Developer Tutorial

Velodoc XP Edition ships with 4 ASP.NET Ajax components built on Microsoft ASP.NET Ajax Extensions 1.0, which you can reuse in your own applications:

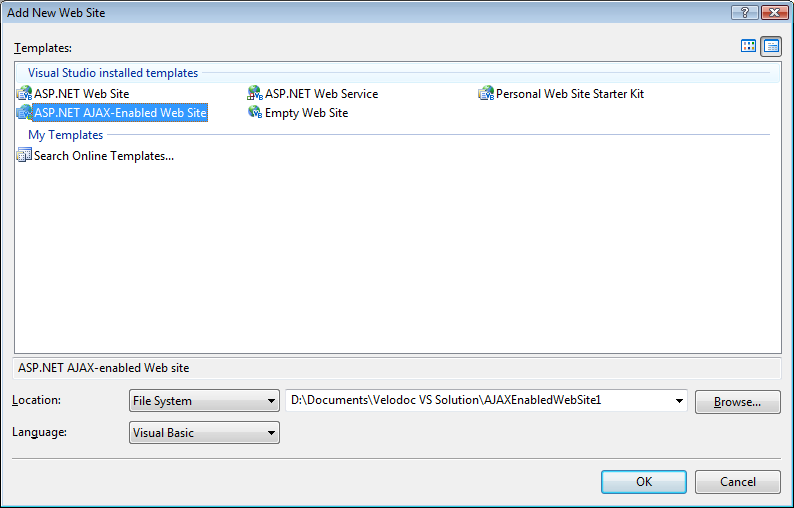
* An InfoBox component which you can use to display messages to the user;
* An ImageList component which you can use to display items with icons. The ImageList component provides for a way to remove items from the list;
* A MultiUpload component to select multiple files to upload ;
* A ProgressReport component to display the progression of a long process.

Velodoc XP Edition also ships with:

* An upload http module to stream the upload of large files and report on upload progress;
* A download handler supporting ETag and Accept-Ranges response headers for resumable downloads.

We recommend that you look at the pages in the Debug subdirectory of the solution web site which are sample minimal implementations of these controls.

# ASP.NET Ajax Extensions



For all server controls in Memba.WebControls to work, you need to have:

1. installed ASP.NET Ajax Extensions 1.0;
2. initially created an ASP.NET Ajax-Enabled Web Site or have configured your web site for ASP.NET Ajax Extensions in web.config;
3. added a reference to the Memba.WebControls.XP assembly to your web site project.
4. added an ASP.NET Ajax Extensions ScriptManager component to your web page hosting the server controls.

Refer to the documentation at <http://www.asp.net/ajax/documentation/> for more information.

Any page you start working with should look like follows:

<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="\_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head runat="server">

<title>Sample Page</title>

</head>

<body>

<form id="form1" runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server" ></asp:ScriptManager>

<% // Page controls go here %>

</form>

</body>

<script type="text/javascript">

<!—-

//Scripts go here

//-->

</script>

</html>

It is also a good idea to add a textarea named “TraceConsole” to display debug output during development:

<textarea id="TraceConsole" cols="100" rows="20"></textarea>

# InfoBox

Start working with a new page as described in the “ASP.NET Ajax Extensions” section of this document. To add an InfoBox component to your page, simply drag and drop the control or add the following code:

<form id="form1" runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server"></asp:ScriptManager>

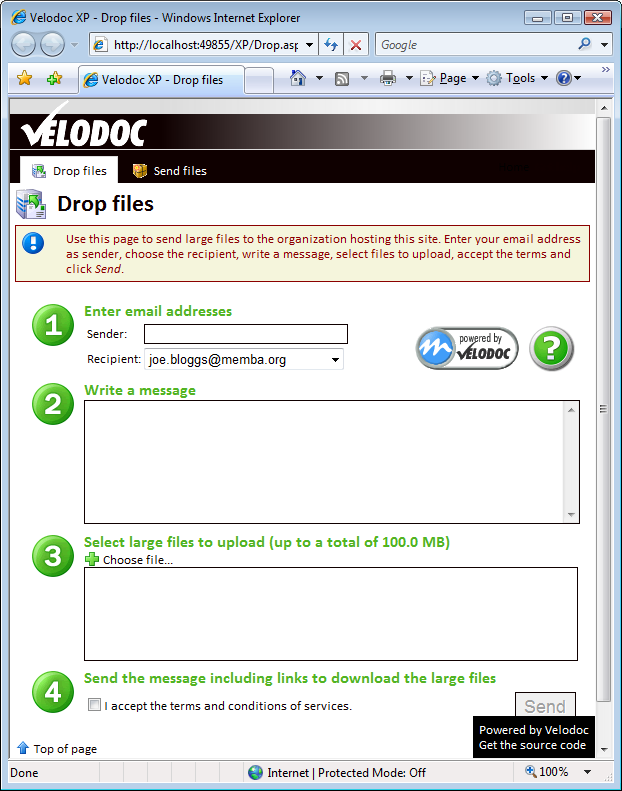
<!-- InfoBox -->

<mbui:InfoBox ID="InfoBox" runat="server" SkinID="sknInfoBox" Text="<%$ Resources:Web.glossary, DropBox\_WelcomeInfo %>" EnableViewState="False" Width="100%"></mbui:InfoBox>

<!-- InfoBox -->

</form>

Using the skin named sknInfoBox from the XPBlack theme, this InfoBox looks like below:



Obviously, you can set the icon and text using server-side code. More interestingly, the icon and text can also be changed client-side using Javascript, for example:

var \_text = "Hello World";

var \_infobox = $find("<%= InfoBox.ClientID %>");

\_infobox.set\_text(\_text);

\_infobox.set\_type(Memba.WebControls.InfoBoxType.Error);

The two last lines can be replaced by the following single line:

\_infobox.setAll(Memba.WebControls.InfoBoxType.Error, \_text);

The following method can also be used to display the message during 0.5s. (or 500ms.) only:

\_infobox.setTemp(Memba.WebControls.InfoBoxType.Warning, \_text, 500);

A sample implementation called InfoBox.aspx is available in the Debug folder of the web site project of the solution.

# ImageList

Start working with a new page as described in the “ASP.NET Ajax Extensions” section of this document. To add an ImageList component to your page, simply drag and drop the control or add the following code:

<form id="form1" runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server">

<Scripts>

<asp:ScriptReference Path="~/ScriptLibrary/Memba.Utils.js" />

</Scripts>

</asp:ScriptManager>

<!-- ImageList -->

<mbui:ImageList ID="ImageList" runat="server"

CssClass="cssList"

ItemCssClass="cssItem"

ItemHoverCssClass="cssItemHover"

ImageCssClass="cssImage"

TextCssClass="cssText"

RemoveCssClass="cssRemove"

RemoveTooltip="Remove from selection"

LinesOfText="2"

Height="100px"

Width="420px"></mbui:ImageList>

<!-- ImageList -->

<textarea id="TraceConsole" cols="100" rows="20"></textarea>

</form>

You may prefer using a skin and resource expressions like in:

<mbui:ImageList ID="FileList" runat="server"

SkinID="sknFileList"

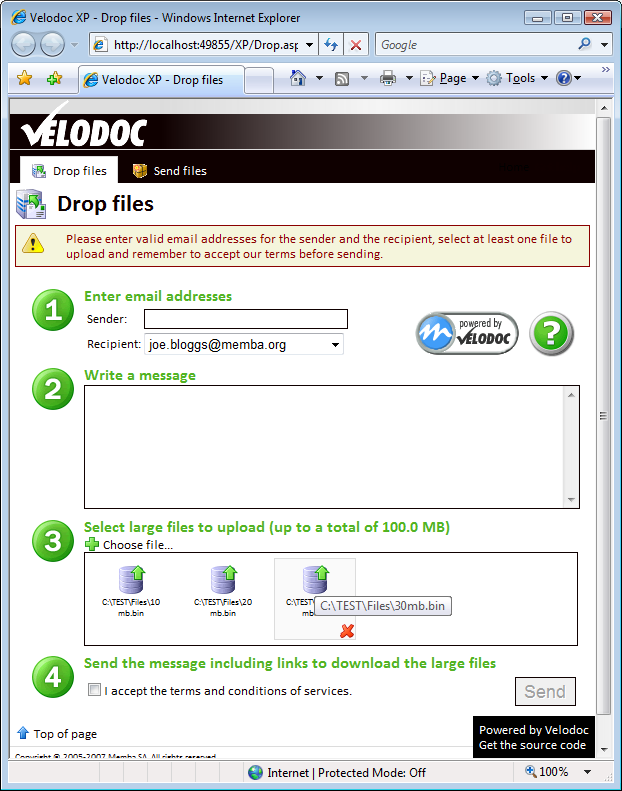
RemoveTooltip="<%$ Resources:web.glossary, sknImageList\_RemTooltip %>"

Height="92px"

Width="100%"></mbui:ImageList>

Adding a reference to Memba.Utils.js is not required to use the ImageList control but we will explain later why we have done it.

Using the sknFileList skin of the XPBlack theme, your ImageList will look like follows after adding items to it:



The ImageList control stores an array of items serialized with JSON in an HTML hidden field:

* When you add items server-side on page load, these items are simply added to the hidden field;
* When the page renders in the browser, the Javascript code built into the control reads the serialized array in the hidden field to render the images;
* When using the client-side API of the control, you simply add or remove items to the array serialized into the hidden field;
* When you post back the page, the hidden field is posted back and the control’s server side code gives you access to the collection if items which it reads from the hidden field.

## ImageList server-side code

Add the following statement to the top of your page’s source code:

using Memba.WebControls;

Add the following code to the Page\_Load event of your page:

protected void Page\_Load(object sender, EventArgs e)

{

if (!Page.IsPostBack)

{

ImageList.ImageListItemCollection.Add(

new ImageListItem(

Guid.NewGuid().ToString(),

Memba.Common.Presentation.Icons.GetIcon32("upload.gif"),

"Sample 1"

));

ImageList.ImageListItemCollection.Add(

new ImageListItem(

Guid.NewGuid().ToString(),

Memba.Common.Presentation.Icons.GetIcon32("upload.gif"),

"Sample 2"

));

ImageList.ImageListItemCollection.Add(

new ImageListItem(

Guid.NewGuid().ToString(),

Memba.Common.Presentation.Icons.GetIcon32("upload.gif"),

"Sample 3"

));

}

}

Memba.Common.Presentation.Icons.GetIcon32 is a utility function which returns the path to an icon, that is ~/App\_Images/32x32/upload.gif.

When you display your page, the ImageList is now loaded with three items.

Add an ASP.NET button to your page and name it PostButton. Wire the click event to an event handler named PostButton\_Click:

<asp:Button ID="PostButton" runat="server" Text="Postback" OnClick="PostButton\_Click" />

Add the following code to the page source code:

protected void PostButton\_Click(object sender, EventArgs e)

{

foreach (ImageListItem item in ImageList.ImageListItemCollection)

{

Response.Write(item.Text + "<br />");

}

}

Run the page and observe that on post back, the list keeps its state and the names of items are displayed on the top of the page.

## ImageList client-side code

The main benefit of this ImageList control is to provide a client-side API.

Add an HTML input button to your page:

<input id="AddItem" type="button" value="add item" />

Add the following Javascript code into the script section of your page:

<script type="text/javascript">

<!--

var g\_AddItemButton;

var g\_ImageList;

var g\_Count=0;

//This function is called by the Ajax framework when the DOM is loaded

function pageLoad()

{

//Get a reference to the AddItem button

g\_AddItemButton = $get("AddItem");

//Add an event handler for the click event of the AddItem button

$addHandler(g\_AddItemButton, "click", onAddItem);

//Get a reference to the ImageList control

g\_ImageList = $find("<%= ImageList.ClientID %>")

}

//Event handler for the click event of the AddItem button

function onAddItem()

{

g\_Count++;

//Instantiate a new item

var item = new Memba.WebControls.ImageListItem(

Memba.Utils.newGuid(),

'<%= this.ResolveClientUrl("~/App\_Images/32x32/upload.gif") %>',

"Sample+ " + g\_Count

);

//Add the item to the list

g\_ImageList.add\_item(item);

}

//-->

</script>

You now realize that the reference to Memba.Utils in the ScriptManager is used to generate a Guid client-side.

Run the page and click the AddItem button. A new item is added to the list.

A sample implementation called ImageList.aspx is available in the Debug folder of the web site project of the solution.

# MultiUpload2

Memba.WebControls releases two versions of the MultiUpload control:

1. MultiUpload uses the click event of an anchor (<A> tag) to trigger programmatically a click event on an Input type=file html control which is hidden. This is supported by Internet Explorer but Firefox (2.0 and below) do not support triggering a click event programmatically on file upload html controls, probably for security reasons.
2. MultiUpload2 uses a trick described at <http://www.quirksmode.org/dom/inputfile.html> to display a link which when clicked opens a file selection dialog. The trick is to position the file upload control on top of the link and to set its opacity to 0 so that it is not seen. MultiUpload2 extends the concept by adding a new input each time the current one is filled. The collection of inputs maintained by the MultiUpload2 control constitutes the list of files to upload.

Start working with a new page as described in the “ASP.NET Ajax Extensions” section of this document. To add a MultiUpload2 component to your page, simply drag and drop the control or add the following code:

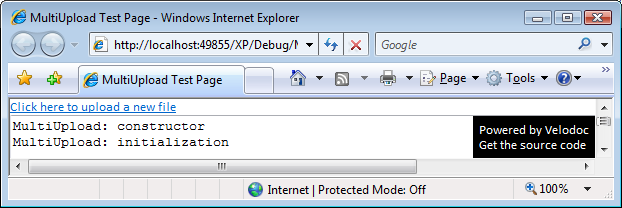
<form id="form1" runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server"></asp:ScriptManager>

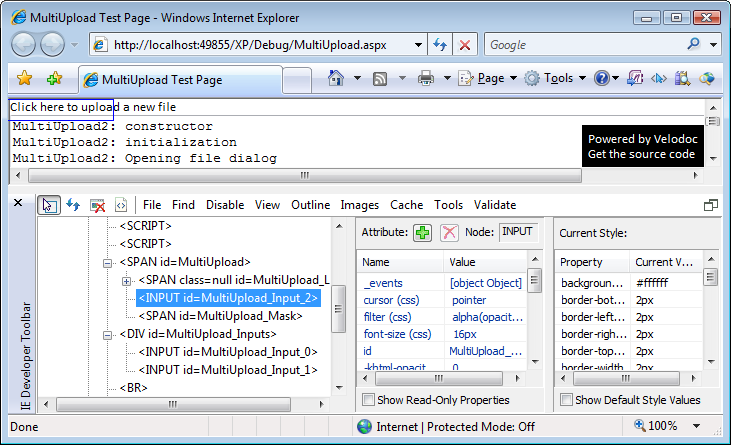
<mbui:MultiUpload2 ID="MultiUpload" runat="server" Text="Click here to upload a new file"></mbui:MultiUpload2>

</form>

Your MultiUpload2 control will look like follows:



There are several ways to process file uploads server-side. All you need to know is that the MultiUpload2 component is actually a series of hidden <INPUT TYPE=FILE> html controls. We will show you later how to implement file uploads with our Upload Http Module.



Archived INPUTs (loaded)

Current INPUT (empty)

For now, we will focus on client side code. Add the following Javascript code into the script section of your page:

<script type="text/javascript">

<!--

//This function is called by the Ajax framework when the DOM is loaded

function pageLoad()

{

//Get a reference to the MultiUpload control

var \_c = $find("<%= MultiUpload.ClientID %>");

//Add en event handler for the browse event

\_c.add\_browse(onBrowse);

}

//Event handler for the browse (click) event of the MultiUpload control

function onBrowse(sender, args)

{

//Display the file path in an alert box

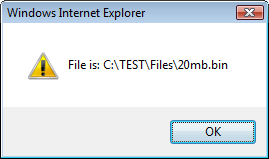
alert("File is: " + args.get\_value());

}

//-->

</script>

Run the page and click the MultiUpload link. An alert box displays the file path.



A sample implementation called MultiUpload.aspx is available in the Debug folder of the web site project of the solution.

# Wiring the MultiUpload2 with the ImageList

Start working with a new page as described in the “ASP.NET Ajax Extensions” section of this document. To add a MultiUpload2 and an ImageList component to your page, simply drag and drop the controls or add the following code:

<form id="form1" runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server">

<Scripts>

<asp:ScriptReference Path="~/ScriptLibrary/Memba.Utils.js" />

</Scripts>

</asp:ScriptManager>

<mbui:MultiUpload2 ID="MultiUpload" runat="server"

SkinID="sknMultiUpload2"

Text="Choose file..."></mbui:MultiUpload2>

<mbui:ImageList ID="ImageList" runat="server"

SkinID="sknFileList"

RemoveTooltip="Remove from selection"

Height="92px"

Width="420px"></mbui:ImageList>

</form>

We will need a reference to Memba.Utils.js to create new guids in Javascript.

Wiring the MultiUpload2 control with the ImageList control consists in:

1. Adding an ImageList item each time a new file is added to the MultiUpload control; This is done by implementing a handler for the browse event;
2. Removing a file (INPUT html control) from teh MultiUpload control each time it is removed from the ImageList; This is done by implementing a handler for the remove event.

Add the following Javascript code into the script section of your page:

<script type="text/javascript">

<!--

var g\_MultiUpload;

var g\_ImageList;

var g\_ClearButton;

//This function is called by the Ajax framework when the DOM is loaded

function pageLoad()

{

//Get a reference to the MultiUpload control

g\_MultiUpload = $find("<%= MultiUpload.ClientID %>");

//Add en event handler for the browse event

g\_MultiUpload.add\_browse(onBrowse);

//Get a reference to the ImageList control

g\_ImageList = $find("<%= ImageList.ClientID %>");

//Add en event handler for the browse event

g\_ImageList.add\_remove(onRemove);

}

//This function is called by the Ajax framework when unloading the page

function pageUnload()

{

//This is good practice to clear your event handlers

g\_MultiUpload.remove\_browse(onBrowse);

g\_ImageList.remove\_remove(onRemove);

}

//Event handler for the browse (click) event of the MultiUpload control

function onBrowse(sender, args)

{

//Search for the item in the Imagelist

if (g\_ImageList.find\_item(args.get\_value()).length > 0)

{

alert("file already in list");

//The item already exists, we can remove the duplicate INPUT

//in the MultiUpload control

g\_MultiUpload.removeInput(args.get\_id());

}

else

{

//Since teh item is not found in the ImageList, create a new item

var item = new Memba.WebControls.ImageListItem(

Memba.Utils.newGuid(),

'<%= this.ResolveClientUrl(

Memba.Common.Presentation.Icons.GetIcon32(

"upload.gif")) %>',

args.get\_value(),

args.get\_value(),

args.get\_id()

);

//Add teh new item to the ImageList

g\_ImageList.add\_item(item);

}

//We can do some tracing which will display

//in the TraceConsole textarea

Sys.Debug.trace(g\_ImageList.get\_count() + " files in image list, and "

+ g\_MultiUpload.get\_count() + " files in MultiUpload control");

}

//Event handler for the remove event of the ImageList control

function onRemove(sender, args)

{

//Upon clicking the remove icon in the ImageList, remove the

//corresponding INPUT in the MultiUpload control

g\_MultiUpload.removeInput(args.get\_tag());

//We can do some tracing which will display

//in the TraceConsole textarea

Sys.Debug.trace(g\_ImageList.get\_count() + " files in image list, and "

+ g\_MultiUpload.get\_count() + " files in MultiUpload control");

}

//-->

</script>

Run the page. You can now add items to the ImageList control by clicking the MultiUpload2 control. Each time you remove an item from the list, the corresponding file is removed from the MultiUpload2 control.

A sample implementation called MultiUpload2.aspx is available in the Debug folder of the web site project of the solution.

# ProgressReport

Start working with a new page as described in the “ASP.NET Ajax Extensions” section of this document. To add a ProgressReport to your page, simply drag and drop the control or add the following code:

<form id="form1" runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server" ></asp:ScriptManager>

<mbui:ProgressReport ID="ProgressReport" runat="server"

DefaultText="Waiting..."

Width="400px"

Interval="1000"

BarCssClass="cssProgressBar"

FillerCssClass="cssProgressFiller"

HandlerUrl="~/Debug/progressHandler.ashx"

TextFormat="Value 1: {0}<br/>Value 2: {1}" ></mbui:ProgressReport>

<input type="button" onclick="go();" value="Go" />

</form>

Also add an html button to start the progress bar.

Add the following styles just above the </head>closing tag of your page:

<style type="text/css">

<!--

div.cssProgressBar

{

border:solid 1px DarkGreen !important;

padding:1px;

background-color:White;

}

div.cssProgressFiller

{

background-color:DarkGreen !important;

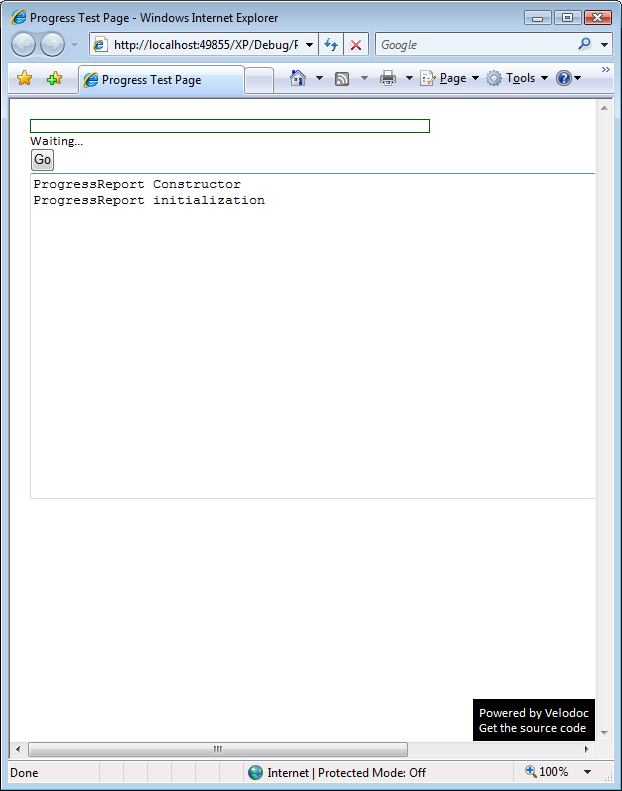
height:10px;

}

//-->

</style>

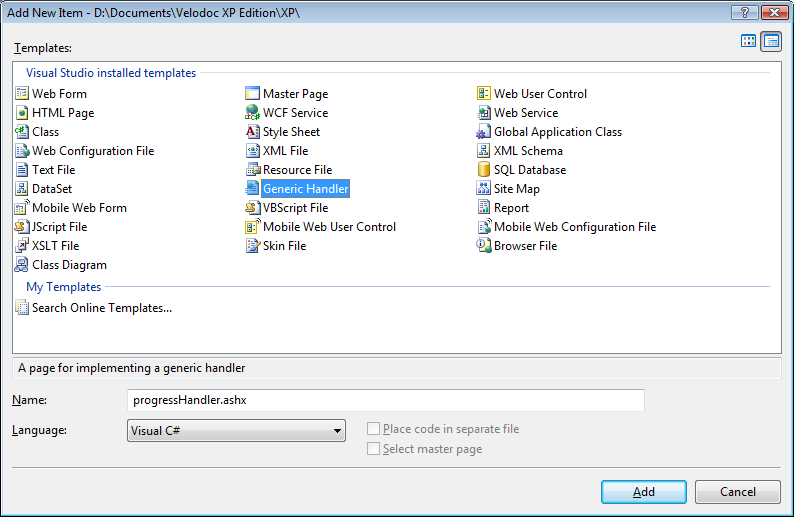
Your ProgressReport control will look like follows:



ProgressReport has two properties which work in conjunction:

1. HandlerUrl points to an http handler which should return a JSON string in a specific format; you could use an ASP.NET page to return this formatted string, but a handler is leaner in this case because ASP.NET pages have a complex lifecycle.
2. TextFormat displays a textual description of the progress using the data returned by the http handler.

Start by creating a new generic handler:



Add the following two statements at the top of your handler:

using Memba.WebControls; //ProgressData4JS

using System.Web.Script.Serialization; //JavaScriptSerializer

Implement your handler as follows:

public class progressHandler : IHttpHandler

{

private static int \_Counter;

public void ProcessRequest (HttpContext context)

{

//First, make sure the content is not cached because we need

//"real-time" progress report

//Using headers or cache attributes to set the cache policy

//should be equivalent

context.Response.AppendHeader("Cache-Control",

"no-cache, no-store, must-revalidate");

context.Response.AppendHeader("Expires", "-1");

context.Response.AppendHeader("Pragma", "no-cache");

//but the following does not work in IIS 7 (Vista 64-bit OS),

//which requires the code above

context.Response.Cache.SetCacheability(HttpCacheability.NoCache);

context.Response.Cache.SetRevalidation(

HttpCacheRevalidation.AllCaches);

context.Response.Cache.SetNoStore();

context.Response.Cache.SetNoServerCaching();

context.Response.Cache.SetExpires(DateTime.Now);

//Create a progress data object which will be used for JSON

//serialization

ProgressData4JS objProgressData4JS = new ProgressData4JS();

//Fill the progress data until progress reaches 100%

if (\_Counter < 100)

{

objProgressData4JS.IsComplete = false;

objProgressData4JS.FillRatio = String.Format("{0}%", \_Counter);

objProgressData4JS.TextValues = new object[] {

"PENDING",

\_Counter };

objProgressData4JS.ErrorMessage = null;

\_Counter+=10;

}

else //Once progress reaches 100% we are done

{

objProgressData4JS.IsComplete = true;

objProgressData4JS.FillRatio = "100%";

objProgressData4JS.TextValues = new object[] {

"COMPLETE",

\_Counter };

objProgressData4JS.ErrorMessage = null;

\_Counter = 0;

}

//Serialize progress data

JavaScriptSerializer objJavaScriptSerializer =  
 new JavaScriptSerializer();

string sResponse =

objJavaScriptSerializer.Serialize(objProgressData4JS);

context.Response.ContentType = "text/json";

//Some use "application/json"

//Write the response

context.Response.Write(sResponse);

context.ApplicationInstance.CompleteRequest();

//See: http://support.microsoft.com/kb/312629

//context.Response.End();

}

public bool IsReusable {

get

{

return true;

}

}

}

Whatever the task you execute on your server and you want to report progress on, always use a ProgressData4JS object which you write to the response stream using JSON serialization.

The progress bar of the ProgressReport control uses the FillRatio property of the PorgressData4JS object to display the status of the progression.

ProgressData4JS also has a property called TextValues which is an array of objects used to display the text under the progress bar in conjunction with the TextFormat property of the ProgressReport control.

In our example, when the progress has reached 100%, TextValues contains “COMPLETE” and “100”. The TextFormat property of our ProgressReport is “Value 1: {0}<br />Value2: {1}”. The result displayed will be:

Value 1: COMPLETE  
Value 2: 100

Now, we need to write the Javascript code to launch the progress.

<script type="text/javascript">

<!--

//Click event handler for the Go button

function go()

{

//Get a reference to the progress control

var \_c = $find("<%= ProgressReport.ClientID %>");

//Add an event handler for the complete event

\_c.add\_complete(onComplete);

//Start progress report

\_c.start();

}

//Complete event handler for the progress report

function onComplete(e)

{

alert("complete");

}

//-->

</script>

A sample implementation called Progress.aspx is available in the Debug folder of the web site project of the solution. The corresponding http handler is progressHandler.ashx.

# Upload module

We recommend you look at the “Architecture and Design” document to understand how the upload module works.

## Configuration

You first need to add a reference to the Memba.FileUpload.XP assembly to your web site project.

Then you need to configure the upload module in web.config:

For IIS 5 and IIS 6, add the following to the <configuration><system.web><httpModules> section of web.config:

<add name="UploadHttpModule" type="Memba.FileUpload.UploadHttpModule, Memba.FileUpload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" />

For IIS 7, add the following line to the <configuration><system.webServer><modules> section of web.config:

<add name="UploadHttpModule" preCondition="integratedMode" type="Memba.FileUpload.UploadHttpModule, Memba.FileUpload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a"/>

Note that UploadHttpModule is compatible with IIS7 integrated pipeline mode.

The UploadHttpModule is instrumented, so you need the following section in <configuration><system.web>:

<healthMonitoring enabled="true">

<eventMappings>

<add name="Upload Error Web Events" type="Memba.FileUpload.UploadRequestErrorEvent, Memba.FileUpload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" />

<add name="Upload Success Web Events" type="Memba.FileUpload.UploadRequestSuccessEvent, Memba.FileUpload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" />

</eventMappings>

<rules>

<add name="Upload Error Rules" eventName="Upload Error Web Events" provider="EventLogProvider" profile="Default" />

<add name="Upload Success Rules" eventName="Upload Success Web Events" provider="EventLogProvider" profile="Default" />

</rules>

</healthMonitoring>

You also need to configure the provider-based file storage. To do so, first add a new section group to the <configuration><configSections> section of your web.config:

<sectionGroup name="system.web">

<section name="fileStorage" type="Memba.FileUpload.Providers.FileStorageConfigurationSection, Memba.FileUpload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" />

</sectionGroup>

Then add the following section to <configuration><system.web> to configure the FSFileStorageProvider (FS = File System) storage provider:

<fileStorage defaultProvider="FSFileStorageProvider">

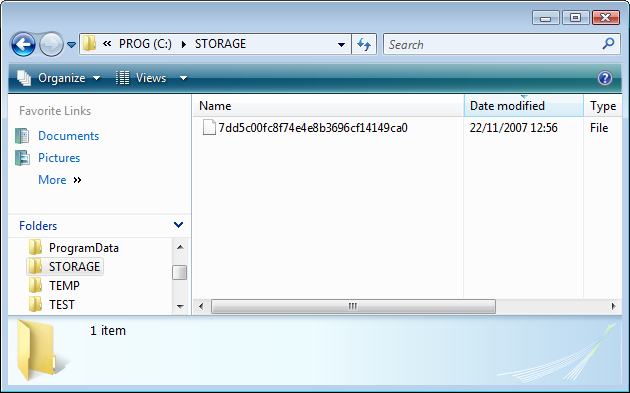
<providers>

<add name="FSFileStorageProvider" type="Memba.FileUpload.Providers.FSFileStorageProvider, Memba.FileUpload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" connectionString="C:\STORAGE" />

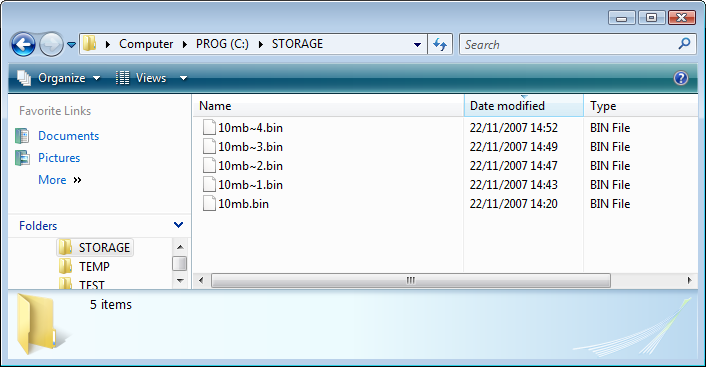
</providers>

</fileStorage>

The FSFileStorageProvider stores files under generated names which have no extension:



Alternatively, you can configure the SimpleFileStorageProvider which stores files with clear file names as represented below. This is not the recommended approach.



The important part is the connectionString which in both cases is the path to the directory where files are stored. You need to set the following permissions to this directory:

1. On Windows XP, MACHINENAME\ASPNET needs modify permissions;
2. On Windows 2003, MACHINENAME\IIS\_WPG (or NETWORK SERVICE) needs modify permissions;
3. On Windows Vista, BUILTIN\IIS\_IUSRS needs modify permissions;

Note that if your application pool runs under a specific identity, this identity needs these permissions.

Finally, you need to configure ASP.NET to allow for large posts. By default ASP.NET posts are limited to 4MB (maxRequestLength = 4096). You do this by adding an <httpRuntime> section either to <configuration><system.web> for your entire site or within a <location> section if you want to limit large posts to specific pages.

<httpRuntime maxRequestLength="102400" executionTimeout="3600" />

In IIS 7, you may get the following error when you upload files larger than 30 MB: HTTP Error 404.13 - The request filtering module is configured to deny a request that exceeds the request content length. To resolve this, you need to add the following section to <configuration><webServer>:

<security>

<requestFiltering>

<requestLimits maxAllowedContentLength="102400000"/>

</requestFiltering>

</security>

If you have error that requestFiltering cannot be overriden you must edit your mashine.config file and change:

* <section name="requestFiltering" overrideModeDefault="Deny" />  
  to
* <section name="requestFiltering" overrideModeDefault="Allow" />

## Coding

Let’s now write some code. Start working with a new page as described in the “ASP.NET Ajax Extensions” section of this document. Add a ProgressReport, an ASP.NET FileUpload (not a MultiUplaod) and an ASP.NET button component to your page, by simply dragging and dropping these controls or by adding the following code:

<form id="form1" runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server" ></asp:ScriptManager>

<mbui:ProgressReport ID="ProgressReport" runat="server"

DefaultText="Waiting..."

Width="400px"

Interval="3000"

BarCssClass="cssProgressBar"

FillerCssClass="cssProgressFiller"

TextFormat="Value 1: {0}<br/>Value 2: {1}" ></mbui:ProgressReport>

<asp:FileUpload ID="FileUpload" runat="server" />

<asp:Button ID="SubmitButton" runat="server"

OnClientClick="go();"

Text="Submit" />

</form>

Please refer to the ProgressReport section of this document to add styles in the <head> section of your page.

Note that we have not set the HandlerUrl property of the ProgressReport. We shall do this in server-side code by implementing the Page\_Load event handler of the page:

protected void Page\_Load(object sender, EventArgs e)

{

//Create a new identifier for the upload data

Guid gUploadId = Guid.NewGuid();

//Set the handler with the proper identifier

ProgressReport.HandlerUrl = "uploadHandler.ashx?muid=" +

gUploadId.ToString();

//Postback to the same url, but add the identifier to the query string

SubmitButton.PostBackUrl = Request.Url.AbsolutePath + "?muid=" +

gUploadId.ToString();

}

The important thing you should notice is that we need a unique upload identifier called muid (Memba Upload Identifier):

1. The upload http module uses this muid as a key to retrieve the UploadData object which is used to maintain the various data relative to the upload (list of files, size, date started, percentage completed, ...) as it progresses;
2. The upload handler used by the ProgressReport uses the same muid to request the relevant UploadData object from the upload http module (more exactly from the UploadMonitor of the upload http module).

We can now write the code for the uploadHandler. Create a Generic handler named uploadHandler.ashx as we did in the ProgressReport section of this document. Add the following statements to the top of the page:

using Memba.FileUpload; //UploadMonitor

using Memba.Files.Business; //BODIsplay

using Memba.WebControls; //ProgressData4JS

using System.Web.Script.Serialization; //JavaScriptSerializer

You may have to add a reference to the Memba.Files.XP assembly in your web site project.

Now implement the handler’s ProcessRequest method as follows:

public void ProcessRequest (HttpContext context)

{

//First, make sure the content is not cached

because we need "real-time" progress report

//Using headers or cache attributes to set the cache policy

//should be equivalent

context.Response.AppendHeader("Cache-Control",

"no-cache, no-store, must-revalidate");

context.Response.AppendHeader("Expires", "-1");

context.Response.AppendHeader("Pragma", "no-cache");

//but the following does not work in IIS 7 (Vista 64-bit OS),

//which requires the code above

context.Response.Cache.SetCacheability(HttpCacheability.NoCache);

context.Response.Cache.SetRevalidation(

HttpCacheRevalidation.AllCaches);

context.Response.Cache.SetNoStore();

context.Response.Cache.SetNoServerCaching();

context.Response.Cache.SetExpires(DateTime.Now);

//Create a progress data object  
 //which will be used for JSON serialization

ProgressData4JS objProgressData4JS = new ProgressData4JS();

try

{

//Use the identifier in query string to access UploadData

UploadData objUploadData = UploadMonitor.GetUploadData(

context.Request.QueryString[UploadMonitor.UploadIdParam]);

if (objUploadData != null)

{

//There is some relevant UploadData

objProgressData4JS.IsComplete =

(objUploadData.ProgressStatus ==

UploadProgressStatus.Completed);

objProgressData4JS.FillRatio = String.Format(

"{0:N0}%", 100 \* objUploadData.ProgressRatio);

objProgressData4JS.TextValues = new object[] {

objUploadData.ProgressStatus.ToString(),

BODisplay.BandwidthFormat(

objUploadData.BytesPerSecond, ByteFormat.Adapt)

};

objProgressData4JS.ErrorMessage = null;

}

else

{

//UploadData cannot be found; Maybe it is not yet available

objProgressData4JS.IsComplete = false;

objProgressData4JS.FillRatio = "0%";

objProgressData4JS.TextValues = new object[] {

"Unknown",

"0MB/s"

};

objProgressData4JS.ErrorMessage = null;

}

}

catch (Exception Ex)

{

//Oops! Houston, there is a problem

objProgressData4JS.IsComplete = true;

objProgressData4JS.FillRatio = null;

objProgressData4JS.TextValues = null;

objProgressData4JS.ErrorMessage = Ex.Message;

}

//Serialize progress data

JavaScriptSerializer objJavaScriptSerializer =

new JavaScriptSerializer();

string sResponse =

objJavaScriptSerializer.Serialize(objProgressData4JS);

//Write the response

context.Response.ContentType = "text/json";

//Some use "application/json"

context.Response.Write(sResponse);

context.ApplicationInstance.CompleteRequest();

//See: http://support.microsoft.com/kb/312629

//context.Response.End();

}

The last thing to do is to start the ProgressReport when the user clicks the Submit button, which is done by adding the following JavaScript code to the page:

<script type="text/javascript">

//Click event handler for the Go button

function go()

{

//Get a reference to the ProgressReport control

var \_c = $find("<%= ProgressReport.ClientID %>");

if (\_c)

{

//Start progress report

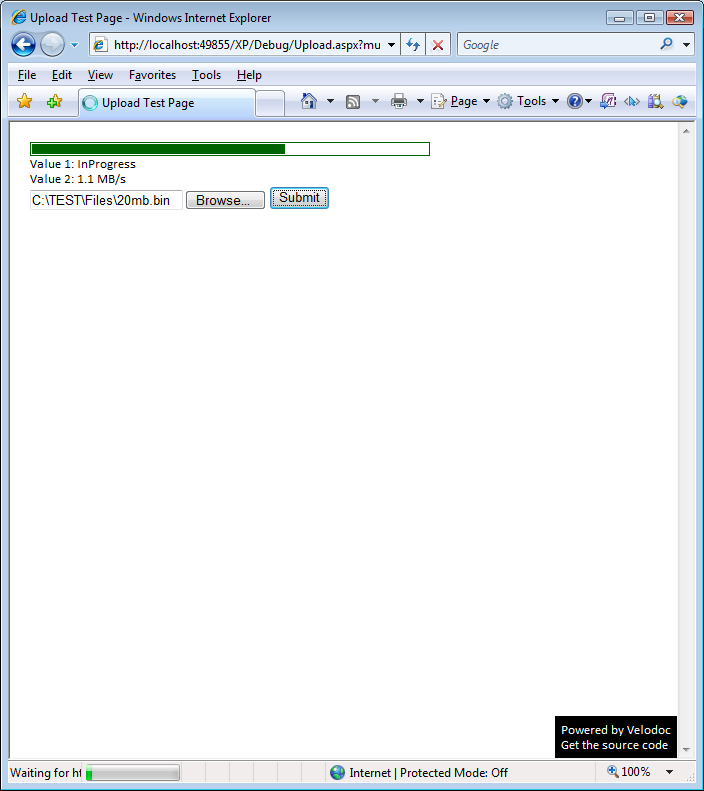
\_c.start();

}

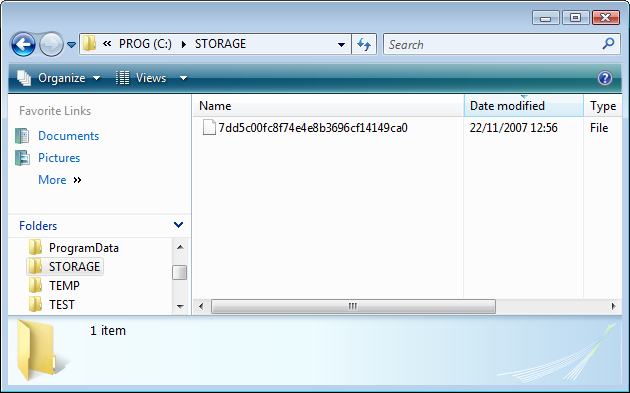
}

</script>

After browsing for a file and clicking submit, teh page should display as follows:



At the end of the progress, you should find a new file in your storage folder:



A sample implementation called Upload.aspx is available in the Debug folder of the web site project of the solution. The corresponding http handler is uploadHandler.ashx.

# Download handler

We recommend you look at the “Architecture and Design” document to understand how the download handler works. The download handler is essentially a C# implementation of the solution described in MSDN Magazine of September 2006 at <http://msdn.microsoft.com/msdnmag/issues/06/09/WebDownloads/default.aspx>.

We have made a few changes to instrument the handler, better report errors and handle Unicode file names. The great benefit of the download handler is that it supports ETag and Accept-Ranges http headers which means downloads are resumable.

## Configuration

You first need to add a reference to the Memba.FileDownload.XP assembly to your web site project.

Then you need to configure the download handler in web.config:

For IIS 5 and IIS 6, add the following to the <configuration><system.web><httpHandlers> section of web.config:

<add verb="GET,HEAD" path="\*.dat" validate="false" type="Memba.FileDownload.DownloadHandler, Memba.FileDownload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" />

For IIS 7, add the following line to the <configuration><system.webServer><handlers> section of web.config:

<add name="DownloadHttpHandler" preCondition="integratedMode" verb="GET,HEAD" path="\*.dat" type="Memba.FileDownload.DownloadHandler, Memba.FileDownload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" resourceType="Unspecified"/>

Add the following section to <configuration><system.web> or merge with the corresponding section which has been added for the upload http module:

<healthMonitoring enabled="true">

<eventMappings>

<add name="Download Error Web Events" type="Memba.FileDownload.DownloadRequestErrorEvent, Memba.FileDownload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" />

<add name="Download Success Web Events" type="Memba.FileDownload.DownloadRequestSuccessEvent, Memba.FileDownload.XP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a4ae091aa8097a5a" />

</eventMappings>

<rules>

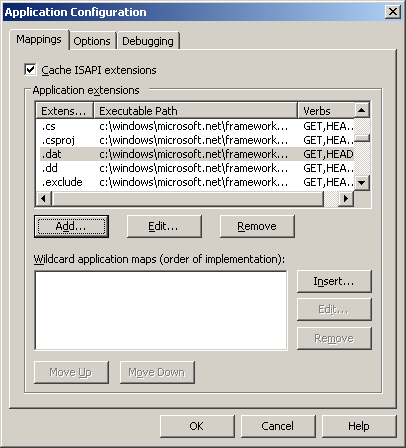
<add name="Download Error Rules" eventName="Download Error Web Events" provider="EventLogProvider" profile="Default" />

<add name="Download Success Rules" eventName="Download Success Web Events" provider="EventLogProvider" profile="Default" />

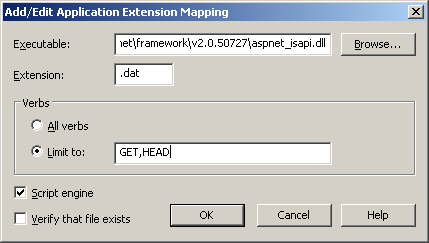
</rules>

</healthMonitoring>

We are using the .dat extension for file downloads. We need to create a script map to ensure our download handler is triggered. This is done by going to the application configuration of your web site in the IIS management console:



Then you should map the .dat extension to the aspnet\_isapi.dll. Make sure that Verify that file exists is unchecked because obviously no file with the .dat extension exists in our storage folder.



## Coding

When you use FSFileStorageProvider, files are stored under generated names without extension, for example 362d9bac67294c1a8c57acd02fd5d68f.

Obviously, users cannot download such files, because they would have to guess the original name and extension. The system would also have to guess the content type. So we need a way to map technical internal names with original name and content type. We can use a database, like we do in Velodoc Enterprise Edition. We can also use XML files which we do in Velodoc XP Edition.

The download handler uses .def files. Here is the content of a definition file for the file above:

<?xml version="1.0" encoding="utf-8"?>  
<File xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns="http://schemas.memba.org/2006/attachments">

<Guid>fa8a136d-3fd5-49f4-ba3e-96230a46dc81</Guid>

<FileName>Important Document.pdf</FileName>

<ContentType>application/pdf</ContentType>

<Key>362d9bac67294c1a8c57acd02fd5d68f</Key>

<Size>274308</Size>

<HashValue>474ab5ca8ee6909be1eef19508cde94f9240d924</HashValue>

<CreatedOn>2007-11-22T15:43:03.2356043Z</CreatedOn>

</File>

In the current stage of developments, these files are not automatically generated by the FSFileStorageProvider. You have to generate them in your code which we will do now.

Start working with a new ASP.NET page. This page does not need to support ASP.NET Ajax Extensions. Add an ASP.NET FileUpload (not a MultiUplaod), an ASP.NET button and an ASP.NET hyperlink component to your page, by simply dragging and dropping these controls or by adding the following code:

<form id="form1" runat="server">

<asp:FileUpload ID="FileUpload" runat="server" />

<asp:Button ID="SubmitButton" runat="server"

Text="Submit"

OnClick="SubmitButton\_Click" />

<asp:HyperLink ID="DownloadLink" runat="server"></asp:HyperLink>

</form>

Add the following statements at the top of your page code:

using Memba.FileUpload; //UploadMonitor, UploadData

using Memba.FileUpload.Providers; //FileStorage

using Memba.Files.Business; //FileBroker, File, BODisplay

You may have to reference the Memba.FileUpload.XP and Memba.Files.XP assemblies.

Add the following code to the Page\_load event handler of your page:

protected void Page\_Load(object sender, EventArgs e)

{

Guid gUploadId = Guid.NewGuid();

//Postback to the same url, but add the identifier to the query string

SubmitButton.PostBackUrl = Request.Url.AbsolutePath

+ "?muid=" + gUploadId.ToString();

}

See the Upload Module section of this document for an explanation of the muid identifier.

Add the following code to the SubmitButton\_Click event handler of your page:

protected void SubmitButton\_Click(object sender, EventArgs e)

{

try

{

//Confirm that we have a muid to access upload data

System.Diagnostics.Debug.Assert(

!String.IsNullOrEmpty(

Request.QueryString[UploadMonitor.UploadIdParam]));

UploadData objUploadData =

UploadMonitor.GetUploadData(

Request.QueryString[UploadMonitor.UploadIdParam]);

if (objUploadData == null)

throw new ApplicationException("Oops! No upload data");

//Check any exception

Exception objUploadException = objUploadData.Exception;

if (objUploadException != null)

throw new ApplicationException(

objUploadException.Message, objUploadException);

//Ensure that upload is complete

if (objUploadData.ProgressStatus != UploadProgressStatus.Completed)

throw new ApplicationException("Oops! Upload not complete");

//Ensure we have at least one uploaded file

//(we should have at least one file input control)

if ((objUploadData.UploadFiles == null) ||

(objUploadData.UploadFiles.Count < 1))

throw new ApplicationException("Oops! No uploaded file");

//Keep it simple: we now there is only

//one file upload control on the page

UploadFile objUploadFile = objUploadData.UploadFiles[0];

if ((objUploadFile == null) || (!objUploadFile.IsComplete))

throw new ApplicationException("Oops! Uploaded file is not

complete");

//Create file object

File f = new File(

objUploadFile.OriginalFileName,

objUploadFile.ContentType,

objUploadFile.ProviderFileKey.ToString(),

objUploadFile.ContentLength,

objUploadFile.HashValue);

//Get path to file storage

string sPath = FileStorage.Provider.ConnectionString;

System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(sPath);

//Add definition to file storage

FileBroker.Insert(f, di);

//Create download link

DownloadLink.Text = "Download: "

+ f.FileName

+ " (" + BODisplay.SizeFormat(f.Size, ByteFormat.Adapt) + ")";

DownloadLink.NavigateUrl =

System.IO.Path.Combine(

this.Request.ApplicationPath,

f.Guid.ToString("N") + ".dat"

);

//Release upload data

UploadMonitor.Release(objUploadData.UploadId);

}

catch (Exception Ex)

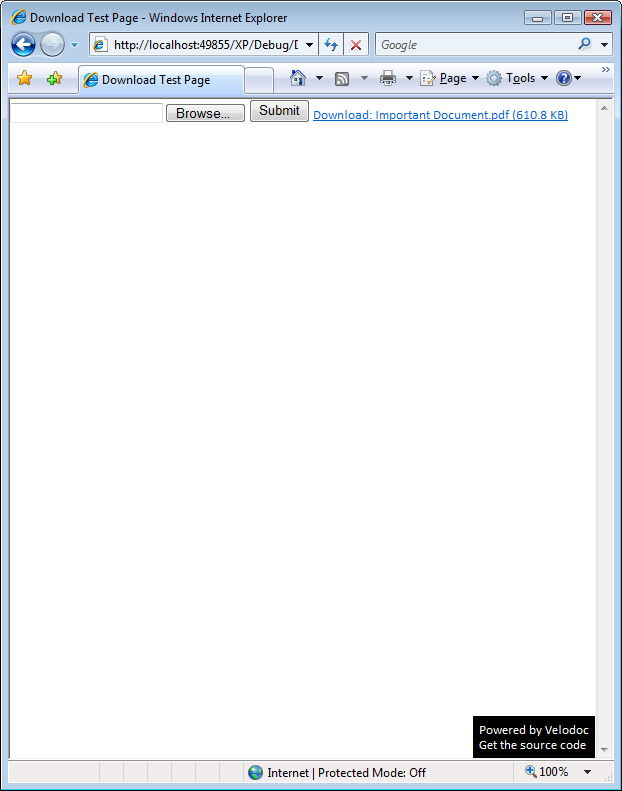
{

Response.Write(Ex.Message);

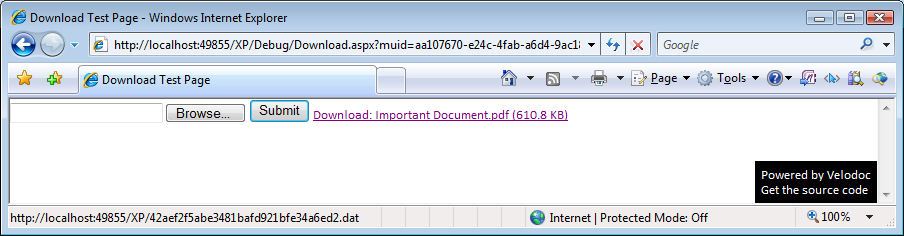
}

}

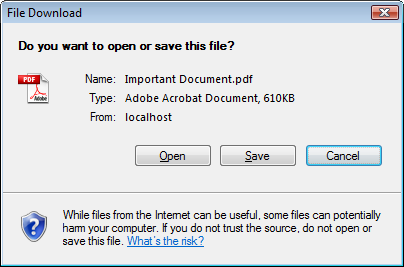
Run the page and upload a document. Your page should look like follows:



Click the download link to download your document and observe that as though the link is in the form:



... you get to download the file under its original name:



A sample implementation called Download.aspx is available in the Debug folder of the web site project of the solution.

# Putting it all together

At this stage, you have a very good knowledge how all components interact together to fulfil the requirements of large file uploads and downloads. We have assembled all the pieces together in the Velodoc XP Edition solution, and more particularly in the quickmessage.ascx user control which you can reuse on your web site. We highly recommend you review the code of quickmessage.ascx to learn how everything is put together.

# New in version 1.1: WCF streaming service

Version 1.1 now features a WCF streaming web service to upload and download files using either:

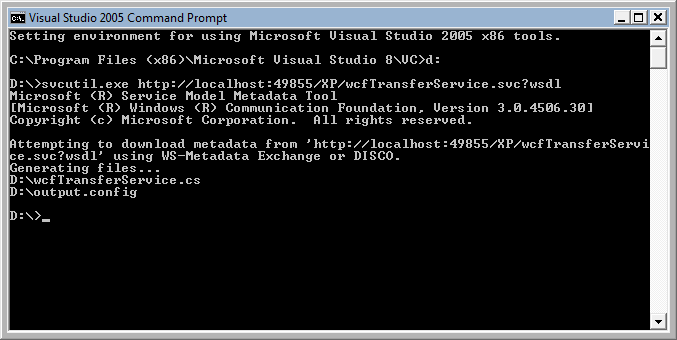
* The Memba Velodoc Outlook Add-In;
* The client API which comes with the add-in;
* The .NET proxy which you can generate using svcutil.exe.

The use of the Memba Velodoc Outlook Add-In and Client API is described in the relevant documentation which you can download with the source code from <http://www.velodoc.com/download>.

We will focus here on how to upload files using the WCF streaming service in a client application.

You generate a client proxy by executing the following command line as explained in [How to: Create a Windows Communication Foundation Client](http://msdn.microsoft.com/en-us/library/ms733133.aspx), where <port> and <virtualDir> are determined by your server configuration. Opening a Visual Studio command prompt sets the environment with a path to svcutil.exe.

svcutil.exe http://localhost:<port>/<virtualDir>/wcfTransferService.svc?wsdl



Two files are generated which you can use in your client application.

It is recommended to add a 3rd file as explained in [WCF Clients and the "Broken" IDisposable Implementation](http://geekswithblogs.net/DavidBarrett/archive/2007/11/22/117058.aspx). Name this file wcfTransferServiceFix.cs and copy the following text content:

using System;

using System.ServiceModel;

public partial class TransferServiceClient : IDisposable

{

public void Dispose()

{

if (this.State == CommunicationState.Faulted)

{

this.Abort();

}

else

{

this.Close();

}

}

}

This will allow you to call your web service in a using statement to ensure all resources are properly disposed:

using (TransferServiceClient objClientProxy = new TransferServiceClient())

{

...

}

If you will be using output.config to configure your web service, you will need to change your binding information to increase maxReceivedMessageSize and more importantly change transferMode to Streamed:

<binding name="BasicHttpBinding\_ITransferService" closeTimeout="00:01:00"

openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"

allowCookies="false" bypassProxyOnLocal="false" hostNameComparisonMode="StrongWildcard"

maxBufferSize="65536" maxBufferPoolSize="524288" maxReceivedMessageSize="~~65536~~1073741824"

messageEncoding="Mtom" textEncoding="utf-8" transferMode="~~Buffered~~Streamed"

useDefaultWebProxy="true">

Note that the streaming service works in IIS but does not work in the Visual Studio Web Server (cassini) environment as explained in [WCF Streaming: upload files over http](http://kjellsj.blogspot.com/2007/02/wcf-streaming-upload-files-over-http.html). This article also explains other caveats you may face when configuring your web service in IIS.

## Uploading a file

Uploading a file from your client application to Velodoc XP Edition requires that you:

* Reference System.ServiceModel in your .NET 3.0 project;
* Add TransferServiceClient.cs and TransferServiceClientFix.cs to this project;
* Merge output.config with your app.config or configure your WCF bindings dynamically;
* Write code similar to:

//Open local input stream

using (FileStream objFileStream = File.OpenRead("C:\\test.zip"))

{

//Start service client with configuration in config file

using (TransferServiceClient objClientProxy = new TransferServiceClient())

{

objClientProxy.Upload(

"application/zip",

"me@acme.com", //an email address listed in the app settings

Guid.NewGuid(), //a unique id which is used in the download uri

"test.zip",

"FC079B8E18014B5FC161FF4CF4D764B65D5A368D", //a hash code

objFileStream.Length,

"1234", //the security code recorded in the app settings

objFileStream);

}

}

## Downloading a file

Downloading a file from Velodoc XP Edition to your client application requires that you:

* Reference System.ServiceModel in your .NET 3.0 project;
* Add TransferServiceClient and TransferServiceClientFix.cs to this project;
* Merge output.config with your app.config or configure your WCF bindings dynamically;
* Write code similar to:

TransferServiceClient objClientProxy = null;

Stream objRemoteStream = null;

Stream objLocalStream = null;

try

{

string sEmail;

Guid gFileGuid;

string sFileName;

string sHashCode;

long lLength;

string sSecurityCode;

//Start service client with configuration in config file

objClientProxy = new TransferServiceClient();

//Launch download and initialize remote stream

objClientProxy.Download(

new Uri("http://localhost/XP/42aef2f5abe3481bafd921bfe34a6ed2.dat"),

out sEmail, //Returns null

out gFileGuid,

out sFileName,

out sHashCode,

out lLength,

out sSecurityCode, //Returns null

out objRemoteStream);

//Remove existing local file

FileInfo objLocalFileInfo = new FileInfo(Path.Combine(("C:\\", sFileName));

if (objLocalFileInfo.Exists)

{

objLocalFileInfo.Attributes &= ~FileAttributes.ReadOnly;

objLocalFileInfo.Delete();

}

//Create and write local stream from reading remote stream

objLocalStream = objLocalFileInfo.Create();

byte[] arrBuffer = new byte[4096];

int iBytesRead;

do

{

iBytesRead = objRemoteStream.Read(arrBuffer, 0, arrBuffer.Length);

objLocalStream.Write(arrBuffer, 0, iBytesRead);

} while (iBytesRead > 0);

}

finally

{

if (objClientProxy != null)

{

objClientProxy.Dispose();

objClientProxy = null;

}

if (objRemoteStream != null)

{

objRemoteStream.Close();

objRemoteStream = null;

}

if (objLocalStream != null)

{

objLocalStream.Close();

objLocalStream = null;

}

}

## Sample code

You can find sample code in the unit tests for the WCF streaming service which are located in the wcfTransferServiceTest.cs class file of the Memba.Tests project.

Memba Velodoc Outlook Add-In also uses the wcfTransferService WCF web service to transfer files to/from Velodoc XP Edition. You can download the source code and the documentation from <http://www.velodoc.com/download>. Memba.Transfer.PlugIns.WCFPlugIn.cs provides a sample implementation of file uploads and downloads using WCF streaming services with programmatic binding configuration.