

QUANTUMART



---

# Developer Guide

**QP7.Framework**

## Table of Contents

<b>Introduction .....</b>	<b>4</b>
---------------------------	----------

<b>QP7.Framework object model.....</b>	<b>5</b>
--	----------

Template .....	5
Object (Template or Page).....	5
Format .....	6
Page.....	6
Overriding Template Objects.....	6
Assembly.....	6
Code Behind / Presentation.....	7
.NET Languages .....	7
.NET Names.....	7
.NET Assembly Settings (Site Properties).....	8
ViewState (Template, Object, Page Properties) .....	9
Mobile Platform (Template Properties) .....	9
Publishing Container.....	9

## **General Programming Principles of QP7.Framework 11**

Types of Object Calls (TOF, OF, O, TO).....	11
Recursive Object Calls.....	12
Object Types .....	12
Passing Parameters or Data from one Object to Another .....	13
Using Publishing Container .....	13
General Description of “Field” Method.....	14
Using Content Library .....	14
Using Site Library.....	14

## **ASP.NET Programming Techniques Specific to QP7.Framework..... 15**

Using IntelliSense – Making API Calls .....	15
Making Object Calls – Presentation .....	15
Making Object Calls – Code Behind .....	16
Code Behind Structure and Sequence of QP7 Events .....	16
Class and Namespace Declaration .....	17
QP7 Class Inheritance.....	17
Import or Using Directives .....	18
Calling QP7 Objects from Code Behind.....	18

Dynamically Loading QP7 Objects from Code Behind .....	19
Calling Objects Recursively .....	19
Passing Data Between Objects (using Values Collection) .....	20
Using Data Binding To Display Data .....	20
Using Publishing Container .....	21
Using Field Method in Publishing Container .....	24
Using Content Library in Publishing Container .....	24
Using Site Library in Publishing Container.....	24
Using SQLDataSource Class for Data Access.....	25
Using Output Caching in Objects .....	25
Using .NET validate Request.....	25
Page encodings.....	25
Manually Creating and Initializing QP7 Pages.....	26

## **Setting Up and Using Integrated Backend Permissions..... 27**

Setting up IBP.....	27
Using IBP to assign permissions via API .....	29

## **Working with QP7.Framework outside of QP7 Development Environment..... 30**

## **Creating and Working with Backend Custom Tabs ... 31**

## **Sharing Code Between Sites, Virtual Applications, Templates..... 34**

## Introduction

QP7.Framework provides a programming platform with a templating environment that allows developers to quickly and easily create and manage CMS-enabled websites. The platform supports two Microsoft technologies: Classic ASP and ASP.NET.

The purpose of this document is to describe the general principles of programming using QP7.Framework, including explanation of terminology, various options and system settings, syntax for invoking QP7 objects, usage examples. This document, however, is not meant to be used as an API reference guide nor as a CMS feature/user interface reference guide. For these types of information please see the API Reference Guide and User Guide, respectively.

## QP7.Framework object model

The key development principle in QP7.Framework is the usage of its unique templating model which at its higher level consists of templates, template objects, template pages, page objects. Utilizing template and template objects developers can create a general page architecture which then can be customized for each template page by overriding template objects with page objects.

### Template

A template is a logical entity that contains template objects and template pages. A combination of template body and template objects creates a general layout and functional framework to be used by template pages. Subsequently, each template page inherits the template structure and may override any of its template objects with page objects to provide a modified layout or functionality. If a page or a series of pages require a significant different layout or functionality it's recommended to create another template to host these pages.

A template always has a template body which contains the initial starting code of the page, e.g. `<html><head></head><body>...</html>`. Intertwined with calls to template objects that contain the layout and functionality for page's header, body and footer

Generally a template is made for every design variant if these variants differ significantly. Template body is a frame for the pages created. Generally a template body includes `<html>`, `<head>`, `<body>` and other tags that are used on the higher levels of a web-page layout. Each page of a site belongs to one and only one template. A set of objects that are called from the template body and contain logics and layout of the lower levels of the web-page is created for a template.

Alternatively, templates may be used just to store template objects for modularization purposes.

### Object (Template or Page)

Object is a logical entity that contains 1 or more formats, which in turn contain code. Generally an object performs one specific task and contains a part of page layout. An object can be created at the template level or at the page level. The page level objects may override template objects to provide own code.

There are several types of objects, each optimized for a specific task. The main types of objects are Generic and Publishing Container. The Generic type may contain any code and does not contain auto-generated code for accessing content. On the other hand, Publishing Container type has auto-generated code for receiving content from QP7 Content Areas.

Some types of objects can contain various parameters that help to easily modify object's behaviour.

## **Format**

A format is a logical entity that contains object's code – layout and/or business logic. An object may have several formats. Code in formats may contain calls to other objects. Its physical equivalent is a UserControl Class; consequently each format has a presentation and a code behind that when assembled become ascx and ascx.vb/ascx.cs respectively.

## **Page**

A page is a logical entity that contains page objects; its physical equivalent is an .aspx page. During the process of page assembly an .aspx page is created with all of the necessary .ascx and ascx.vb/.ascx.cs that are based on the code contained within object formats.

## **Overriding Template Objects**

Any template object may be overridden by a page level object which subsequently can provide its own code. A page object overriding a template object will receive the same name as the template object. Thus, during the page invocation the overridden page object will be called instead of the template object. However, if in code the name of the object is prefixed with its template name then the template object will always be called.

## **Assembly**

Assembly is a process when physical entities corresponding to logical entities of QP7.Framework are created. For instance, all pages will become .aspx files while all object formats and template body will become UserControl classes – .ascx and .ascx.vb/.ascx.cs files.

When a page is assembled, by default (this can be changed), only the objects which names are explicitly mentioned in the code will be assembled.

## **Code Behind / Presentation**

Each format contains Presentation and Code Behind and correspond to the same concepts of .NET Framework. Generally, Presentation is designated to contain various elements of user interface including HTML code and ASP.NET tags where as Code Behind should contain business logic and implementation of various stages of object or page processing like event processing code. The syntax for calling QP7.Framework API methods or invoking QP7 objects are slightly different for Presentation and Code Behind.

## **.NET Languages**

QP7.Framework supports two languages for working with .NET Framework: C# (by default) and VB.NET. The language can be selected for each format or for each template body. During the assembly process the language selection determines file extension of resulting user controls.

## **.NET Names**

All pages of a site are always assembled using VB.NET with .aspx and aspx.vb file extensions.

All formats and template bodies are assembled in the form of user controls with .ascx for Presentation and .ascx.cs or .ascx.vb for Code Behind.

Further, .aspx pages do not contain any code created by developers; instead, each object call causes QP7.Framework to dynamically load an appropriate assembled format (or user control) at run-time.

The first control that gets loaded is the template body followed by the other objects according to the way object calls were nested.

When creating a template, object or format, .NET Template Name, .NET Object Name or .NET Format Name can be specified respectively.

These fields have to contain only Latin characters and cannot contain any white space. During the process of assembly, these names are used to name the resulting files. For example, when an object with .NET Name “myObject” containing a format “myFormat”

is assembled the resulting file name will be myObject\_myFormat.ascx (and corresponding code behind file - myObject\_myFormat.ascx.cs). The resulting Class will be called “myObject\_myFormat”.

If .NET Names are not specified then QP7.Framework attempts to create them automatically from regular object names by converting white space into underscore character “\_”.

Also, during the assembly process the following naming rules are applied for the resulting controls:

- if the default format “myFormat” is assembled for object “myObject” then QP7.Framework will create two controls with the same code, one will be named myObject.ascx (and the corresponding code behind file) and the other one myObject\_myFormat.ascx (and the corresponding code behind file).

- if any other format is assembled except for the default format then only one control will be created and named myObject\_myFormat.ascx (and the corresponding code behind file).

## **.NET Assembly Settings (Site Properties)**

The Site Properties tab contains a series of specific fields that influence how and where QP7.Framework assembles controls or pages.

**.NET Framework Version** – affects which settings are available to objects of type Publishing Container and whether or not Partial Classes can be used.

**Force Assemble** – enabling this option will cause QP7.Framework to always assemble objects whether or not any code changes were made. Otherwise, only elements that had any changes will be assembled to expedite the assembly time.

**Assemble All Objects** – causes all elements to be assembled whether or not they are explicitly mentioned in the code. The *force assemble* option is also taken into account.

**Assemble Using Partial Classes** – only available if .NET Framework 2.0 or higher is chosen. It sets whether or not controls are assembled using Partial Classes syntax.

**Assembly File Location (Live and Stage)** – specifies physical paths to folder bin for the current site for live and stage locations. Quantumart.dll file is copied to this folder when the site is updated or assembled at the “Site Properties” tab.



## ViewState (Template, Object, Page Properties)

Template, Object and Page properties contain “Enable ViewState” option. For each of the elements it determines whether or not EnableViewState attribute is set to true or false. The settings on the lower levels override the ones on the higher levels (e.g. object overrides page, page overrides template).

## Mobile Platform (Template Properties)

Template Properties contains “Designed for Mobile Devices” option. If it’s selected then all objects (including the template body) and all pages will be assembled using QP7 Mobile Classes which in turn inherit from Mobile Classes of .NET Framework.

The table below shows the inheritance hierarchy for each assembled QP7 entity:

QP7 entity	QP7 Regular Class	QP7 Mobile Class	Parent Class
<b>Page</b>	QPage		System.Web.UI.Page
		QMobilePage	System.Web.UI.MobileControls.MobilePage
<b>Object</b> (except publishing container)	QUserControl		System.Web.UI.UserControl
		QMobileUserControl	System.Web.UI.MobileControls.MobileUserControl
<b>Publishing Container Object</b>	QPublishControl		QUserControl
		QMobilePublishControl	QMobileUserControl

All QP7.Framework methods and properties are the same for regular and mobile objects.

## Publishing Container

Publishing Container serves as a means of accessing and displaying Content Data. By default the Presentation (each object has a Presentation and Code Behind) contains a .NET Repeater Control, however, any other control can be used as well.

Among basic properties of the Publishing Container like number of articles to display, which article to start from, etc., there are several options specific to .NET 2.0 (QP7 site has to be set to run under .NET 2.0 at the “Site Properties” tab).

**Data Access Class** – specifies type of .NET class that is used to receive data from the CMS. There are two options for DataReader and SQLDataSource classes. By default for 1.1 Assemblies DataReader is used and for 2.0 it's SQLDataSource (DataReader is the only option available for 1.1 Assembly)

Using both classes will result in receiving a DataTable class instance.

**Enable Data Caching** – If SQLDataSource is selected as 'Data Access Class' then this option will appear to enable caching of data.

In turn, enabling 'Enable Data Caching' will allow to specify 'Duration' – number of minutes the data will be cached and also 'Enable Cache Invalidation'

**Enable Cache Invalidation** – enabling this option will cause the cached data to be renewed upon any changes to the underlying CMS content.

QP7 implementation of this functionality is compatible with SQL Server 2000 and 2005 and involves the notification mechanism of SQL Server and the polling of .Net Platform.

Additional configuration steps need to be performed in order to setup cache invalidation:

### 1. Web.Config

To <system.web> section add <caching> tag with something like the following:

```
<caching>
  <sqlCacheDependency enabled="true" pollTime="1000" >
    <databases>
      <add name="qp_database" connectionStringName="qp_database" />
    </databases>
  </sqlCacheDependency>
</caching>
```

- connectionStringName should reference the "qp\_database" connection name inside <connectionStrings> tag which is used to connect with your QP7 database when NET 2.0 assembly is enabled.

```
<ConnectionStrings>
  <add name="qp_database"
    connectionString="Password=publishing;User ID=publishing;Initial
    Catalog=publishing;Data Source=localhost"
    providerName="System.Data.SqlClient" />
</ConnectionStrings>
```

### 2. Register database and tables (only SQL 2000)

SQL 2000 also requires to register the database and the underlying tables with the help of aspnet\_regsql utility:

```
>aspnet_regsql.exe -S "localhost" -E -d "publishing" -ed
>aspnet_regsql.exe -S "localhost" -E -d "publishing" -et -t "content_271"
```

- This utility is located in the .NET Framework folder  
(C:\WINDOWS\Microsoft.NET\Framework\<version of .NET Framework>)

## General Programming Principles of QP7.Framework

The items described in this section are applicable to all types of assemblies supported by QP7.Framework.

### Types of Object Calls (TOF, OF, O, TO)

Every template or page object call maybe specified in different ways, using various syntax structures.

Let's assume there is a template "myTemplate" containing object "myObject" that has 2 formats – "myFormat1" и "myFormat2" where format "myFormat1" is the default format.

There is also a page, "defaultPage", which contains an object that overrides "myObject" object of the "myTemplate" template.

Below are described different ways of calling objects from any page object of this page:

"myTemplate.myObject.myFormat1" – calls a specific format of the template object (so called TOF object call). In this case "myFormat1" format belonging to "myObject" object of "myTemplate" template will be invoked despite the fact that this object is overridden by a page object.

"myTemplate.myObject" – calls a specific template object without specifying the format name (so called TO object call). In this case the "myFormat1" format will be invoked since it's the default format of "myObject" object of "myTemplate" template.

"myObject" – calls an object without specifying template or format name (so called O object call). In this case "myFormat1" format of the overriding "myObject" page object of "defaultPage" page will be invoked. If "defaultPage" page would not have an object overriding "myObject" template object then "myFormat1" format of "myObject" object of "myTemplate" template would be invoked instead.

The rule for creating object calls without specifying format is the same for template and page objects – the default format will be used, otherwise the specified format.

“myObject.myFormat1” – calls a format without specifying the template name, only object and format names (so called OF object call). In this case “myFormat1” format of the overriding “myObject” page object of “defaultPage” page will be invoked.

Page objects belonging to a certain page cannot be called by page object belonging to another page. Also the same prohibitive rule applied for calling page objects from a template object belonging to another template.

In order to reuse objects across different templates or objects must be template objects. QP7 provides a way to promote page objects to the template level for this purpose.

## Recursive Object Calls

Recursive object calls are used by objects to call themselves one or multiple times. For example, they can be used to construct hierarchical trees from articles of a certain Content Area. Careful considerations (conditions for initiating and terminating recursive calls, construction of proper filters if using Publishing Container) must be taken when making recursion is used. The maximum depth of recursive calls is limited to 32 nested calls to one object by QP7.Framework.

## Object Types

QP7.Framework provides several types of objects. Among them, Generic and Publishing Container are the main types.

Generic – basic object type that does not contain any kind of special parameters and is usually used to encapsulate business logic, layout code or calls to other objects. This type of object does not provide any automatic connections to receive articles from QP7 Content Areas.

Publishing Container – designed to automatically receive articles from QP7 Content Areas. It’s usually used to display articles in the form of html. This object type contains a number of configurable parameters that allow developers to define the resulting data set. The configuration of these parameters is similar to constructing a SQL SELECT statement.

The main parameters are:

**Content** – used to select which Content Area will supply the data (similar to SQL FROM).

**Allow Dynamic Content Change** – if enabled specifies ‘Dynamic Content Variable’ used to specify the name of a variable (created by using AddValue() function) which at run time would contain the name of a Content Area. This is useful for code reuse where the Publishing Container Object is set up to access several similarly structured Content Areas depending on the business logic.

**Dynamic Content Variable** – see above.

**Filter** – used to specify free-form filter using SQL syntax (similar to SQL WHERE). Combination of static strings and variables may be used here.

**Start from** – used to specify the starting record of the resulting data set. The records before this record will not be retrieved. A static number or a variable can be used here. The setting becomes very useful in conjunction with ‘Show Articles’ to define create paging functionality.

**Show Articles** – used to specify how many articles should be retrieved. A static number or a variable can be used here.

**Order** – used to specify the sorting order of the resulting data set. Multiple fields can be set here in either ascending or descending order (similar to SQL ORDER BY).

Access to retrieved data in the form of a DataTable class and via QP7 API is available in the Code Behind and Presentation sections of the Publishing Container object.

## Passing Parameters or Data from one Object to Another

When working with objects, it is often the case that some kind of data needs to be stored somewhere to be used in another place. In addition, reading data from HTTP request (query string, form fields, uploaded files) is frequently required. To simplify these tasks QP7.Framework provides “Values” collection.

Upon page load, “Values” collection is populated with various parameters passed in via HTTP request and subsequently it becomes available for access by object formats. QP7 developer can control the contents of “Values” collection by adding new or modifying existing entries by utilizing the framework API, methods like Value, AddValue, NumValue, StrValue, DirtyValue and others.

## Using Publishing Container

In order to effectively utilize Publishing Container object type it’s necessary to get familiar with its main methods for displaying data (Please see QP7.Framework API).

## ***General Description of “Field” Method***

Field method – is used to access one of the fields of a selected record.

In ASP.NET, for C#, the method is invoked by using this syntax:

```
Field(((DataRowView) (Container.DataItem)), "Title")
```

And for VB.NET:

```
Field(CType(Container.DataItem, DataRowView), "Title")
```

In the end, the method returns the value of field “Title” for the current record. If the field value does not need to be modified or formatted then it’s usually sufficient to use this method in the Presentation section with databinding (databinding is automatically invoked by QP7.Platform).

## ***Using Content Library***

QP7.Framework allows files, like PDF or image, to be stored in fields of type file or image. These files are stored in special content folders on the web server.

In order to publish link to these files inside Publishing Container object types it’s not just enough to call Field method, as it contains only the name of the file, it’s necessary to call “ContentUploadURL()” method to receive the virtual (absolute path can be configured at “Site Properties”) path to the file. For example, the syntax in C# would be the following:

```
ContentUploadURL() + "/" +  
Field(((DataRowView) (Container.DataItem)), "Title")
```

## ***Using Site Library***

QP7.Framework also provides a general file repository for a site, links to which can be published in any object format. Any file type can be stored in the Site Library – CSS, Images, Word, PDF, etc.

In order to publish links to these files you would need to call “upload\_url” property in order to establish the path to the Site Library and only then call Field method.

For example, C#:


```
upload_url + "/" + "myfile.pdf"
```

Using “upload\_url” property is recommended rather than hard-coding path to files in case path to Site Library might change in the future. The virtual path to Site Library is configured at “Site Properties” (Upload URL, Use Absolute Upload URL, Upload URL Prefix).

# 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>
    <p>
        <%# Field(((DataRowView) (Container.DataItem)), "Title")%> --
        <%# Field(((DataRowView) (Container.DataItem)), "Date")%>
    </p>
</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((e.Item.ItemType == ListItemType.Item) || (e.Item.ItemType
        == ListItemType.AlternatingItem))
        // {
        //     ***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 and 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>
    <p>
        <%# Field(((DataRowView) (Container.DataItem)), "Title")%> --
        <%# Field(((DataRowView) (Container.DataItem)), "Date")%>
        <asp:placeholder id="myPlaceHolder" runat="server" />
    </p>
</ItemTemplate>

```

#### Code Behind

```

protected void OnItemCreated(Object sender, RepeaterItemEventArgs e)
{
    if((e.Item.ItemType == ListItemType.Item) || (e.Item.ItemType ==
    ListItemType.AlternatingItem))
    {
        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(((DataRowView) (Container.DataItem)), "content_item_id")%>
```

In Code Behind:

```
Field(((DataRow) (this.Data.Rows[e.Item.ItemIndex])), "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.

## Manually Creating and Initializing QP7 Pages

Sometimes it might be necessary to create a QP7 page manually (without standard QP7 page creation functionality) to be used with or without QP7 assembled objects. A good case for creating QP7 pages manually is the one where Master Pages .NET technology has to be used along with QP7.Framework objects.

In order for a page to work the following two conditions have to be met:

- Page Class has to inherit from QPage or QMobilePage classes of QP7.Framework
- The resulting page has to be initialized by calling on of the three methods:

```
public void Initialize(int siteId, string uploadUrl, string
siteUrl, string pageFileName, string templateNetName)
```

```
public void Initialize(int siteId, string uploadUrl, string
siteUrl, string pageFileName, string templateNetName, Hashtable
pageObjects)
```

```
public void Initialize(int siteId, string uploadUrl, string
siteUrl, string pageFileName, string templateNetName, Hashtable
pageObjects, Hashtable templates)
```

Parameters:

**siteId** – id of QP7 website

**uploadUrl** – upload url used by QP7 website (see “Site Properties” in the backend)

**siteUrl** – website url, will be used to calculate paths to QP7 controls and also for QP7-managed links.

**pageFileName** – name of the page file (e.g. default.aspx). This will be used to calculate the name of the folder where QP7 controls are located. A file name default.aspx will be converted to “page\_controls\_default.aspx”.

**templateNetName** – .NET name of the QP7 template where this current page could reside. This can be used to calculate template folders if templates parameter is provided. Empty string can be specified if

**pageObjects** – Hashtable with a list of file names and file folders for assembled QP7 objects (e.g. “faq.ascx”, “faq\_folder”. If object/control does not have a folder, empty string is supplied for folder name.

**templates** – Hashtable with a list of .NET template names and file folders for QP7 templates (e.g. “Main\_Template”, “main\_template/subfolder/subfolder”. If template does not have a folder, empty string is supplied for folder name.

## Setting Up and Using Integrated Backend Permissions

Often certain sections of a website require visitors to login in order to see protected content or content that only pertains to the logged-in visitor/user. Integrated Backend Permissions (**IBP**) serves as a way to expose QP7 Backend Permissions System to be used for front-end websites.

Q7 Backend Permissions module allows content managers to set up article permissions in the backend for various groups or users using backend’s graphical user interface. At the same time, with a click of a button, developers can enable IBP for Publishing Container object types and be able to filter content based on the level of permissions specified for each article.

### Setting up IBP

In order to use IBP the following steps need to be taken:

- Define variables that will be used by your site to store logged-in user\_id and logged-in group\_id for each .NET language (C# and VB.NET) in the “Q-Publishing Configuration.xml”:

```
<app_var app_var_name="security_UID_varname_VB">Session("qp_UID")
</app_var>
```

```
<app_var app_var_name="security_GID_varname_VB">Session("qp_GID")
</app_var>
```

```
<app_var
app_var_name="security_UID_varname_CSharp">Session["qp_UID"]
</app_var>
```

```
<app_var
app_var_name="security_GID_varname_CSharp">Session["qp_GID"]
</app_var>
```

Each .NET language supported by QP7.Framework (VB.NET and C#) requires its own entry since syntax maybe different between the languages.

By default Session("qp\_UID") and Session("qp\_GID") for VB.NET or Session["qp\_UID"] and Session["qp\_GID"] for C# will be used.

- Refer to the .NET QP7 API documentation for full reference to QP7 Permissions class.
- Create a login form or use an existing one with user name and password to authenticate users against the Backend Permissions Module.
- Use `Quantumart.QPublishing.Permissions.AuthenticateUser(string username, string password)` method to return `user_id` as integer and store it inside your variable defined in "Q-Publishing Configuration.xml", e.g. `Session["qp_UID"]`. Make sure that the returned `user_id` is greater than zero (`user_id > 0`) which means that the login credentials are correct and user exists.

```
Session["qp_UID"] =
Quantumart.QPublishing.Permissions.AuthenticateUser(username,
password);
```

Alternatively, you can store `group_id` in `Session["qp_GID"]` and set `Session["qp_UID"]` to null. This way the Framework will use `group_id` to filter out records that fall out of the range of the specified permission levels.

- If visitors have to be able to create new accounts create a form (include fields: User Name, First Name, Last Name, Email Address) and use this method:  
`Quantumart.QPublishing.Permissions.AddUser(string username, string password, string First_Name, string Last_Name, string Email)`

```
Quantumart.QPublishing.Permissions.AddUser(username, password,
First_Name, Last_Name, Email);
```

When a new user account is created it becomes part of the users list in the backend but with disabled backend access (at "User Info" tab, "options" field is set to "disabled").

- Normally, new users are placed in one or more groups by calling this method:  
Quantumart.QPublishing.Permissions.AddUserToGroup(int userId, int groupId)

```
Quantumart.QPublishing.Permissions.AddUserToGroup(int userId, int groupId);
```

To find a group use this method:

```
DataTable dt = GetGroupInfo(group_name); //group_name is a string
int group_id = dt.Rows[0]("group_id");
```

- Enable “Use Backend Integrated Security” and define “Start Level” and “End Level” (“deny”, “list”, “read”, “modify”, “full-access”) for each Publishing Container object type that will be filtering out records based on IBP (in the backend at “Object Parameters” tab, in Visual Studio Add-on

## Using IBP to assign permissions via API

QP7 supports several API methods for assigning permissions to individual content items (records or articles) or to Content Areas. This is especially useful in cases where sites contain user-generated content which can be permitted to be viewable or editable by a user/group to other users/groups.

QP7 also supports nested groups, where a child group inherits parent group permissions, affecting child group users. There are methods available to establish parent-child associations between groups.

There are many methods available to work with IBP via API. The most notable ones are listed below (see QP7 .NET API for the full list):

- AddUserToItemPermission(int userId, int itemId, int permissionId);
- AddGroupToItemPermission(int userId, int itemId, int permissionId);

adds user or group to a content item (article) with specified permission level (as permission level id)

- GetPermissionLevels()

returns a DataTable with a list of permission levels and permission ids. Permission\_level\_id is needed to be used for methods like AddUserToItemPermission().

```
the returned fields are: permission_level_id, permission_level,  
permission_level_name
```

- `AddChildGroupToParentGroup(int parent_group_id,int child_group_id)`

```
Creates parent-child associations between groups
```

## Working with QP7.Framework outside of QP7 Development Environment

Even though QP7.Framework and its development platform provide a very robust and fast way to create custom functionality, sometimes it might be required to use QP7 API outside of its development environment.

There are two such examples that come with the demo site, with and without inheritance from QP7 classes. The source code for them is available in the folder `\\qp_demo_net\api_example`.

Both examples can viewed inside the “backend” via custom tabs functionality. The tabs names are: “API Example (no inheritance)” and “API Example (with inheritance)”.

## Using External Modules in QP7

### *Controls*

To use your own controls follow regular .NET syntax to register tag prefix and then create an instance of your control. The tag prefix registration should appear at the top of the Presentation code inside a QP7 object format:

```
<%@ Register TagPrefix="myPref" TagName="myName"  
Src="myControl.ascx"%>  
...  
<myPref:myName runat="server" />
```

You can place your controls into a directory of your choice.

If you’re using compiled classes (assembly with .dll file extension) then place them into the bin folder (live and stage) and use the following syntax at the top of the Presentation code inside a QP7 object format:

```
<%@ Register TagPrefix="myPref" NameSpace="MySpace.MySubSpace"  
Assembly="MyAssembly" %>
```

## ***Classes***

To use classes place your assembly files (.dll) into the bin folder of your site (live and stage locations). For unassembled / uncompiled text files (.cs or .vb) use "App\_Code" directory.

## ***VS Add-on for QP7***

VS Add-on will automatically download everything that is located in your "bin" folder so that you can use the intellisense feature of Visual Studio.

# **Creating and Working with Backend Custom Tabs**

Backend interface can be extended with the help of Custom Tabs.

Custom Tabs allows administrators/developers to extend the functionality and user interfaces of QP7's backend administrative module. For example, using QP7.Framework you can build wizard-like interfaces for managing several types of QP7 content within one highly customized application.

A Custom Tab links the backend with custom applications built on QP7.Framework or any other development platform. Clicking on a custom tab loads an application in the work frame of the backend.

Once a user clicks on a Custom Tab the backend will pass certain query-string parameters (backend\_sid) to the custom application, which in turn can call a QP7.Framework method to transparently authenticate the backend user as the application user. Essentially, the application can find out the id of the user logged in to the backend. (This is very useful if using "Backend Integrated Permissions" or QP7.Framework applications assembled in stage mode. Pages assembled in stage mode automatically call this method).

To create a custom tab start by going to "Custom Tabs" tab and:

1. right click on any of the Backend's tabs
2. select to create a child tab or a sibling tab
3. fill out the form by providing "Tab Name", "URL" or your application, "Tab Group", "Hide in Tree"
4. once the custom tab is created its fields can be edited by right clicking on the tab and selecting "modify"

5. use the right-click menu to delete custom tabs or create sub child or sibling tabs by.
6. custom tabs, once created, can also be moved by using drag-and-drop functionality (simply drag your custom tab and drop it as a child or a sibling of any other tab). To drop as a sibling position the custom tab underneath a tab's arrow. To drop as a child position the custom tab underneath a tab's name.

## Debugging

QP7 currently does not offer a true debugging environment where break points and stepping in line by line is possible (see this [link for a work-around](#)). QP7 code is kept in the database and through the process of assembly code is transformed into .NET controls with presentation and code behind files (ascx and ascx.c or ascx.vb). QP7 assembled .NET pages – aspx – cannot be edited and are closed to development (to create your own pages you will need to inherit from QPage class, [follow this link](#)).

In case of errors, once the assembled pages are executed .NET platform will show the compilations source and stack trace in the browser if you have `<customErrors mode="Off" />` tag in the `<system.web>` section set to “off” in the web.config.

.NET debug information will point to a certain line in an assembled file. The path to the file and the file name can be used to determine which object format generated the error. To fix the error, find the object format and then the line where the error occurred.

## Assembled Folder and File Names

Assembled template objects are stored in folder with “template\_controls\_\_TemplateName” naming convention. For template with name “Default” the folder name would be “template\_controls\_\_Default”.

Assembled page objects are stored in folder with “page\_controls\_\_PageFileName\_aspx” convention. For a page with file name “home.aspx” the folder name would be “page\_controls\_\_home\_aspx”.

For assembled file names conventions please [follow this link](#).

## Setting-up debugging using assembled files.

Even though QP7 does not directly support a true debugging environment with break points and line by line step through, it is possible to set up a debugging environment by setting up a project in Visual Studio and using assembled files. Follow this link from



Microsoft to setup Remote Debugging: <http://msdn2.microsoft.com/en-us/library/bt727f1t.aspx>.

The limitation of this technique, of course, is that assembled files are overwritten if QP7 assembly is invoked. So, once debugging is complete, to save your changes, it's important to update QP7 code either through VS Add-On or the backend.

## Sharing Code Between Sites, Virtual Applications, Templates

If you want to share code between sites (for non-Enterprise versions of QP7, each site requires purchasing an additional license) you can keep code in one QP7 site and for each QP7 site you can create a new template with its own folder in that one site. Make sure you copy global.asa and/or web.config to make this folder an application and also to expose this folder as a site in IIS (point your new IIS site to this template folder)

If, however, your applications do not require a new dns (example: [www.mynewsite.com](http://www.mynewsite.com)) but only another application within your site (example: [www.mycurrentsite/mynewapp](http://www.mycurrentsite/mynewapp)) then you don't have to create another QP7 site. For your new app it's just enough to create a new template with a separate folder. You would then need to expose this directory as an application in IIS (right click on directory, go to properties, click on "create" button on the "Virtual Directory" tab).

To reuse the code that you created in another template you would need to call objects using this syntax "TemplateName.ObjectName.[FormatName]" with optional FormatName. You would also have to make sure that once you call another template object "TemplateName.ObjectName", that this object also calls its child objects for the same template using TemplateName.ObjectName notation. For example, calling object TemplateXYZ.Object5 from object TemplateABC.Object1 you have to be certain that if TemplateXYZ.Object5 is calling other child objects residing in TemplateXYZ that it calls them using TemplateXYZ.ChildObject syntax. The same would apply for all nested object calls belonging to the same template. Otherwise if a template name is not specified, QP7 will think that TemplateXYZ objects are trying to call object from TemplateABC.

If you are creating QP7 sites you can enable "share" option for each Content Area (at the "Content Info" tab). This way, your code in the site that has all of the templates will "see" other contents. In addition, it might be the case where you would not know the name of the Content Area that the publishing container should be accessing at design time, instead, the name will be determined at run time. In those cases, please enable "Allow Dynamic Content Change" at the "Object Properties" tab and then specify "Dynamic Content Variable" (you just need to put the name of the variable -- my\_Dynamic\_ContentName), which would have to be provided via -- AddValue "my\_Dynamic\_ContentName", NameOfTheContent.