModelComponent();
        virtual ~SampleAppModelComponent();

        bool Init();

    private:
        Courier::Int32 mSpeedDelta;
        Courier::UInt32 mDataBindingTestCount;
        Courier::DataItemContainer<Generated::RadioDataBindingSource> mRadioData;
        Courier::DataItemContainer<Generated::CarStatusDataBindingSource> mCarStatus;
        Courier::DataItemContainer<Generated::SettingsMenuDataBindingSource>
mSettingsMenuData;
        Courier::DataItemContainer<Generated::TestValuesDataBindingSource> mTestValuesData;
        // store for complete menu items list
        Courier::FixedSizeList<FeatStd::String, Generated::SettingsMenuData::cMenuEntriesMax>
mSettingsMenu;

        void OnSpeedChanged(Courier::UInt32 newSpeed);
```

```
        void OnAutoPilot();

        void OnStartupMsg();

        bool OnSettingsMenuListRequest(const Courier::Request &request);
        bool OnTestValuesRequest(const Courier::Request &request);

        bool UpdateListFragment(Courier::DataItemKey listItemKey, Courier::UInt32
fragmentStart);
        bool SendBindingSourceUpdate(Courier::DataItemKey bindingSourceKey);

        virtual bool OnMessage(const Courier::Message & msg);
        void OnUpdateModelMsg(const Courier::UpdateModelMsg *msg);
};
```

The class defines data members for the binding sources defined in XCDDL. The data types generated from XCDDL can, but don't have to be used in the model component.

```
        Courier::DataItemContainer<Generated::RadioDataBindingSource> mRadioData;
        Courier::DataItemContainer<Generated::CarStatusDataBindingSource> mCarStatus;
        Courier::DataItemContainer<Generated::SettingsMenuDataBindingSource>
mSettingsMenuData;
        Courier::DataItemContainer<Generated::TestValuesDataBindingSource> mTestValuesData;
        // store for complete menu items list
        Courier::FixedSizeList<FeatStd::String, Generated::SettingsMenuData::cMenuEntriesMax>
mSettingsMenu;
```

The SettingsMenuDataBindingSource contains a fragment list. Therefore the binding source data types can only store a fragment of the menu item list. The complete list must be stored somewhere else (in the sample in a FixedSizeList container).

Courier::DataItemContainer is a convenience class to handle binding source data.

As in any other component messages are received with Courier::Component::OnMessage method.

```
// ---------------------------------------------------------------------
bool SampleAppModelComponent::OnMessage(const Courier::Message & msg)
{
    // central message bay for the SampleAppModelComponent

    switch(msg.GetId()) {
```

```
        case StartupMsg::ID:
            OnStartupMsg();
            break;

        case UpdateModelMsg::ID:
            // request from controller / view to change model data
            OnUpdateModelMsg(message_cast<const UpdateModelMsg*>(&msg));
            break;

        case ToggleAutoPilotMsg::ID:
            // external request (normally triggered by hitting the 'a' key)
            OnAutoPilot();
            break;

        case SpeedMsg::ID: {
            // external speed value message (triggered normally by pressing keys 0-9
            const SpeedMsg * speedMsg = message_cast<const SpeedMsg *>(&msg);
            if(speedMsg != 0) {
                OnSpeedChanged(speedMsg->GetSpeed());
            }
            break;
        }

        // corresponding widget code was removed
        //case NextDataBindingTestMsg::ID: {
        //    // test execution control message. sent from the SampleViewController
        //    // run the next test step
        //    bool result = RunDataBindingTests();
        //    // respond with DataBindingTestExecutedMsg
        //    DataBindingTestExecutedMsg *theMsg =
COURIER_MESSAGE_NEW(DataBindingTestExecutedMsg)(result);
        //    if (theMsg != 0) {
        //        (void) theMsg->Post();
        //    }
        //    break;
        //}

        default:
            break;
    }
```

```
    return false;
}
```

The OnMessage method both receives external data input (in sample application SpeedMsg or ToggleAutoPilotMsg) and messages from other components (controller and view). The most important message from other components is Courier::UpdateModelMsg. This message carries requests from the components that the model shall process.

Of course it is up to the application developer to define any number of additional messages that the model can receive from other components (like the NextDataBindingTestMsg that triggers test execution).

Processing of Courier::UpdateModelMsg takes place in OnUpdateModelMsg.

```
// ----------------------------------------------------------------------
void SampleAppModelComponent::OnUpdateModelMsg(const UpdateModelMsg *msg)
{
    using namespace Generated;

    // controller / view request changes in the model data. An UpdateModelMsg
    // may contain multiple change requests for multiple binding sources.
    // the ChangeTracker helps to keep track of the changes and takes over the
    // house keeping tasks like list fragment update, sending of DataItemMsg
    // with modified data, and list event notifications.
    ChangeTracker changeTracker(this,
                                &SampleAppModelComponent::UpdateListFragment,
                                &SampleAppModelComponent::SendBindingSourceUpdate);

    bool ok = true;
    // for every request in the UpdateModelMsg
    for (UInt32 i = 0; ok && i < msg->RequestCount(); ++i) {
        // get request from UpdateModelMsg
        Request request(msg->GetRequest(i));
        switch (request.BindingSourceKey()) {
            case ItemKey::SettingsMenuItem:
                if (OnSettingsMenuListRequest(request)) {
                    ok = changeTracker.RegisterChange(i, request);
                }
                break;

            case ItemKey::TestValuesItem:
```

```
                if (OnTestValuesRequest(request)) {
                    ok = changeTracker.RegisterChange(i, request);
                }
                break;

            case ItemKey::RadioItem:
                break;

            default:
                // unknown binding source
                COURIER_DEBUG_FAIL();
                ok = false;
                break;
        }
        if (!ok) {
            COURIER_LOG_ERROR("update request failed");
        }
    }

    // trigger send of DataItemMsg and ListEvent notifications for the requested
    // changes. changeTracker keeps track which binding source data to update and
    // which list events to send.
    if (!changeTracker.SendChanges(msg)) {
        COURIER_LOG_ERROR("failed to update data");
    }
}
```

The processing of requests for the TestValues binding source is done in SampleAppModelComponent::OnTestValuesRequest.

```
// ----------------------------------------------------------------------
bool SampleAppModelComponent::OnTestValuesRequest(const Courier::Request &request)
{
    // handle request towards TestValues binding source data

    bool appliedChange;
    switch(request.ItemKey()) {
        case ItemKey::TestValues::TestValueItem:
            appliedChange = mTestValuesData.SetValue(request.ItemKey(),
request.GetItemValue());
```

```
            COURIER_LOG_INFO("TestValue->%u", (*mTestValuesData).mTestValue);
            break;

        case ItemKey::TestValues::TestValueListItem:
            // as the list is read only (allowAdd=false etc.) only NewFragment and Prefetch
            // request may come in.
            appliedChange = true;
            break;

        default:
            appliedChange = false;
            break;
    }

    return appliedChange;
}
```

Full handling of a modifiable list is implemented in SampleAppModelComponent::OnSettingsMenuListRequest.

```
// ------------------------------------------------------------------------
bool SampleAppModelComponent::OnSettingsMenuListRequest(const Request &request)
{
    // handle requests towards SettingsMenuList

    COURIER_LOG_INFO("request %d, newIndex=%u, oldIndex=%u", request.RequestType(),
request.NewIndex(), request.OldIndex());

    // determine if the request is based on an outdated data revision
    // this sample will gracefully ignore change requests on outdated data base. the only
request allowed on outdated
    // data is the request for a new list fragment
    // request.DataRevision returns the revision number in which context the request has been
done by the view component
    // BindingSourceRevisionStore::GetRevision returns the current revision number of the
binding source in the model
    bool outDated = Courier::CompareRevisions(request.DataRevision(),
BindingSourceRevisionStore::GetRevision(request.ItemKey())) < 0;
    if (outDated && request.RequestType() != ListRequestType::NewFragment) {
        COURIER_LOG_DEBUG("received out dated request (%u vs. %u",
```

```
                        request.DataRevision(),
                        BindingSourceRevisionStore::GetRevision(request.ItemKey()));
        return false;
    }

    // will be set if the request requires an update to be sent
    bool requiresUpdate = false;
    // will be set if the request caused a change in the data model
    bool listChanged = false;

    // modify list data according to the request type
    switch(request.RequestType()) {
        case ListRequestType::AddItem: {
            // add a new list item at NewIndex
            const FeatStd::String *newItem = request.GetItem<FeatStd::String>();
            requiresUpdate = newItem != 0 && mSettingsMenu.InsertAt(request.NewIndex(),
*newItem);
            break;
        }

        case ListRequestType::ModifyItem: {
            // existing list item gets modified at index NewIndex
            const FeatStd::String *newItem = request.GetItem<FeatStd::String>();
            if (newItem != 0 && request.NewIndex() < mSettingsMenu.Count()) {
                mSettingsMenu[request.NewIndex()] = *newItem;
                listChanged = true;
            }
            break;
        }

        case ListRequestType::ClearList:
            // clear the list
            mSettingsMenu.Clear();
            listChanged = true;
            break;

        case ListRequestType::MoveItem:
            // move a list item to a new index
            listChanged =  mSettingsMenu.Move(request.NewIndex(), request.OldIndex());
            break;
```

```
        case ListRequestType::RemoveItem:
            // remove a list item
            listChanged = mSettingsMenu.RemoveAt(request.NewIndex());
            break;

        case ListRequestType::NewFragment:
            // explicit request for a new list fragment
            requiresUpdate = true;
            break;

        case ListRequestType::PrefetchItems:
            // nothing to do here - the list is not prefetching
            // this request causes normally the model to read data from external
            // data sources. it prepares near future NewFragment requests. for example
            // a list widget might send prefetch request for next list pages when the user
            // scrolls down.
            // No events are sent back to the view component in response to the prefetch
            // request.
            break;

        case ListRequestType::None:
        default:
            COURIER_DEBUG_FAIL();
            break;
    }

    if (listChanged) {
        // if the list got modified, increase data revision number
        (void) BindingSourceRevisionStore::IncRevision(ItemKey::SettingsMenuItem);
    }

    // return true if the request requires updates to be sent
    return requiresUpdate || listChanged;
}
```

> The helper class ChangeTracker is implemented in SampleApplication and is not part of Courier framework.

Frequently Asked Questions

I get unresolved externals for symbols Courier::DataBindingPrivate::gInfrastructure

DataBinding requires certain application specific infrastructure for operation (e.g. description of your data model). Normally this infrastructure will be generated by CourierGenerator during XCDDL generation. If the application does not use a generated data model, the infrastructure is missing and will cause the listed unresolved external.

To resolve the link error add the macro COURIER_DATABINDING_DEFAULT_INFRASTRUCTURE to the top level scope of one of the applications source (not header!) file.

```
// no generated data items in this build -> define default infrastructure
#include <Courier/Courier.h>
COURIER_DATABINDING_DEFAULT_INFRASTRUCTURE();
```

My list widget invokes RequestPrefetch on a bindable list property but the request never gets to the model component

---

If your list is not updated in fragments (windowSize is defined in XCDDL), requesting to prefetch the list makes no sense (you already receive the complete list in a Courier::DataItemMsg) and therefore the request is ignored.

My widget changes a bound property value but the change is not propageted back to the model

Most likely you forgot to inform data binding about the change with FrameworkWidget::PropertyModified.

```
void SampleWidget::SetEnabled(bool val)
{
    if ((mEnabled && !val) || (!mEnabled && val)) {
        mEnabled = val;
        if (GetNode() != 0) {
            GetNode()->SetRenderingEnabled(mEnabled);
        }
        PropertyModified("Enabled");
```

```
    }
}
```

I get a compile error in BindableProperty.h (use of undefined type Courier::Diagnostics::Private::CtaOnlyTrue)

Most likely you defined a bindable property with a pointer or reference. As the ownership to referred data is not correctly managed, data binding does not support pointer, C/C++ arrays, and references as property type. Check Reference Types how to handle buffer data.

```
    typedef const Char* ConstCharPtr;
    const ConstCharPtr GetLabelX() const {
        return 0;
    }

    void SetLabelX(ConstCharPtr) {
    };
    ...

    CdaWidgetDef(SampleWidget, Widget)
        CdaBindableProperty
(LabelX, ConstCharPtr, GetLabelX, SetLabelX)     // njet - will cause compile time assertion
        CdaBindablePropertyEnd()
    CdaWidgetDefEnd()
```

My widget properties gets updates disregarding the data did change or not

Data binding does not check if the widget property data changed prior to updating the property. This is in control of the model component which can set the modified flags in the Courier::DataItemMsg accordingly. If the model fails to do this correctly, it is a good practice to check in the setter of the property if the new property value differs from the existing. If the values did not change, the widget should ignore the setter invocation.

```
void SampleWidget::SetEnabled(bool val)
{
    if ((mEnabled && !val) || (!mEnabled && val)) {
        mEnabled = val;
        if (GetNode() != 0) {
            GetNode()->SetRenderingEnabled(mEnabled);
        }
        PropertyModified("Enabled");
    }
}
```

My attempts to establish a binding between a data item and widget property fails

Reasons of failure

- The data types of widget property and data item don't match and type conversion is disabled or according Courier::TypeConverter objects could not be found.
- The binding requires type conversion and the type exceeds COURIER_TYPE_CONVERSION_BUFFER_SIZE.
- The data item is not an elementary item
- Typo in widget type or property type name

If data types don't match, enabling standard type converts may help (Courier::RegisterStdTypeConverters). They supply conversion functions between built-in numerical types.

Revision #10
Created 2023-02-27 05:19:52 UTC by Tsuyoshi.Kato
Updated 2025-06-01 22:19:56 UTC by Elisabeth Zauner