AltioLive Best Practices (client-side)

A description of some best practice approaches to developing applications with AltioLive.

Document Version 1.0

Author : Jim Crossley & Graham Howarth

Contents

1 Introduction 3
2 Best Practices 3
2.1 General 3
2.2 Service Function Invocation 4
2.3 Hidden Controls 7
2.4 Re-usable Code 7
2.4.1 Application architecture 7
2.4.2 Using Plugin Modules 7
3 Data updates and local data 7
3.1 How Altio processes data updates 7

Copyright © 2007 IntegraSP Ltd.

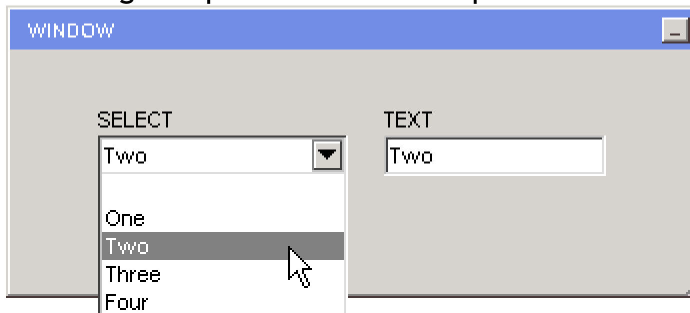
1 Introduction

This document is targeted at advanced AltioLive application developers. It describes recommended coding practices for application developers to adhere to. It also clarifies how best to implement parts of an application, especially when there appears to be more than one apparent way of approaching a task.

2 Best Practices

2.1 General

Where possible, rather than using an action to set the value of a control, assign the target control a dynamic data source. So for the following example where selecting an option from the dropdown sets the value of the text box:



Whilst both of the following two implementations work fine, the first one is preferred. Firstly because it is tidier, and secondly because since it is a data driven action, it is more in keeping with the logic for the data source of the text box to reference that of the select box.

Method 1:

```
<WINDOW NM="WINDOW" CAPTION="WINDOW" DATA="${windowcontext}" STARTUP="Y" X="400" Y="120" H="119" W="339" DEFERLOAD="Y">
  <PANEL>
    <SELECT NM="SELECT" CAPTION="SELECT" X="42" Y="43" H="20" W="125" DATA="${windowelement}">
      <OPTION VALUE=""/>
      <OPTION>One</OPTION>
      <OPTION>Two</OPTION>
      <OPTION>Three</OPTION>
      <OPTION>Four</OPTION>
    </SELECT>
    <TEXT NM="TEXT" CAPTION="TEXT" X="185" Y="43" H="20" W="111" DATAFLD="${SELECT.value}"/>
  </PANEL>
</WINDOW>
```

Copyright © 2007 IntegraSP Ltd.

Method 2:

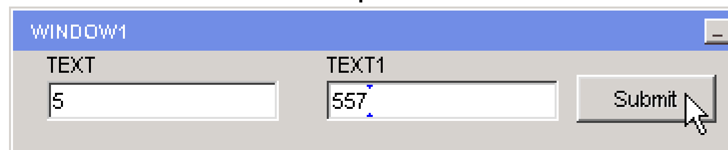
```
<WINDOW NM="WINDOW" CAPTION="WINDOW" DATA="${windowcontext}" STARTUP="Y" X="400" Y="120" H="119" W="339" DEFERLOAD="Y">
  <PANEL>
    <SELECT NM="SELECT" CAPTION="SELECT" X="42" Y="43" H="20" W="125" DATA="${windowelement}">
      <OPTION VALUE=""/>
      <OPTION>One</OPTION>
      <OPTION>Two</OPTION>
      <OPTION>Three</OPTION>
      <OPTION>Four</OPTION>
      <ACTIONRULES TRIGGER="ONCHANGE" DESC="Change">
        <ACTIONRULE>
          <ACTSETPROPERTY CONTROL="TEXT" PROPERTY="VALUE" VALUE="${SELECT.value}"/>
        </ACTIONRULE>
      </ACTIONRULES>
    </SELECT>
    <TEXT NM="TEXT" CAPTION="TEXT" X="185" Y="43" H="20" W="111"/>
  </PANEL>
</WINDOW>
```

- Keep the XML data structure as simple and flat as possible. This improves performance, readability and maintainability.
- Keep Datakeys as short in length as possible. This improves performance, since Altio uses Datakeys to index the data.
- Limit the manipulation of local data xml as much as possible. This is good coding practice, and avoids 'spaghetti' style code that is difficult to maintain.

2.2 Service Function Invocation

In general, use an argument type of STRING rather than WINDOWDATA when making service calls. This is easier to maintain, and also reduces the length of the argument string sent to the server.

In this sample, the values of the two text fields will be sent to a servlet, invoked when the “Submit” button is pressed.



Here are four different implementations of the same piece of functionality (note XML has been tidied slightly for readability).

Method 1: Window Data.
For simple, small windows such as this one, WINDOWDATA can be used. Note though, that if there were ten other controls on this window that we didn't care about, all of their values would be passed back to the server as well, using up unnecessary bandwidth.

```
<WINDOW NM="WINDOW1" CAPTION="WINDOW1" DATA="${windowcontext}" STARTUP="Y" X="400" Y="120" H="48" W="359" DEFERLOAD="Y">
  <PANEL>
    <TEXT NM="TEXT" CAPTION="TEXT" X="15" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <TEXT NM="TEXT1" CAPTION="TEXT1" X="155" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <BUTTON NM="SUBMIT" CAPTION="Submit" X="280" Y="12" W="70" H="24">
      <ACTIONRULES TRIGGER="CLICK" DESC="Click">
        <ACTIONRULE>
          <ACTSERVER CMD="Service" DATATYPE="WINDOWDATA"/>
        </ACTIONRULE>
      </ACTIONRULES>
    </BUTTON>
  </PANEL>
</WINDOW>
```

Method 2: String data.
This is a good method, since only the necessary values are sent back to the server. A bit of extra coding work is required to put in the name-value pairs, but this in fact does give the advantage of improved code readability.

```
<WINDOW NM="WINDOW1" CAPTION="WINDOW1" DATA="${windowcontext}" STARTUP="Y" X="400" Y="120" H="48" W="359" DEFERLOAD="Y">
  <PANEL>
    <TEXT NM="TEXT" CAPTION="TEXT" X="15" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <TEXT NM="TEXT1" CAPTION="TEXT1" X="155" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <BUTTON NM="SUBMIT" CAPTION="Submit" X="280" Y="12" W="70" H="24">
      <ACTIONRULES TRIGGER="CLICK" DESC="Click">
        <ACTIONRULE>
          <ACTSERVER CMD="Service" DATATYPE="STRING" DATA="TEXT='eval(${TEXT.value})' TEXT1='eval(${TEXT1.value})'"/>
        </ACTIONRULE>
      </ACTIONRULES>
    </BUTTON>
  </PANEL>
  <FUNCTIONS/>
</WINDOW>
```

Method 3:
Function. This removes the bulk and processing syntax from the button into a function. This is useful when lots of actions need to be called, and the developer wants to tidy the appearance of the code by grouping the actions together in a function. Note that parameters are not passed to the function in this example, and the text values are obtained from within the function.

```
<WINDOW NM="WINDOW1" CAPTION="WINDOW1" DATA="${windowcontext}" STARTUP="Y" X="400" Y="120" H="48" W="359" DEFERLOAD="Y">
  <PANEL>
    <TEXT NM="TEXT" CAPTION="TEXT" X="15" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <TEXT NM="TEXT1" CAPTION="TEXT1" X="155" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <BUTTON NM="SUBMIT" CAPTION="Submit" X="280" Y="12" W="70" H="24">
      <ACTIONRULES TRIGGER="CLICK" DESC="Click">
        <ACTIONRULE>
          <ACTFUNCTION NM="doService"/>
        </ACTIONRULE>
      </ACTIONRULES>
    </BUTTON>
  </PANEL>
  <FUNCTIONS>
    <FUNCTION NM="doService">
      <ACTIONRULE>
        <ACTSERVER CMD="Service" DATATYPE="STRING" DATA="TEXT='eval(${TEXT.value})' TEXT1='eval(${TEXT1.value})'"/>
      </ACTIONRULE>
    </FUNCTION>
  </FUNCTIONS>
</WINDOW>
```

Method 4:
Function with parameters. While perhaps overkill for this example, this presents the most re-usable code, allowing the function to be called with parameters even from different windows.

```
<WINDOW NM="WINDOW1" CAPTION="WINDOW1" DATA="${windowcontext}" STARTUP="Y" X="400" Y="120" H="48" W="359" DEFERLOAD="Y">
  <PANEL>
    <TEXT NM="TEXT" CAPTION="TEXT" X="15" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <TEXT NM="TEXT1" CAPTION="TEXT1" X="155" Y="15" H="20" W="116" DATA="${windowelement}"/>
    <BUTTON NM="SUBMIT" CAPTION="Submit" X="280" Y="12" W="70" H="24">
      <ACTIONRULES TRIGGER="CLICK" DESC="Click">
        <ACTIONRULE>
          <ACTFUNCTION NM="doService" CONTEXT="TEXT='eval(${TEXT.value})' TEXT1='eval(${TEXT1.value})'"/>
        </ACTIONRULE>
      </ACTIONRULES>
    </BUTTON>
  </PANEL>
  <FUNCTIONS>
    <FUNCTION NM="doService">
      <ACTIONRULE>
        <ACTSERVER CMD="Service" DATATYPE="STRING" DATA="TEXT='eval(@TEXT)' TEXT1='eval(@TEXT1)'"/>
      </ACTIONRULE>
    </FUNCTION>
  </FUNCTIONS>
</WINDOW>
```

Copyright © 2007 IntegraSP Ltd.

2.3 Hidden Controls

Hidden controls can be used to force actions to take place sequentially in an application. This is usually done by adding an action under the Data Changed trigger of the hidden control. Thus, when the underlying data is updated, the action is executed.

Another example of when the use hidden controls is needed is when data needs to be presented in a certain format for the next step, e.g. submission to a service function, but such formatting cannot be done using the functions available in the client, or XPath.

2.4 Re-usable Code

2.4.1 Application architecture

For a larger sized application, try to make windows modular and self-contained. Service calls for data appearing on a window should be located on this window, in a function if applicable.

Use functions and parameterised functions, especially when code has the potential to be re-used. Place the function on a relevant window where data updates will be received; otherwise make it a global function at the view level.

2.4.2 Using Plugin Modules

AltioLive applications can be extended on the server side using Plugin modules, but that is beyond the scope of this document and will be discussed in a separate document.

3 Data updates and local data

3.1 How Altio processes data updates

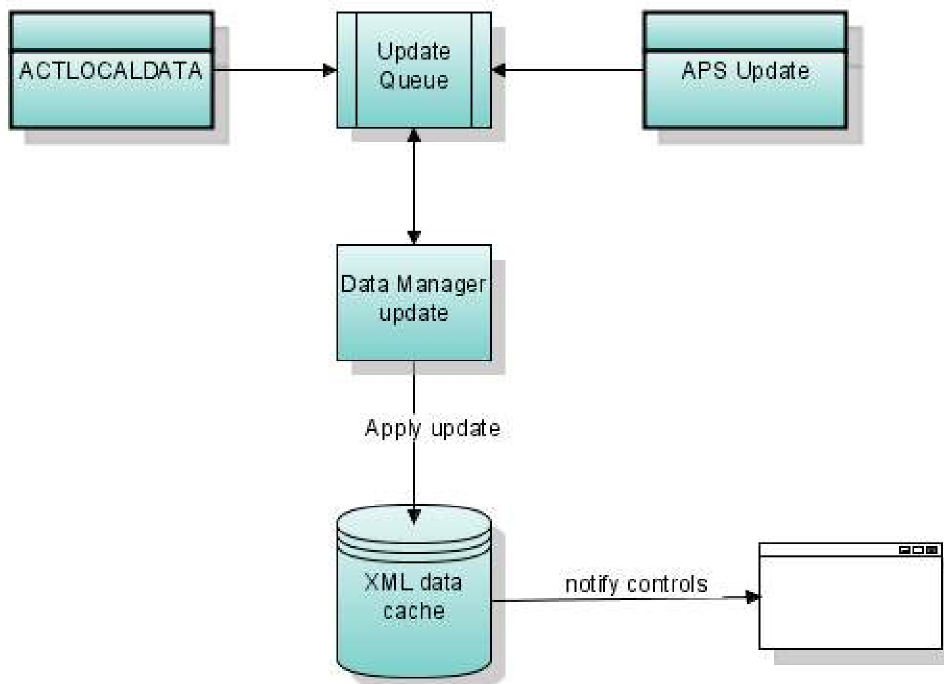
There has recently been some discussion with customers who wanted clarification on how the AltioLive SmartClient's data engine processes data updates, particularly with regard to ACTLOCALDATA actions that operate directly on the local data cache, rather than data updates sent from the AltioLive Presentation Server.

AltioLive's data layer is designed around an event-driven model similar to the Observer design pattern (http://en.wikipedia.org/wiki/Observer_pattern), in which controls register with the Data Manager which XML elements they subscribe to, and they are then notified when that data is changed, such as when an update is received from the APS.

Copyright © 2007 IntegraSP Ltd.

When the data manager receives a data update each XML element in the update block is applied sequentially to the data cache, and then the listener controls are notified of the update. In the situation where an update is received while the data manager is already processing an update, the updates are queued up, to be processed in the order in which they are received.

The diagram below illustrates the basic architecture:



This is has several implications, the most important of which is that when calling an Altio ACTION that can modify data (e.g. ACTSERVER, ACTLOCALDATA) the data update is **not** guaranteed to be applied when the next ACTION in the sequence is processed. ACTLOCALDATA was originally implemented as a useful action for small manipulations of the local data cache - the reliable and preferred method of reacting to data changes is to use event-driven code, such as the Data added/updated/deleted/changed events available on data bound controls.

Copyright © 2007 IntegraSP Ltd.

In the following example:

```
1 <ACTIONRULES>
2     <ACTIONRULE>
3         <ACTLOCALDATA XML="<new><sampledata value='1' 
anothervalue='2'/></new>"/>
4         <ACTSETPROPERTY CONTROL="TEXT" PROPERTY="VALUE" 
VALUE="/new/sampledata/@value" />
5     </ACTION>
6 </ACTIONRULES>
```

In most cases, the data specified in line 3 will be applied to the local data cache by the time line 4 is executed. However, under certain circumstances this may not be true - for example, if the data manager was processing an update from the server when line 3 was executed then the ACTLOCALDATA update would be queued for processing and line 4 would be executed without that data being applied. If your application makes use of ACTLOCALDATA calls with subsequent actions that rely on the data set by those calls, then you should bear in mind that the data is not guaranteed to have been applied at that point.

The availability of Action rules within Altio has led customers to implement complicated application and business logic in Action rules and functions, which is not an issue except when presuming that ACTLOCALDATA calls always return when their data update is complete. An alternative way to structure your application logic is to use data update events on controls to respond to data changes. Action rules in data change event blocks are obviously not going to be triggered until the data has been applied by the Altio data manager - regardless of whether updates have been queued or not. The easiest way to implement this is to use hidden controls, such as Text boxes or Labels, bound to data elements that your application updates elsewhere.

Copyright © 2007 IntegraSP Ltd.