ASP.NET Programming Techniques Specific to QP7.Framework

Using IntelliSense – Making API Calls

When working with inside Visual Studio with install QP7 Add-On, IntelliSense features are readily available like for any Visual Studio Project. Also, the Add-On tree elements can be dragged-and-dropped onto the code windows, Presentation or Code Behind.

When programming in the backend directly a pseudo-IntelliSense is available from the floating information window (invoked by clicking on small round icon at the top right corner). Once opened, double clicking on various lines will place a method call inside Presentation or Code Behind tab of the coding window. Making Object Calls – Presentation

Presentation section of object format is basically the same as the one used by .NET Framework. To call another QP7 object the following syntax is used (used drag-n-drop in Visual Studio Add-on or info box if programming in the backend) :

<qp:placeholder calls=”[TemplateName.]ObjectName[.FormatName]” runat=”server” />

To use your own controls follow regular .NET syntax to register tag prefix and then create an instance of your control:

<%@ Register TagPrefix=“myPref” TagName=“myName” Src=“myControl.ascx”%> …
<myPref:myName runat=“server” />

It’s not recommended to use Response.Write since the location of its output does not correlate to its placement in the code (Response.Write statements will usually be outputted in the beginning before anything else is sent to browser). Instead, use data-bound variables using this syntax: <%# %> or use Literal or Label or any other .NET control (text of such controls can be updated dynamically via Code Behind).

It’s also not recommended to use server scripts in the Presentation section:

<script language=”c#” runat=”server” > …Code… </script>

Instead, use Code Behind to place all of your page/object processing code.

Making Object Calls – Code Behind

Code Behind section of object format is basically the same as Code Behind section of .NET classes, meant to store all page and object event processing code. Code Behind Structure and Sequence of QP7 Events By default when a new format is created, Code Behind will contain the following code:

protected void LoadContainer(Object sender, EventArgs e) {

```
container.DataSource = Data;
```

}

override public void InitUserHandlers(EventArgs e) {

```
LoadContainer(this,e);
```

}

InitUserHandlers() is the first method that will be called once the object/control is loaded by QP7 via LoadControl(). If the object call was made by using <qp:placeholder /> in the Presentation then actual object will be loaded during OnInit of qp:placeholder control. During the invocation an instance of the object class is created, loaded and added to current control's control hierarchy (it can also be loaded into another control's control hierarchy, see further down the document) immediately followed by invocation of the loaded control's OnInit() and DataBind().

InitUserHandlers() is invoked from OnInit() of the loaded control. Then, after OnInit() and InitUserHandlers() complete their processing, the loaded control is added to the current (or specified) control's collection.

In cases where an object is invoked by calling ShowObject() method, the loading of the control is done from the event or method containing this ShowObject() call.

Important point to keep in mind that .NET Framework will load all of the controls in Presentation before the ones in Code Behind. In fact it will load all of child controls in Presentation before QP7 has a chance to call OnInit() for the parent control. This is usually never an issue since the sequence of calling controls remains the same (OnInit() get executed in the calling sequence) and the fact that there several opportunities with page events to modify control collection and/or modify control data.

In addition, there is no need to call DataBind() on control of child controls as it's handled automatically by QP7.

InitUserHandlers() may call other methods or event handlers. By default: LoadContainer (for Publishing Container) and LoadGeneric (for other object types) are defined in Code Behind and called by it.

When an object of type Publishing Container is assembled another method, LoadControlData, is added to Code Behind. This method is tasked with making a database call to receive articles from a Content Area and setting Data property to contain the resulting DataTable:

override public void LoadControlData(System.Object sender, System.EventArgs e) {

}

This method does not need to be overridden and is invoked automatically by OnInit(). Class and Namespace Declaration As you programming in the QP7 environment, Visual Studio or backend, you will notice that Code Behind does not have class declaration. This is because it's added automatically during assembly with Code Behind code inside it, thus, class declaration should not be present in Code Behind. Same goes for the NameSpace declaration. The class declaration looks similar to the syntax below:

namespace Quantumart.QPublishing.Site34 {

public class objectNetName: Quantumart.QPublishing.QUserControl { … code … } }

, where objectNetName corresponds to .NET Object Name automatically or explicitly defined during

object creation. QP7 Class Inheritance Depending on the object type, the resulting assembled class is inherited from different QP7 classes. For example, for Generic object type, the resulting class is inherited from Quantumart.QPublishing.QUserControl class. For Publishing Container type, the class would be inherited from Quantumart.QPublishing.QPublishControl class. Import or Using Directives

To simplify writing code, just like in .NET, in Code Behind import/using directives may be used (you can add your own ones) For example:

using System; using System.Collections; using System.Web.UI; using System.Web.UI.WebControls;

At the time of assembly, the directives are written before the class declaration inside Code Behind. Calling QP7 Objects from Code Behind ShowObject method is normally used to call other QP7 objects when working inside Code Behind. The following two versions of this method exist.

ShowObject(string name, System.Object sender) ShowObject(string name)

The version with two arguments will add the specified object to the end of the collection of the control specified as the second argument (the second argument has to be a reference to an instance of a control, not a QP7 object name).

For example, if it’s necessary to load an object into a specific place in Presentation then asp:placeholder control can be used and in Code Behind the following call can be made to ShowObject:

ShowObject (“objectName”, this.FindControl (“placeholderName”))

The version with one argument adds the specified object to the end of the current Page collection and is basically equivalent to the one with two arguments:

ShowObject (“objectName”, QPage)

At the time of assembly, the calls to ShowObject are replaced with calls to ShowControl which actually load assembled QP7 controls (when assembled QP7 objects become QP7 controls – UserControl class controls)

NOTE: the first argument of ShowObject has to be a literal string and not a string variable; otherwise the assembly won’t know which control to load. If you need to load objects dynamically with a string variable as the first argument see the section below. Dynamically Loading QP7 Objects from Code Behind Sometimes it becomes necessary to call objects dynamically where it’s not known at design time which objects should be called. ShowObject method uses object name as the first argument and is replaced by ShowControl at assembly time with first argument becoming control file name (.ascx) (template .NET name and format .NET name might also be present depending on the type of call).

For this purpose ShowControl method should be used. Its signatures are similar in syntax and in purpose to ShowObject:

ShowControl (string name, System.Object sender) ShowControl (string name)

However, this method needs the file name of the assembled control, it can be obtained by calling the GetObjectFullName method:

string GetObjectFullName(string templateNetName,string objectNetName, string formatNetName)

The first and third arguments, templateNetName and formatNetName, can be specified as empty strings “” if they’re not necessary.

So, your call to ShowControl should look something like this:

ShowControl (GetObjectFullName (“MyTemplate”, “MyObject”, “MyFormat”), this)

The second argument for ShowControl is optional and serves the same purpose as the one for ShowObject.

IMPORTANT: before you start loading objects dynamically you have to make sure that these objects are actually assembled. By default QP7 will only assemble objects that it recognizes via ShowObject or <qp:placeholder> calls. You would need to assemble all objects by setting “Assemble All Objects” in the backend at the “Site Properties” tab. Calling Objects Recursively Recursion, object calling itself, can be achieved by calling ShowObject method for its second parameter passing the reference to the current object. However, it’s not recommended to use recursion with <qp:placeholder /> constructions because it can lead timeouts during page processing. The depth of recursion is limited by the Framework to 32 levels. Passing Data Between Objects (using Values Collection) To pass data between objects it’s recommended to use Values collection in Code Behind.

The following methods can be used to modify or read contents of this collection:

void AddValue(string key, Object value) – to add a value with a specified key; use this method to also hold object references.

string Value(string key) – returns string value of a collection entry with a specified key where single quotes are stripped off – use DirtyValue to return original value or StrValue to return value with doubled single quotes;

string Value(string key, string defaultValue) – returns string value of a collection entry with a specified key; if the entry’s value is equal to empty string “” then defaultValue will be returned;

long NumValue(string key) – returns numeric value of a collection entry with a specified key;

string StrValue(string key) – returns string value of a collection entry with a specified key, where the value’s single quotes are doubled;

string DirtyValue(string key) – returns original, unmodified string value of a collection entry with a specified key;

Hashtable Values – property that returns the actual Values collection implemented as Hashtable Class

Object Values(string key) – returns value of type Object for a specified key.

IMPORTANT: When working with Values collection it’s imperative to carefully plan the order of object calls and setting collection values. The value has to be set in the collection before the object is called. It’s recommended to set collection values and call objects in Code Behind to ensure proper processing sequence. Using Data Binding To Display Data

Often, during development there is a need to quickly output some data in a certain spot of the Presentation section of object. For these cases it’s recommended to use Data Binding or also known as single value data binding

In Presentation pick a spot where data should appear and specify <%# variableName%> something analogous to the following:

<table border=”0” cellpadding=”0” cellspacing=”0”> <tr><td><%#sStr%></td></tr> </table>

In Code Behind declare class member that will output data in Presentation. In a method, like InitUserHandlers, set class member equal to some value.

Code Behind (CS)

public string sStr = ””;

override public void InitUserHandlers(EventArgs e) {

```
sStr= "Hello World!";
```

….. }

Code Behind (VB.NET)

Public sStr as String = “”

Overrides Public Sub InitUserHandlers(e as EventArgs)

```
sStr= "Hello World!"
```

….. End Sub

IMPORTANT: DataBind() should not be called explicitly, because it’s called for the whole format automatically when the assembled control is loaded at runtime. Using Publishing Container By default when a format of a Publishing Container is created, its code is set to a predefined code template (default code can also be set in the backend at the “Object Format” tab by clicking on the “Set Default Values” button):

Presentation

<asp:Repeater id=“container” OnItemDataBound=“OnItemDataBound” OnItemCreated=“OnItemCreated” runat=“server”> <HeaderTemplate> </HeaderTemplate>

<ItemTemplate>

</ItemTemplate>

<FooterTemplate> </FooterTemplate> </asp:Repeater>

From the above code it’s clear that a Repeater control with empty templates and defined event handlers for ItemDataBound and ItemCreated is created. Custom code should be placed inside template tags: <HeaderTemplate>, <ItemTemplate> or <FooterTemplate>. Whatever is inside <ItemTemplate> will be repeated for every record that is retrieved from the CMS because the Repeater control is bound to Data property which contains the retrieved data in the form of DataTable class. The code inside other template tags will be executed only once.

Of course, the repeater control can be replaced by any other code or control. It’s purpose is to provide

a sufficient start to development.

Further, in order to publish Field data of each record databinding can be used, using <%# %> tags. For example, inside <ItemTemplate> (fields can be dragged-and-dropped in Visual Studio or inserted by using information box in the backend):

<ItemTemplate> <%# Field[1)], “Title”)%> – <%# Field[2)], “Date”)%> </ItemTemplate>

The pre-generated Code Behind is going to look like the following:

Code Behind

using System; using System.Collections; using System.Web.UI.WebControls;

protected Repeater container;

protected void LoadContainer(Object sender, EventArgs e) {

```
container.DataSource = Data;
```

}

override public void InitUserHandlers(EventArgs e) {

```
LoadContainer(this,e);
```

}

protected void OnItemDataBound(Object sender, RepeaterItemEventArgs e) {

}

protected void OnItemCreated(Object sender, RepeaterItemEventArgs e) {

if[3)] { *please use if needed* } }

The DataTable instance generated during the data retrieval process is available via Data property of Publishing Container.

There are also unimplemented Repeater event handlers: OnItemDataBound и OnItemCreated.

NOTE: There is no need to declare controls in Code Behind if you’re using “Assemble Using Partial Classes” option (set at “Site Properties”)

IMPORTANT: In order to use nested Publishing Container objects (one object calling another) or pass information between objects ItemCreated or ItemDataBound should be used. To add a nested object in Presentation for every retrieved record inside <ItemTemplate> tag, an <asp:placeholder id=”myPlaceHolder”/> control should be placed there; and in the code behind ShowObject should be called inside OnItemCreated or OnItemDataBound. The second argument of ShowObject should include the reference to the <asp:placeholder id=”myPlaceHolder” />. The following example shows how a nested object can be called with passed-in record id via AddValue call.

Presentation

<ItemTemplate> <%# Field[4], “Title”)%> – <%# Field[5], “Date”)%> <asp:placeholder id=”myPlaceHolder” runat=”server” /> </ItemTemplate>

Code Behind

protected void OnItemCreated(Object sender, RepeaterItemEventArgs e) { if[6] {

```
 AddValue (“filterValue_for_MyNextObject”,
Field(((DataRow)(this.Data.Rows[e.Item.ItemIndex])), "content_item_id");
```

```
 ShowObject(“MyNextObject”,e.Item.FindControl(“myPlaceHolder”));
```

} }

Using Field Method in Publishing Container

Method Field is used to read field values of retrieved records. There are several variations of this method with the following syntax:

• Field(DataRowView pDataItem, string key) returns string for a DataRowView class with a specified field name as the second parameter, key.

• Field(DataRowView pDataItem, string key, string defaultvalue) analogous to the previous but will also return defaultValue if the actual field value is an empty string “”

• Field(DataRow pDataItem, string key) • Field(DataRow pDataItem, string key, string defaultvalue) analogous to the previous two methods except that it works with DataRow class instead of DataRowView

In Presentation a call to Field would be similar to the following using common databinding syntax:

<%# Field[7], “content_item_id”)%>

In Code Behind:

Field[8], “content_item_id”)

, where Data property is used in conjunction with e as RepeaterItemEventArgs class commonly used in ItemDataBound and ItemCreated events.

Using Content Library in Publishing Container Please refer to this section. Using Site Library in Publishing Container Please refer to this section. Using SQLDataSource Class for Data Access Please refer to this section. Using Output Caching in Objects

To cache object output and all of its child objects a standard .NET <%@ OutputCache%> directive can be used. The directive has to be placed at the very top of the Presentation:

<%@ OutputCache Duration = “10” VaryByParam = “none” %>

Duration attribute – the time, in seconds, that the QP7 object as “user control” is cached. Setting this attribute on a page or user control establishes an expiration policy for HTTP responses from the object and will automatically cache the page or user control output.

VaryByParam attribute – a semicolon-separated list of strings used to vary the output cache. By default, these strings correspond to a query string value sent with GET method attributes, or a parameter sent using the POST method. When this attribute is set to multiple parameters, the output cache contains a different version of the requested document for each combination of specified parameters. Possible values include none, an asterisk (*), and any valid query string or POST parameter name.

For other options for output caching please consult Microsoft .NET Framework documentation: http://msdn2.microsoft.com/en-us/library/hdxfb6cy.aspx Using .NET validate Request

By default .NET Framework has a built-in mechanism for checking incoming requests (GET or POST) where certain content is not allowed, like HTML code. If it’s necessary to accept html submissions the following .NET entry should be made to web.config:

<pages validateRequest = “false” />

Since QP7 Onscreen technology usually has HTML submissions the request validation should be disabled for the “stage” site. Page encodings

All QP7 pages are assembled with certain encodings. These encoding settings are defined for each page, at “Page Info” tab or as a property in Visual Studio Add-on. QP7 developers need to make sure that these encodings match the settings of the globalization section of web.config

<globalization

```
          requestEncoding="utf-8"
          responseEncoding="utf-8"
```

/>

There may be issues with field values of submitted site forms if these settings are different.

[1)] , [2)] , [4)] , [5)] , [7)] (DataRowView)(Container.DataItem
[3)] , [6)] e.Item.ItemType == ListItemType.Item) || (e.Item.ItemType == ListItemType.AlternatingItem
[8)] (DataRow)(this.Data.Rows[e.Item.ItemIndex]