INFO: Changes Introduced in DotImage 10.7

DotImage 10.7 was released in August of 2016. There were several key changes introduced, some of which may "break" existing applications.

This document is meant to extend the official 10.7.0 release notes.

If you prefer a full set of release notes all in one place for easy searching:

INFO: 10.7 Full Release Notes

WebDocumentViewer

BREAKING

jQueryUI CSS

In the past, our atalaWebDocViewer.css included a link to our jquery-ui-1.8.14.custom.css. However, in order to support NuGet deployment, we needed to unlink these.

If you have a 10.6 or older solution that uses WebDocumentViewer/WebDocumentThumbnailer, you will need to add in a link to the jquery-ui-1.8.14.custom.css we provide BEFORE the markup on the page links in atalaWebDocumentViewer.css

```
<!-- Style for Web Viewer -->

<link href="WebDocViewer/jquery-ui-1.8.14.custom.css" rel="Stylesheet"
type="text/css" />

<link href="WebDocViewer/atalaWebDocumentViewer.css" rel="Stylesheet"
type="text/css" />
```

DocumentSave

Saving ~only~ Annotations

INFO: Changes Introduced in DotImage 10.7

If you have an existing application that uses the DocumentSave event to only save annotaitons in your DocumentSave event and you wish to prevent the document from saving, you can set the PreventDefaultSaving property in your DocumentSave handler to true:

```
e.PreventDefaultSaving = true;
```

When this value is set to true, the document itself will not be saved and the DocumentStreamWritten will never fire.

Change to Allow Specifying Save Format

WebDocumentViewer configuration now includes a savefileformat option. Use this to force saving to a specific file type.

Supported types: 'tif', 'tiff' (both TaggedImageFileFormat, just different file extensions), or 'pdf'

Example: force all saves to PDF

```
var _serverUrl = '/WebDocumentHandler.ashx';
var _docRul = 'Images/foo.tif';

var _viewer = new Atalasoft.Controls.WebDocumentViewer({
 'parent': $('#_container1'),
    'toolbarparent': $('#_toolbar1'),
    'serverurl': _serverUrl,
    'documenturl': _docUrl,
    'savepath': _savePath,
    'allowannotations': true,
    'savefileformat': 'pdf'
});
```

You can also set this value in the SaveFileFormat property of a DocumentSave handler in your WebDocumentRequestHandler

```
public WebDocViewerHandler()
{
    this.DocumentSave += new
DocumentSaveEventHandler(WebDocViewerHandler_DocumentSave);
}

void WebDocViewerHandler_DocumentSave(object sender,
DocumentSaveEventArgs e)
{
    e.SaveFileFormat = "pdf"; //Force it to PDF.
    //// other valid values
    //e.SaveFileFormat = "tif"; //Force it to Tif.
    //e.SaveFileFormat = "tiff"; //Force it to Tiff (same as
Tif just different extension).
}
```

Change to Handling of Client-side Provided Save Folder

Previous 10 10.7, the behavior of _viewer.save("prepend_"); would work as follows

```
_viewer.OpenUrl("SomeFile.tif");
_viewer.Save("sometext_");
```

would result in the saving of a file with the name "sometext_SomeFile.tif" to the default save directory

In 10.7, an unintended breaking change was introduced where the same actions result in a new folder named "sometext_" to be created in the default save directory and the file "SomeFile.tif" to be saved to it. If you were relying on prepending text to file names, this change could affect you. the issue is being actively looked into and we will update this

article when the fix is available

As a workaround, you can handle the DocumentSave, DocumentStreamWritten and AnnotationStreamWritten events in the WebDocumentRequestHandler

```
public WebDocViewerHandler()

{

    this.DocumentSave += new
DocumentSaveEventHandler(WebDocViewerHandler_DocumentSave);

    this.DocumentStreamWritten += new
DocumentSaveEventHandler(WebDocViewerHandler_DocumentStreamWritten);

    this.AnnotationStreamWritten += new
DocumentSaveEventHandler(WebDocViewerHandler_AnnotationStreamWritten);

}


/// <summary>

/// Prior to 10.7, save worked slightly differently. There's a known issue in
10.7.0-10.7.1 which requires that we override

/// the save paths with our own

/// </summary>

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

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

void WebDocViewerHandler_DocumentSave(object sender, DocumentSaveEventArgs e)

{


    // note: the e.FileName has the file extension stripped, so you'll need to
supply it manually...

    // At this time it's not possible to access information about the
original file extension a fix is forthcoming

    e.DocumentStream = new
System.IO.FileStream(HttpContext.Current.Request.MapPath(e.SaveFolder +
```

```
e.FileName + ".tif"), System.IO.FileMode.Create);

    e.AnnotationStream = new
System.IO.FileStream(HttpContext.Current.Request.MapPath(e.SaveFolder +
e.FileName + ".xmp"), System.IO.FileMode.Create);

}

/// <summary>

/// See the WebDocViewerHandler_DocumentSave comments for reasons

/// this code is used to close up the stream after having written it

/// </summary>

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

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

void WebDocViewerHandler_AnnotationStreamWritten(object sender,
DocumentSaveEventArgs e)

{

    if (e.AnnotationStream != null)

    {

        e.AnnotationStream.Close();

        e.AnnotationStream.Dispose();

    }

}

/// <summary>

/// See the WebDocViewerHandler_DocumentSave comments for reasons

/// this code is used to close up the stream after having written it

/// </summary>

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

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

void WebDocViewerHandler_DocumentStreamWritten(object sender,
```

```
DocumentSaveEventArgs e)
{
    if (e.DocumentStream != null)
    {
        e.DocumentStream.Close();
        e.DocumentStream.Dispose();
    }
}
```

NON-BREAKING / NEW FEATURES

Client Side Page Operations

Our WebDocumentViewer.document has had methods for delete, insert and move, but before 10.7, these were not yet wired up for public API usage. In 10.7, these features are now available for use.

Selectable Text

10.7 introduces support for clientside text search / highlighting and text selection/copy. There are a few prerequisites to getting this up and running

LICENSING NOTE:

The default implementation of Clientside selectable text relies upon having a "Searchable PDF" (so it only works when loading a PDF that is "searchable") and having a license for Atalasoft's PdfReader WITH text extraction. If you do not have a license for Text Extraction, you will not be able to take advantage of this feature "out of the box"

Enabling Selectable Text

There are two text-related configuration directives in the WDV

INFO: Changes Introduced in DotImage 10.7

allowtext: true/false

This must be set to true to enable text searching on the page. If it's set to false then _viewer.text functions will not be available and may throw errors if used.

mousetool: { **hookcopy**: *true/false*, **allowsearch**: *true/false* }

The mouse tool is used for more than just searching, but there are two text / search-related features you can control in this directive.

hookcopy

Set to true to allow clipboard copy using CTRL+C (when interactive text selection is enabled setting _viewer.setMouseTool(Atalasoft.Utils.MouseToolType.Text))

if this is set to false (default setting if not defined) then setting the mouse tool to Text will enable highlighting, but copy to clipboard will be disabled. Direct triggering of copy text to clipboard may still be performed using

```
_viewer.text.copySelected();
```

allowsearch

Set this to false to tell the WebDocumentViewer Toolbar to hide the Text search box. The default value is true if allowtext is true)

Example:

```
'mousetool': {
    text: {
        hookcopy: true,
        allowsearch: true
    }
}
```

Searchable Text: Determining If Text Search Is Available

If you need to determine whether a given page has searchable text, you can use

_viewer.text.isPageTextLoaded(pageindex);

NOTE: allowtext configuration must be true or else calling this function will error out.

Selectable Text: Programmatic Text Selection

WebDocumentViewer has a new property: text that text property has several functions

`.copySelected()`

Copies currently selected text to clipboard

`.getSelected()`

Returns formatted selected text as a js object

`.isPageTextLoaded(index)`

Returns true if the page at index has text loaded/available for search, false if not

index: index of the frame to check

`.selectPageText(index[, region][, line, word][, success][, fail])`

causes text to be selected on a given page...

index: index of frame for selection

region: optional region index

line: optional line index

word: optional word index

success: optional callback function to call on success of selection

fail: optional callback function to call if select fails

`.clearSelection()`

Clears current text selection

`.resetPageText([index])`

Clears text data for page or whole document depending on index

index: optional page index to clear. When called without index, Clears page text data for the whole document (all pages) and marks it for reload on next request. When called with index, resets just the specified page, marking it for reload on next request

`.reloadPageText([index][,success][,fail])`

Forces text to reload

index: optional index for page to reload. When called without index, it forces text load for whole document. When called with index, it forces just that page's text data to reload

success: optional callback function for successful reload

fail: optional callback function to call if reload fails

`.isPageTextLoaded([index])`

Indicates whether page text is loaded

index: Optional index for page in question - applies to whole document if index is omitted

`.search(query, pageIndex, callback)`

Searches text in the document and causes results to be highlighted

query: Search query must be at least 3 characters in length

startFrameIndex: technically optional, but not really (see callback below) if defined will start the search at the frame indicated... if omitted or set to null, it will start form the first page.

callback: technically optional except if you do not provide one, then the search won't actually work. If you aren't going to handle the callback from the serach you still want to define it. Just give it

function(){} to let the default search highlight

example: search the whole document for 'hello' and highlight found text using default

```
_viewer.text.search('hello', null, function() {});
```

Interactive Text Selection

If you would like to enable mouse selection of text, simply call

```
_viewer.setMouseTool(Atalasoft.Utils.MouseToolType.Text);
```

Please note: if you wish to enable copy/paste from mouse selected text, you must ensure you set the mousetool: { hookcopy: true } in your WDV Config (see above). If not explicitly enabled, the default will be that selection will function, but text will not be copied to clipboard when the user hits CTRL+C. However, you can use

```
_viewer.text.copySelected();
```

to trigger a copy action regardless of the setting of mousetool: { hookcopy: true/false }

Thumbnail Captions

Caption text can be added to a thumbnail in order to provide additional information or context for users.

AnnotationRotation

Added interactive annotation rotation capabilities as well as appropriate JavaScript APIs to perform rotation programmatically

Please see 10426 - INFO: New Feature (10.6.1.5) Rotation in WebDocumentViewer for more info... specifically have a look at the section "ROTATING ANNOTIONS INDIVIDUALLY"

ShowEditor for Text Annotation Objects

We've added the ability to show/hide the editor for TextAnnotations programmatically. You need to pass the specific annotation object to the function for it to work:

```
// just using for example
var annotation = _viewer.getAnnotationsFromPage(0)[0];

// show the editor
_viewer.annotations.showEditor(annotation);
```

```
// hide the editor

_viewer.annotations.hideEditor(annotation);
```

On hiding, any updated text is saved.

PDF Improvements

Linearized PDF

DotImage 10.7 introduces support to save PDFs with Linearization (Also called "Fast Web View")

PdfEncoder

when using PdfEncoder to save to Linearized PDF, simply set the Linearized property to true:

```
PdfEncoder enc = new PdfEncoder();

enc.SizeMode = PdfPageSizeMode.FitToPage();

enc.Linearized = true;

enc.Save(saveStream, ImageSource, null);
```

PdfDocument and **PdfGeneratedDocument**

When using either of these classes to save to Linearized PDF, simply use the Save overload which supports PDfSaveOptions

```
PdfDocument doc = new PdfDocument("SomeIncoming.pdf");

doc.Save("NowLinearized.pdf", new PdfSaveOptions() { Linearized = true });
```

Compression with ImageSegmentation

The PdfEncoder was updated to support an advanced image segmentation algorithm to split the image into several segments with different color information (text, logo image, etc.) and compress these segments separately using the best possible algorithm.

INFO: Changes Introduced in DotImage 10.7

In order to enable Segmentation when saving with the PdfEncoder, simply set pdfEncoder.Mode = PdfCompressionMode.Segmented

Please note: There was a known issue with memory leak in the Segementation code in the initial release. This issue was addressed in 10.7.0 FixPack 3 (10.7.0.3.02788)If you're seeing OutOfMemoryException while using PdfEncoder with Segmentation enabled, the fix is to update to 10.7.0.3.02788 or later.

Also note: before FixPack 3 there was a known issue where using Segmentation would force grayscale images to black and white (binary). The fix involved adding a new event called ProcessSegment to the PdfEncoder. This event can be handled so that you can make active decisions about weather you want the segment to be BlackAndWhite, Grayscale, or Color

Example:

```
private void MakePdfFromSingleImage(string inFile, string outFile)
{
   AtalaImage image = new AtalaImage(inFile);
    PdfEncoder enc = new PdfEncoder();

    // enable segmentation
    enc.Mode = PdfCompressionMode.Segmented;

    // This sets the correct handler to choose segmentation as desired
    enc.ProcessSegment += new
EventHandler<SegmentProcessEventArgs>(enc_ProcessSegment);

    image.Save(outFile, enc, null);
}
```

```
{
    switch (e.Segment.PixelFormat)
    {
        // first the types we want to make bitonal (Black and white)
        // Move any case you want Black and white up here
        case PixelFormat.Pixel1bppIndexed:
            e.ColorType = SegmentColorType.BlackAndWhite;
            break;
        // now, any format you want to map to grayscale...
        case PixelFormat.Pixel4bppIndexed:
        case PixelFormat.Pixel16bppGrayscale:
        case PixelFormat.Pixel8bppGrayscale:
            e.ColorType = SegmentColorType.Gray;
            break;
        // finally, everything else is color
        default:
            e.ColorType = SegmentColorType.Color;
            break;
    }
}
```

Visual Studio Integration

NuGet

If you are using Visual Studio 2010 or newer, Atalasoft components are now available as NuGet packages in the NuGet Package Manager.

Activation Wizard Plugin

If you are using Visual Studio 2010 or newer, the Atalasoft Activation wizard is now available as a Visual Studio Plugin (you can still use the standard Atalsoft Activaion wizard, the Visual Studio Plugin version is just a "convenience feature").

To install:

- In Visual Studio, select the menu option for Tools -> Extension Manager
- Select the Online Gallery
- Search For Atalasoft DotImage Activation Wizard
- You can then click the Download to install the Atalasoft Activation wizard as a Visual Studio Extension.
- You will need to restart Visual Studio to complete the install The Activation Wizard
- Atalasoft Activation Wizard will now be an option in Visual Studio under the Tools menu

Office Decoder

DotImage 10.6 introduced our OfficeAdapterDecoder, but this was a COM wrapper around MS Office and required that MS Office be installed on the system which was opening the files. There were some serious performance and other drawbacks to this component (see Q10415 - INFO: OfficeAdapterDecoder Design Implications for details). If you are still using OfficeAdapterDecoder, it's still present in 10.7, but we have a much better solution...

OfficeDecoder

This component works like our other ImageDecoder classes - you need to reference the Atalsoft.dotImage.Office.dll which contains it and then you need to add it to your RegisteredDecoders.Decoders collection to use it:

```
RegisteredDecoders.Decoders.Add(new OfficeDecoder() { Resolution = 200 });
```

There is one other important requirement... although the OfficeDecoder does not require MS Office to be installed on the machine, it does require the "PerceptiveDocumentFilters" dlls .. and (here's the rub) you can't just directly add these as references in your project.... Visual Studio "doesn't like" that. So, what you need to do is to use Visual Studio Solution Explorer to Add Existing to the root of your project and add the following 4 files:

32-Bit Process:

```
C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-32\ISYS11df.dll

C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-32\ISYSreaders.dll

C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-32\ISYSreadershd.dll

C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-32\Perceptive.DocumentFilters.dll
```

64-bit Process:

```
C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-64\ISYS11df.dll

C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-64\ISYSreaders.dll

C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-64\ISYSreadershd.dll

C:\Program Files (x86)\Atalasoft\DotImage
10.7\bin\PerceptiveDocumentFilters\intel-64\Perceptive.DocumentFilters.dll
```

Then, set their CopyToOutputDirectory property to "Copy Always"

Alternately, just ensure the correct "Bitnes" version of the 4 files ends up in the bin directory of the application that is using the OfficeDecoder.

ABBYY OCR Engine

DotImage 10.7 has a new OcrEngine: AbbyyEngine. For the technical details, we've created a full KB article on just this subject.

Please see INFO:AbbyyEngine - Overview

Large TIFF Files

INFO: Changes Introduced in DotImage 10.7

The TIFF specification allows for TIFF files to be up to 4 GiB in size, however, past versions of DotImage had a 2 GiB limit. As of 10.7, we now fully support TIFF files up to 4GiB.

Please note: the TIFF specification itself does not allow for files larger than 4GiB, so it's still wise to "code defensively" to ensure you don't surpass that 4 GiB limit.

Original Article:

Q10436 - INFO: Changes Introduced in DotImage 10.7

Atalasoft Knowledge Base
https://www.atalasoft.com/kb2/KB/50064/INFO-Changes-Introduced-in-DotImage-...