HOWTO: Implement a Custom Web Document Viewer Handler

This article goes over how to create a custom Web Document Viewer handler. This allows you to provide custom image sources to the Web Document Viewer, whether it is a document loaded from a database, a procedurally generated image or simply a stream that doesn't map directly to a file in the relative web path.

The first step is to make sure you have the relevant references in your project. Make sure you have Atalasoft.dotImage.dll, Atalasoft.dotImage.Lib.dll, Atalasoft.dotImage.WebControls.dll, Atalasoft.dotImage.Lib.dll and Atalasoft.Shared.dll added to your project. You will also need any additional libraries you may be using.

The next step is to either create or include a "Generic Handler" (an .ashx file.) It can be either C# or VB.Net. I've attached pre-made handlers for both C# and VB.Net to this KB, so you can skip most of this and jump straight to grabbing those .ashx files.

Once you have that added to your project, open it up and update the definition of the class. Instead of Implementing the IHttpHandler interface we're going to inherit from Atalasoft.Imaging.WebControls. WebDocumentRequestHandler

(Please note that from here on out, you will need to replace our default " Handler " with whatever class name you have chosen. Also, I'm not going to assume you've added the relevant using or Imports statements to your handler and will include the full class name of all the classes we're using. Feel free to add them and save yourself some space.)

C#

```
public class Handler : WebDocumentRequestHandler
```

VB.Net

```
Public Class CustomWebDocumentViewer_VB
    Inherits WebDocumentRequestHandler
```

Then in your class constructor, add the handlers for the two events we're going to be working with, DocumentInfoRequested and ImageRequested .

```
public CustomWebDocumentViewer()
  {
    this .DocumentInfoRequested += new
 DocumentInfoRequestedEventHandler
(CustomWebDocumentViewer_DocumentInfoRequested);
    this .ImageRequested += new  ImageRequestedEventHandler
(CustomWebDocumentViewer_ImageRequested);
  }
```

VB.Net

```
  Public Sub CustomWebDocumentViewer_VB()
   AddHandler Me .DocumentInfoRequested, AddressOf Me
.CustomWebDocumentViewer_DocumentInfoRequested
   AddHandler Me .ImageRequested, AddressOf Me
.CustomWebDocumentViewer_ImageRequested
  End Sub
```

Okay, so all we have left to do is to implement those two events. Before we jump in to that, let me take a moment to explain what each event does. The DocumentInfoRequested event fires only once when .openUrl is called on the viewer or thumbnailer to build a basic document outline for the viewer. It gets the page count and page size before the viewer starts requesting pages. The ImageRequested event fires once per document page, and handles serving the page to the user.

C#

```
void CustomWebDocumentViewer_DocumentInfoRequested( object
sender, DocumentInfoRequestedEventArgs e)
{
  //e.FilePath //Requested image
  //e.PageCount //We need to return how many pages to load
  //e.PageSize //We need to return default page size.
```

```
  e.PageCount = CodeHereToGetActualNumberOfPages();
  e.PageSize = CodeHereToGetSizeOfFirstPage();
}

void CustomWebDocumentViewer_ImageRequested( object sender,
ImageRequestedEventArgs e)
{
  //e.FilePath //Tells you what image is requested
  //e.FrameIndex //What page is requested(0-indexed)
  //e.Image //Return the image to display to client here.
  e.Image = CodeHereToGetAtalaImageOfSpecificFrameDesired());
}
```

VB.Net

```
Public Sub CustomWebDocumentViewer_DocumentInfoRequested(
ByVal sender As Object , ByVal e As
Atalasoft.Imaging.WebControls.DocumentInfoRequestedEventArgs)
    'e.FilePath 'Requested image
    'e.PageCount 'We need to return how many pages to load
    'e.PageSize 'We need to return default page size.
    e.PageCount = CodeHereToGetActualNumberOfPages()
    e.PageSize = CodeHereToGetSizeOfFirstPage()
End Sub

Public Sub CustomWebDocumentViewer_ImageRequested( ByVal sender
As Object , ByVal e As ImageRequestedEventArgs)
    'e.FilePath 'Tells you what image is requested
    'e.FrameIndex 'What page is requested(0-indexed)
    'e.Image 'Return the image to display to client here.
```

```
    e.Image = CodeHereToGetAtalaImageOfSpecificFrameDesired()
End Sub
```

In the DocumentInfoRequested we're telling the viewer to expect a single page at 800, 600 and in the ImageRequested simply serving a new 800, 600 red AtalaImage. You will need to pass in the required information to either get or make your image(s) in the FilePath field. It should be fairly easy with this to use your own image repository to serve to the Web Document Viewer.

The last step is to go in to your .aspx file where you have the WebDocumentViewer and change the _serverUrl to point to your new custom handler.

```
< script type ="text/javascript" language ="javascript" >
    var _docUrl = 'document12345' ;
    var _serverUrl = 'CustomWebDocumentViewer.ashx' ;
    //You get the idea.
```

That _docUrl will be passed to your handler in the e.FilePath to both the DocumentInfoRequested and ImageRequested events. In your client-side javascript, you can change the document using _viewer.OpenUrl("newDocument") to change the currently loaded document, again whatever you pass in that Open will get passed directly as the e.FilePath

Example Handler with All Events

C#

```
using System;
using System.Web;
using Atalasoft.Imaging.Codec;
using Atalasoft.Imaging.Codec.Pdf;
```

```
using Atalasoft.Imaging.WebControls;

public class WebDocViewerHandler : WebDocumentRequestHandler
{

    static WebDocViewerHandler()
    {
       // for PDF support you need to reference
Atalasoft.dotImage.PdfReader.dll and have a PdfReader license

        RegisteredDecoders .Decoders.Add( new PdfDecoder ()
       {
           Resolution = 200,
           RenderSettings = new RenderSettings ()
           {
              AnnotationSettings = AnnotationRenderSettings
.RenderNone
           }
       });

       //// This adds the OfficeDecoder .. you need proper
licensing and additional OfficeDecoder
       //// dependencies (perceptive filter dlls)
       //RegisteredDecoders.Decoders.Add(new OfficeDecoder() {
Resolution = 200 });
    }

    public WebDocViewerHandler()
    {

        //// ************************** BASE DOCUMENT VIEWING
```

```
EVENTS **************************

        //// these two events are the base events for the loading
of pages

        //// the documentInfoRequested fires to ask for the
number of pages and size of the first page (minimum requirements)

        //// then, each page needed (and only when it's needed -
lazy loading) will fire an ImageRequested event

        //this.DocumentInfoRequested += new
DocumentInfoRequestedEventHandler(WebDocViewerHandler_DocumentInfoRequested)

        //this.ImageRequested += new
ImageRequestedEventHandler(WebDocViewerHandler_ImageRequested);


        //// ************************** ADDITIONAL DOCUMENT
VIEWING EVENTS **************************

        //// These events are additional/optional events ..

        //this.AnnotationDataRequested +=
WebDocViewerHandler_AnnotationDataRequested;

        //this.PdfFormRequested +=
WebDocViewerHandler_PdfFormRequested;


        //// this is the event that would be used to manually
handle page text requests

        //// (if left unhandled, page text requested wil be
handled by the default engine which will provide searchabel text
for Searchable PDF and office files)

        //// Manually handling this event is for advanced use
cases only

        //this.PageTextRequested +=
WebDocViewerHandler_PageTextRequested;
```

```
        //this.DocumentSave += WebDocViewerHandler_DocumentSave;
        //this.DocumentStreamWritten +=
WebDocViewerHandler_DocumentStreamWritten;
        //this.AnnotationStreamWritten +=
WebDocViewerHandler_AnnotationStreamWritten;

        //// This event is save related but conditional use - see
comments in handler
        //this.ResolveDocumentUri +=
WebDocViewerHandler_ResolveDocumentUri;
        //this.ResolvePageUri +=
WebDocViewerHandler_ResolvePageUri;

        //// ************************** OTHER EVENTS (usually
ignored) **************************
        //this.ReleaseDocumentStream +=
WebDocViewerHandler_ReleaseDocumentStream;
        //this.ReleasePageStream +=
WebDocViewerHandler_ReleasePageStream
    }

   #region BASE DOCUMENT VIEWING EVENTS
   /// <summary>
   /// The whole DocumentOpen process works like this:
   /// On initial opening of the document, DocumentInfoRequested
fires once for the document
   /// The AnnotationDataRequested event may also fire if
coditions merit\
   /// then the ImageRequested event will fire for subsequent
pages..
   ///
   /// This event fires when the first request to open the
```

```
document comes in

   /// you can either set e.FilePath to a different value (useful
for when the

   /// paths being opened simply alias to physial files somewhere
on the server)

   /// or you can manually handle the request in qhich case you
MUST set both e.PageSize and e.PageCount

   /// </summary>

   /// <param name="sender"></param>

    /// <param name="e"></param>

    void WebDocViewerHandler_DocumentInfoRequested( object
sender, DocumentInfoRequestedEventArgs e)

    {

       ////THESE TWO MUST BE RESOLVED IF YOU ARE MANUALLY
HANDLING

        //// set e.PageCount to the number of pages this document
has

        //e.PageCount =
SomeMethodThatGetsPageCountForDocumentRequested(e.FilePath);

        //// e.PageSize to the System.Drawing.Size you want pages
forced into

        //e.PageSize =
SomeMethodThatGetsSizeOfFirstPageOfDocument(e.FilePath);

    }
```

HOWTO: Implement a Custom Web Document Viewer Handler

```
   #endregion BASE DOCUMENT VIEWING EVENTS

    #region ADDITIONAL DOCUMENT VIEWING EVENTS
   /// <summary>
   /// When an OpenUrl that includes an AnnotationUri is called, the viewer default action is to go to the
   /// url specified ... which an XMP file containing all annotation layers (serialized LayerData[]) is read and loaded
   /// Manual handling of this event would be needed if one were to be loading annotations from a Byte[] or from a databasae, etc
   /// NOTE: this event WILL NOT FIRE if there was no annotationsUrl or a blank /null AnnotaitonsUrl was passed in the OpenUrl call
   /// </summary>
   /// <param name="sender"></param>
   /// <param name="e"></param>
   void WebDocViewerHandler_AnnotationDataRequested( object sender, AnnotationDataRequestedEventArgs e)
   {
       // READ-ONLY INPUT VALUE
       //e.FilePath = the passed in FilePath of where to find the annotations (when the OpenUrl is called, this will populate with the AnnotationsUrl value


       // WHAT YOU NEED TO HANDLE THIS EVENT
       // To successfully handle this event, you must populate the e.Layers
       //e.Layers = a LayerData[] containing ALL Annotation Layers for the whole document
   }
```

```
   /// <summary>
   /// This event lets you intercept requests for page text to provide your own
   /// manually handling this is not recommended, but the hook is here
   /// </summary>
   /// <param name="sender"></param>
   /// <param name="e"></param>
   void WebDocViewerHandler_PageTextRequested( object sender, PageTextRequestedEventArgs e)
   {
       // You can use the e.FilePath and e.Index to know which document and page are being requested respectively
       // you can then go extract text/ get the text to the control you must ultimately set
       // e.Page = .. an Atalasoft.Imaging.WebControls.Text.Page object containing the page text and position data
   }
```

```
    /// <param name="e"></param>
    void WebDocViewerHandler_PdfFormRequested( object sender,
PdfFormRequestedEventArgs e)
    {
        //// READ ONLY VALUE
        //e.FilePath contains the original File path in the intial
request use this to figure out which file/document/record to
fetch
        //// Required if handling you must provide an
Atalasoft.PdfDoc.Generating.PdfGeneratedDocument
        //// containing the PDF with the fillable form if you
manually handle this event
        //FileStream fs = new System.IO.FileStream(e.FilePath,
FileMode.Open, FileAccess.Read, FileShare.Read);
        //e.PdfDocument = new
Atalasoft.PdfDoc.Generating.PdfGeneratedDocument(fs);
        // If you return NULL for e.PdfDocument then the system
will treat the PDF as not being a fillable PDF form
    }
    #endregion ADDITIONAL DOCUMENT VIEWING EVENTS

     #region DOCUMENT SAVING EVENTS
    /// <summary>
    /// This event fires initially on DocumentSave
    /// Document saving rundown:
    /// The DocumentSave fires first.. it gives you the chance to
provide alternate streams for
    /// e.DocumentStream and e.AnnotationStream
    /// Then when the Document is written the
DocumentStreamWritten event will fire.. e.DocumentStream will
```

```
give oyu access to the
   /// document stream
   /// Then the AnnotationStreamWritten will fire (if there were
annotations to save) and e.AnnotationStream will give you access
   /// to the annotation stream.
   /// </summary>
   /// <param name="sender"></param>
   /// <param name="e"></param>
   void WebDocViewerHandler_DocumentSave( object sender,
DocumentSaveEventArgs e)
   {

       //// If you want to handle Annotations / Document into a
database or similar,
       //// do this and then handle the writing to db in the
AnnotationStreamWritten / DocumentStreamWritten event
       //e.AnnotationStream = new MemoryStream();
       //e.DocumentStream = new MemoryStream();
       //// If you want to provide an alternate location.. you
can't modify e.FilePath - its read only
       //// so instead, do this
       //e.AnnotationStream = new
FileStream("FullyQualifiedPathForAnnotationsToWriteToHere",
FileMode.Create, FileAccess.ReadWrite, FileShare.Read);
       //e.DocumentStream = new
FileStream("FullyQualifiedPathForDocumentToWriteToHere",
FileMode.Create, FileAccess.ReadWrite, FileShare.Read);
    }
```

```
   /// e.AnnotationStream will contain the annotations that were
written

   /// The DocumentStreamWritten will ALWAYS fire before the
AnnotationStreamWritten

   /// even thought this event has e.DocumentStream.. it will
always be null when

   /// AnnotationStreamWritten fires.. use the
DocumentStreamWritten event to handle the DocumentStream

   /// </summary>

   /// <param name="sender"></param>

   /// <param name="e"></param>

   void WebDocViewerHandler_AnnotationStreamWritten( object
sender, DocumentSaveEventArgs e)

   {

       //// You should rewind first, then you can store in a DB
or similar

       //// EXAMPLE: passed an e.MemoryStream for writing to a
DB:

       //MemoryStream annoStream = e.AnnotationStream as
MemoryStream

       //if (annoStream != null)

       //{

       //    annoStream.Seek(0, SeekOrigin.Begin);

       //
   SomeMethodToStoreByteArrayToDb(annoStream.ToArray());

       //}


       //// NOTE: if you set e.AnnotationStream to a file stream
in the DocumentSave you can skip handling

       //// this event and the system will take care of closing
```

```
it. Y ou would only need to handle this event
        //// if you are doing something else with it other than
letting it write to where you specified in that
        //// FileStream, such as post-processing or similar
    }

    /// <summary>
    /// e.DocumentStream will contain the entire document being
saved
    /// the DocumentStreamWritten will ALWAYS fire before
the AnnotationStreamWritten
    /// Even thought this event has e.AnnotationStream.. it will
always be null when
    /// DocumentStreamWritten fires.. use the
AnnotationStreamWritten event to handle the AnnotationStream
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    void WebDocViewerHandler_DocumentStreamWritten( object sender,
DocumentSaveEventArgs e)
    {
        //// You should rewind first, then you can store in a DB
or similar
        //// EXAMPLE: passed am e.MemoryStream for writing to a
DB:
```

```
        // docStream.Seek(0, SeekOrigin.Begin);
        // SomeMethodToStoreByteArrayToDb(docStream.ToArray());
        //}

        //// NOTE: if you set e.DocumentStream to a file stream in
the DocumentSave you can skip handling
        //// this event and the system will take care of closing
it you would only need to handle this event
        //// if you are doing something else with it other than
letting it write to where you specified in that
        //// FileStream such as post-processing or similar
    }

    /// <summary>
    /// The ResolveDocumentUri event is only needed if you've
manually handled DocumentInfoRequested and ImageRequested
    /// so that the e.FilePath is not pointing to the original
document (before any alterations like
rotation/deletion/reordering etc..)
    /// The event fires on save
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    void WebDocViewerHandler_ResolveDocumentUri( object sender,
ResolveDocumentUriEventArgs e)
    {
        ////EXAMPLE:
        // e.DocumentStream =
SomeMethodThatReturnsAValidStreamObjectContainingTheFullOriginal(e.DocumentU
    }
```

```
    /// <summary>
    /// Fires when a source page stream is requested while performing save operation
    /// During document save it is necessary to get the source document pages to combine them into the destination stream.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
   void WebDocViewerHandler_ResolvePageUri( object sender, ResolvePageUriEventArgs e)
    {
       //// it is often best to leave this unhandled.. the ResolveDocumentUri is usually sufficient for situations where
       //// The time to use this event is if your original opened document is a
       //// c ombination of multiple different source streams/documents: it is for very advanced use cases
   }
   #endregion DOCUMENT SAVING EVENTS

   #region OTHER EVENTS
    ////    The events in this section are almost never used manually by customers..
     //// They may have some use in extremely difficult/complex use cases, but for the most part should be left
     //// un-handled in your custom handler .. let the control use its defaults
    /// <summary>
    /// Fires when a document release stream occurs on document save. This event is raised only for streams that were provided in
```

```
ResolveDocumentUri event.
    /// After document save operation it is necessary to release
the obtained in ResolveDocumentUri event document streams. Fires
once for each stream.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    void WebDocViewerHandler_ReleaseDocumentStream( object
sender, ResolveDocumentUriEventArgs e)
    {
        //// Usually just leaving this to the default works
fine...
        //// e.DocumentUri contains the original request uri
(path) for the document
        //// The default activity will pretty much be
        //e.DocumentStream.Close();
        //e.DocumentStream.Dispose();
    }

    /// <summary>
    /// Fires when a page release stream occurs on document save.
    /// This event is raised only for streams that were provided
in ResolvePageUri event.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    void WebDocViewerHandler_ReleasePageStream( object sender,
ResolvePageUriEventArgs e)
    {
        //// Consider leaving this event alone as the default
```

```
built in handling works well

        //// e.DocumentPageIndex - The index of the page in the
DocumentStream if it is a multi-page document. Default value is
0.

        //// e.DocumentUri - the original document location /
path that was passed in

        //// e.SourcePageIndex - The requested index of the page
in the document.

        //// e.DocumentStream - Stream used to override the
default file system save while saving the document.

        //// Setting this property will take precedence over
using the DocumentUri property.


        //// Manually handling: if the original ResolvePageUri
was used and you set e.DocumentUri to an alias value.. do that
here

        //// if you provided a DocumentStream in ResolvePageUri,
then you may need to call

        //e.DocumentStream.Close();

        //e.DocumentStream.Dispose();

    }

    #endregion OTHER EVENTS

}
```

A note about .NET Core support

With the release of DotImage 11.0, we have added support for .NET Core web applications under .NET Framework.

.NET Core does not use generic handlers... however, you can still get at the WDV middleware to handle the same types of events.

HOWTO: Implement a Custom Web Document Viewer Handler

Please see the following articles for more:

[FAQ: Support for ASP.NET Core / .NET Core]

[INFO: Changes Introduced in DotImage 11.0]

Original Article:

Q10347 - HOWTO: Implementing a Custom Web Document Viewer Handler

Atalasoft Knowledge Base
https://www.atalasoft.com/kb2/KB/50139/HOWTO-Implement-a-Custom-Web-Documen...