EVO HTML to PDF Converter for .NET

User Guide

Table of Contents

6.6 Bookmarks

7. Licensing

1. Introduction

The EVO HTML to PDF Converter for .NET consists in a .NET library that can be used directly in any .NET application (ASP.NET, Windows Forms, Console, Web Services, Windows Services, etc). The converter does not require any installation and it does not use any printer driver or other third party components to perform conversion. The converter is delivered as a main assembly evohtmltopdf.dll that can be directly linked with your .NET application and a helper file evointernal.dat internally referenced by evohtmltopdf.dll. The full HTML / CSS set is supported and the main goal of the converter is to preserve unchanged the original aspect of the converted HTML page.

It can be used as a general purpose tool for converting web pages and HTML code to PDF or to easily create PDF reports directly from web pages. If you think that a converted ASP.NET page can contain server controls like charts, barcodes, data bound controls like data grids and repeaters you can realize how powerful this tool can be in creating rich formatted and easily maintainable PDF reports.

The converter API offers methods to convert a web page from an URL to PDF or to convert a HTML string to PDF. Additionally you can convert web pages and HTML code to images in any format supported by .NET framework (BMP, JPEG, PNG, GIF, etc).

If you want to get started immediately without reading the next sections of this document this is something perfectly possible. First you have to add a reference to the converter library assembly ***evohtmltopdf.dll*** in your .NET or ASP.NET project. Then you have to add the following two lines of code in your application. The first one will import the converter namespace and the second one will call the converter to render the web page from the specified url as an array of bytes representing the resulted PDF document:

```
using EvoPdf;
byte[] pdfBytes = new PdfConverter().GetPdfFromUrlBytes(url);
```

Further you can save the PDF document bytes into a file on disk or you can send the bytes as a response to the client browser. We provide full sample applications, both in C# and VB.NET to exemplify both situations.

The code above will produce a PDF document based on the default settings of the library which is enough for the most of the situations. However, the converter library offers a large number of parameters that you can set to customize the conversion process. You can add headers and footers with text and images to the resulted PDF document, specify page orientation, page size, compression level of the resulted PDF document, encrypt the resulted document and set user and owner password, set the permissions for printing.

2. Installation

The EVO HTML to PDF Converter for .NET is delivered as a zip archive and it doesn't have an installer. You have to unzip the archive in a folder on the disk. Below is a brief description of the folders from the archive.

2.1 Bin Folder

Bin folder contains the .NET assemblies you can use in your application and a prebuilt Windows Forms sample application that you can use to quickly evaluate the converter in your environment.

evohtmltopdf.dll - is the HTML to PDF converter library that you can link in any .NET application, either Windows Forms or ASP.NET.
evohtmltopdf.xml - is the HTML to PDF converter API reference in XML format for Visual Studio IntelliSense.
quickdemo.exe - is a Windows Forms application that can be used to convert a web page from an URL or a HTML string to PDF or Image.

2.1 Doc Folder

Doc folder contains the HTML to PDF Converter manual and the API reference in chm and html format.

EvoHtmlToPdfApi.chm - contains the HTML to PDF converter library API reference

UserGuide.pdf and UserGuide.htm - contain this developer manual in PDF and HTML formats

2.1 Samples Folder

Samples folder contains C# and VB.NET full sample applications for .NET 2.0 and .NET 4.0 to offer you ready to use code for ASP.NET and Windows Forms applications. Each sample has a solution file .sln that you can directly open in Visual Studio.

EvoHtmlToPdfWebDemo - is an ASP.NET application which shows how to convert web pages and HTML code to PDF and images. The application includes many demo pages, each page showing various features of the converter.

EvoHtmlToPdfDemo - is a Windows Forms application which shows how to convert web pages and HTML code to PDF and images. The application includes many demo forms, each form showing various features of the converter.

3. Requirements and Recommendations

The recommended hardware and software resources for successfully running the EVO HTML to PDF converter for .NET are listed below. Basically this is the environment we used for testing the product.

Operating System: Windows XP, Windows 2003 Server, Windows Vista, Windows Server 2008, Windows 7, Windows Server 2008 R2, Windows 8, Windows Server 2012
Hardware Architecture: 32-bit, 64-bit (recommended to run the converter in a 64-bit process)
Free RAM: 2GB
Microsoft .NET Framework 2.0 or 4.0
Full trust level when used in ASP.NET applications

4. Converter API

The converter API is fully documented in the Doc/EvoHtmlToPdfApi.chm. In order to use the converter library you have include the EvoPdf namespace in your application. The main classes in this namespace are the *PdfConverter* class and the *ImgConverter* class which expose the methods you can use to render a PDF document or an image from a URL or a HTML string. Below is a brief description of the main classes and properties of the converter.

4.1 PdfConverter Class

This class defines a set of methods to render a PDF document from a URL or from a HTML string. The conversion result can be a stream of bytes as byte[] object or a file on the disk. The PDF bytes can be further saved in a disk file or can be send a HTTP response to the client browser.

4.1.1 PdfConverter Save/Render Methods

The method below converts the URL specified as parameter to a PDF document returned as bytes array. The bytes array can be sent to a web browser from an ASP.NET application, saved in a file on disk or into a database or sent by email as an attachment.

```
public byte[] GetPdfBytesFromUrl( string url )
```

To convert a HTML string to PDF you can use one of the following methods below. The first method simply renders the HTML string as a PDF document. The second one accepts an additional parameter *urlBase* which is the full URL of the page from where you have retrieved the HTML string. The *urlBase* parameter is used by converter to determine the full URL of the images and other external files like CSS and JavaScript referenced in the HTML string by a relative URL. If you don't set this parameter the images referenced by relative URLS won't appear in the document and the styles from external CSS files won't be applied to the rendered document.

```
public byte[] GetPdfBytesFromHtmlString (string htmlString)
public byte[] GetPdfBytesFromHtmlString (string htmlString,string urlBase)
```

The correspondent methods you can use to render the PDF document in disk file are listed below. These methods internally use the methods above to get the bytes array and then they simply save the bytes in the specified file on disk.

```
public void SavePdfFromUrlToFile (string url, string outFile)
public void SavePdfFromHtmlStringToFile (string htmlString, string outFile)
public void SavePdfFromHtmlStringToFile (string htmlString, string outFile, string urlBase)
```

The methods above first obtain the PDF document as a byte array and then simply return that array of bytes or save it in a file. There is a set of methods which can save the PDF document directly into a stream without storing it first in the memory. The name of these methods have the *ToStream* suffix instead of the *ToFile* suffix from the set of Save methods prototyped above:

```
public void SavePdfFromUrlToStream(string url, Stream outPdfStream)
public void SavePdfFromUrlToStream(string url, string internalLinksDocUrl, Stream outPdfStream)
public void SavePdfFromHtmlFileToStream(string htmlFilePath, Stream outPdfStream)
public void SavePdfFromHtmlFileToStream(string htmlFilePath, string internalLinksDocUrl, Stream outPdfStream)
public void SavePdfFromHtmlStringToStream(string htmlString, Stream outPdfStream)
public void SavePdfFromHtmlStringToStream(string htmlString, string urlBase, Stream outPdfStream)
public void SavePdfFromHtmlStringToStream(string htmlString, string urlBase, string internalLinksDocUrl, Stream outPdfStream)
public void SavePdfFromHtmlStreamToStream(System.IO.Stream htmlStream, Encoding streamEncoding, Stream outPdfStream)
public void SavePdfFromHtmlStreamToStream(System.IO.Stream htmlStream, Encoding streamEncoding, string urlBase, Stream outPdfStream)
public void SavePdfFromHtmlStreamToStream(System.IO.Stream htmlStream, Encoding streamEncoding, string urlBase, string internalLinksDocUrl, Stream outPdfStream)
```

There is also a set of methods returning a Document object that can be used to further modify the generated PDF document as described in the Post Convert Customization of the Generated PDF Document section by adding new PDF pages and new PDF elements to the document. The Document class and the elements that can be added to a Document object are defined in the *EvoPdf* namespace.

```
public Document GetPdfDocumentObjectFromUrl(string url)
public Document GetPdfDocumentObjectFromUrl(string url, string internalLinksDocUrl)
public Document GetPdfDocumentObjectFromHtmlFile(string htmlFilePath)
public Document GetPdfDocumentObjectFromHtmlFile(string htmlFilePath, string internalLinksDocUrl)
public Document GetPdfDocumentObjectFromHtmlString(string htmlString)
public Document GetPdfDocumentObjectFromHtmlString(string htmlString, string urlBase)
public Document GetPdfDocumentObjectFromHtmlString(string htmlString, string urlBase,
       string internalLinksDocUrl)
public Document GetPdfDocumentObjectFromHtmlStream(System.IO.Stream htmlStream, Encoding streamEncoding)
public Document GetPdfDocumentObjectFromHtmlStream(System.IO.Stream htmlStream, Encoding streamEncoding,
       string urlBase)
public Document GetPdfDocumentObjectFromHtmlStream(System.IO.Stream htmlStream, Encoding streamEncoding,
       string urlBase, string internalLinksDocUrl)
```

The generated Document object can be modified and then saved into memory, file or stream using one of the Save methods of the Doocument class.

4.1.2 PdfConverter Configuration Properties

The conversion process and the aspect of the generated PDF document can be configured in many ways. You can set the PDF document page size (A4, A3, etc), orientation (Portrait or Landscape), compression level, encryption and passwords, document info (author, title, subject, etc), add headers and footers with page numbering, etc. The main properties of the converter are listed below.

To set the license key you received after purchase and unlock the product you can use the *LicenseKey* property. If this property is not set with any value the converter will enter in demo mode.

```
public string LicenseKey { get; set; }
```

The *HtmlViewerWidth* and *HtmlViewerHeight* properties allows you to set the width and height of the HTML viewer windows. The web page content is rendered based on the HTML viewer width specified as an integer value in pixels. Setting these properties has the same effect as the effect produced when resizing a web page in a browser window to the specified dimensions.

The default value of the *HtmlViewerWidth* property is 1024 pixels. The default value of the *HtmlViewerHeight* property is 0 pixels which means the height will be automatically determined by the converter based on the HTML viewer width and the web page content. These values are producing good results in most of the cases but there are also some situations when you'll have to change these properties.

```
public int HtmlViewerWidth { get; set; }
public int HtmlViewerHeight { get; set; }
```

The *PdfDocumentOptions* property allows you to change the aspect and properties of the rendered PDF document like setting the margins, add header and footer, embed true type fonts, generate a document with selectable texts and images or a document with an embedded image, enable or disable live links, pdf page size and page orientation, compression level, show or hide the headers and footers.

This property exposes an object of *PdfDocumentOptions* type which is automatically created in the *PdfConverter* constructor. Therefore you don't have to set this property directly with a value from your code but you'll have to set the properties of the exposed PdfDocumentOptions object.

The main properties of the PdfDocumentOptions class are described in a later section.

```
public PdfDocumentOptions PdfDocumentOptions { get; }
```

The *PdfSecurityOptions* class property allows you to change the permissions of the rendered PDF document like allow or disallow printing, etiding, etc and also to set user and owner passwords.

This property exposes an object of *PdfSecurityOptions* type which is automatically created in the *PdfConverter* constructor. Therefore you don't have to set this property directly with a value from your code but you'll have to set the properties of the exposed *PdfDocumentOptions* object.

The main properties of the PdfDocumentOptions class are described in a later section.

```
public PdfSecurityOptions PdfSecurityOptions { get; }
```

The *PdfDocumentInfo* property allows you to set the rendered PDF description like title, author, subject, keywords, etc.

This property exposes an object of *PdfSecurityOptions* type which is automatically created in the *PdfConverter* constructor. Therefore you don't have to set this property directly with a value from your code but you'll have to set the properties of the exposed PdfDocumentOptions object.

The main properties of the PdfDocumentOptions class are described in a later section.

```
public PdfDocumentInfo PdfDocumentInfo { get; }
```

The *PdfHeaderOptions* and *PdfFooterOptions* properties allows you to customize the aspect of the headers and footers added to the rendered PDF document. Note that the header and footer are visible in the resulted PDF document only if the corresponding *ShowHeader* and *ShowFooter* properties from the *PdfDocumentOptions* property are true.

These properties expose objects of *PdfHeaderOptions* type and *PdfFooterOptions* type which are automatically created in the *PdfConverter* constructor. Therefore you don't have to set this property directly with a value from your code but you'll have to set the properties of the exposed PdfHeaderOptions and PdfFooterOptions object.

The main properties of the PdfHeadersOptions and PdfFooterOptions classes are described in a later section.

```
public PdfHeaderOptions PdfHeaderOptions { get; }
public PdfFooterOptions PdfFooterOptions { get; }
```

4.2 ImgConverter Class

This class defines a set of methods to render an image from an URL or from a HTML string. The conversion result can be a stream of bytes as byte[] object or a file on the disk. The image bytes can be further saved in a disk file or can be send a HTTP response to the client browser.

4.2.1 ImgConverter Save/Render Methods

The method below retrieves the image bytes from a URL. There is also a similar method which produces an System.Drawing.Image object from a specified URL. The second parameter allows you to specify the format of the resulted image as a value from the System.Drawing.Imaging.ImageFormat enumeration.

The URL must be anonymously accessible from the computer running your application otherwise a *'Get web page content cancelled or invalid URL supplied'* exception is thrown by the converter. The best way to debug this type of exception is to load the URL in the Internet Explorer browser running on the same machine with your application and see if the page is correctly loaded.

```
public byte[] GetImageBytesFromUrl (string url,ImageFormat format)
public Image GetImageFromUrl (string url)
```

To convert a HTML string to image you can use one of the following methods below. The first method simply renders the HTML string as a Image object or as a byte[]. The second one accepts an additional parameter *urlBase* which is the full URL of the page from where you have retrieved the HTML string. The *urlBase* parameter is used by converter to determine the full URL of the images and other external files like CSS and JavaScript referenced in the HTML string by a relative URL. If you don't set this parameter the images referenced by relative URLS won't appear in the document and the styles from external CSS files won't be applied to the rendered image.

You can notice there are similar methods producing a System.Drawing.Image object instead of a byte[].

```
public Image GetImageFromHtmlString (string htmlString,ImageFormat format)
public Image GetImageFromHtmlString (string htmlString,ImageFormat format,string urlBase)

public byte[] GetImageBytesFromHtmlString (string htmlString,ImageFormat format)
public byte[] GetImageBytesFromHtmlString (string htmlString,ImageFormat format,string urlBase)
```

The correspondent methods you can use to render the image in disk file are listed below. These methods internally use the methods above to get the bytes array and then they simply save the bytes in the specified file on disk.

```
public void SaveImageFromUrlToFile (string url, ImageFormat format, string outFile)

public void SaveImageFromHtmlStringToFile (string htmlString, ImageFormat format, string outFile)
public void SaveImageFromHtmlStringToFile (string htmlString, ImageFormat format, string outFile, string urlBase)
```

In the full API reference document you'll notice some other similar methods for converting a HTML stream to image or a HTML file to image file but they are derived from the methods described above.

4.2.2 ImgConverter Configuration Properties

The conversion process and the aspect of the generated image can be configured with the configuration properties below.

To set the license key you received after purchase and unlock the product you can use the *LicenseKey* property. If this property is not set with any value the converter will enter in demo mode.

```
public string LicenseKey { get; set; }
```

The *HtmlViewerWidth* and *HtmlViewerHeight* properties allows you to set the width and height of the HTML viewer windows. The web page content is rendered based on the HTML viewer width specified as an integer value in pixels. Setting these properties has the same effect as the effect produced when resizing a web page in a browser window to the specified dimensions.

The default value of the *HtmlViewerWidth* property is 1024 pixels. The default value of the *HtmlViewerHeight* property is 0 pixels which means the height will be automatically determined by the converter based on the HTML viewer width and the web page content. These values are producing good results in most of the cases but there are also some situations when you'll have to change these properties.

```
public int HtmlViewerWidth { get; set; }
public int HtmlViewerHeight { get; set; }
```

5. Features

In this section will be described the main features of the converter with code samples for each feature.

5.1 Headers and Footers

In order to show or hide the header or footer on the rendered document you have to set the ShowHeader and ShowFooter properties of the PdfDocumentOptions property of the PdfConverter class. For example, to add both footer and header to the generated document you can use the following code:

```
PdfConverter pdfConverter = new PdfConverter();
pdfConverter.PdfDocumentOptions.ShowHeader = true;
pdfConverter.PdfDocumentOptions.ShowFooter = true;
```

The aspect of the header and footer can be controlled with the PdfHeaderOptions and PdfFooterOptions properties of the PdfConverter object.

You can set the header and footer height and in header and footer you can add HTML, images and texts, you can set the background color draw a line in header or footer, add page numbering:

```
            pdfConverter.PdfHeaderOptions.HeaderBackColor = Color.WhiteSmoke;
            pdfConverter.PdfHeaderOptions.HeaderHeight = 50;
```

The dimensions are specified in points and a point is 1/72 inches. The A4 page size in points is 595x842. At a screen resolution of 96 dpi, a A4 PDF page has 794 pixels in width and 1123 pixels in height.

The *Page Numbering* can also be added in position in the header and footer as a TextElement element containing &p; and &P; placeholders for current page number and total number of pages.

Bellow is a complete example for adding HTML in header and footer and page numbering in footer. The code was taken from the Getting Started sample application for ASP.NET:

```
private void AddHeaderElements(PdfConverter pdfConverter)
{
    // set the header HTML area
    HtmlToPdfElement headerHtml = new HtmlToPdfElement(0, 0, 0,
            pdfConverter.PdfHeaderOptions.HeaderHeight,
            "<b>HTML in Header</b>", null, 1024, 0);
    headerHtml.FitHeight = true;
    headerHtml.EmbedFonts = cbEmbedFonts.Checked;
    pdfConverter.PdfHeaderOptions.AddElement(headerHtml);
}

private void AddFooterElements(PdfConverter pdfConverter)
{
    //write the page number
    TextElement footerText = new TextElement(0, pdfConverter.PdfFooterOptions.FooterHeight - 15,
                "This is page &p; of &P;  ",
                new System.Drawing.Font(new System.Drawing.FontFamily("Times New Roman"),
                10, System.Drawing.GraphicsUnit.Point));
    footerText.EmbedSysFont = true;
    footerText.TextAlign = HorizontalTextAlign.Right;
    pdfConverter.PdfFooterOptions.AddElement(footerText);

    // set the footer HTML area
    HtmlToPdfElement footerHtml = new HtmlToPdfElement(0, 0, 0,
                pdfConverter.PdfFooterOptions.FooterHeight,
                "<i>HTML in Footer</i>", null, 1024, 0);
    footerHtml.FitHeight = true;
    footerHtml.EmbedFonts = cbEmbedFonts.Checked;
    pdfConverter.PdfFooterOptions.AddElement(footerHtml);
}
```

5.2 Security Options

With the security options you have the possibility to allow or disallow printing, editing, copying, filling form fields, set a user password and an owner password. When you password protect the generated PDF document it is recommended to set both the user and owner password. When you enter the user password to open a password protected document the permissions restrictions you have set are applied. When you enter the owner password to open a password protected document the permissions restrictions are not applied and the document can be edited. Below you can see a sample code which you can use to set the security options of the generated

```
            pdfConverter.PdfSecurityOptions.CanCopyContent = true;
            pdfConverter.PdfSecurityOptions.CanEditContent =
                                            true;
            pdfConverter.PdfSecurityOptions.CanFillFormFields = true;
            pdfConverter.PdfSecurityOptions.CanPrint = true;
            pdfConverter.PdfSecurityOptions.CanEditAnnotations = true;
            pdfConverter.PdfSecurityOptions.CanAssembleDocument =
                                            true;

            pdfConverter.PdfSecurityOptions.KeySize = EncryptionKeySize.EncryptKey128Bit;
            pdfConverter.PdfSecurityOptions.UserPassword = "user";
            pdfConverter.PdfSecurityOptions.OwnerPassword = "owner";
```

5.3 Document Description

You can set the document description like author, title, subject, keyword using the PdfDocumentInfo property.

```
            pdfConverter.PdfDocumentInfo.AuthorName =
                                            "EvoPdf";
            pdfConverter.PdfDocumentInfo.Title = "PDF Document
                                            Info";
            pdfConverter.PdfDocumentInfo.Subject =
                                            "HTML to PDF Converter";
            pdfConverter.PdfDocumentInfo.Keywords = "HTML, PDF,
                                            Converter";
            pdfConverter.PdfDocumentInfo.CreatedDate = DateTime.Now;
```

5.4 Automatic and Custom Page Breaks, Keep Together

The converter supports the following CSS styles to control the page breaks: page-break-before:always, page-break-after:always and page-break-inside:avoid. For example, with the page-break-after:always style applied to a HTML element (image, text, etc) you instruct the converter to insert a page break right after that element is rendered.

By default the converter always tries to avoid breaking the text between PDF pages. You can disable this behavior using the PdfConverter.AvoidTextBreak property. Also you can enable the converter to avoid breaking the images between PDF pages using the PdfConverter.AvoidImageBreak . By default this property is false.

An advanced and very useful feature when creating PDF reports is the Keep Together feature which can be implemented with the page-break-inside:avoid style. This instructs the converter to avoid breaking the content of a group of HTML elements you want to keep together on the same page. If you think you can apply this style to a table, a table row or a div element you can easily understand the utility of this feature.

Below is an example of using the page-break-inside:avoid style. The table contains a large number of rows, each row containing an image in the left and a text in the right and we don't want a row to span on two pages.

```
<table>
        <tr style="page-break-inside : avoid">
            <td>
                <img width="100" height="100" src="img1.jpg">
            </td>
            <td>
                My text 1
            </td>
        </tr>

        <tr style="page-break-inside : avoid">
            <td>
                <img width="100" height="100" src="img2.jpg">
            </td>
            <td>
                My text 2
            </td>
        </tr>
</table>
```

5.5 Live HTTP Links

The converter can convert any HTTP link from the HTML document into a link in the PDF document. This works on links containing text, image or any other combination supported by the HTML code. This is the default behavior of the converter. If you don't want to get active links in the generated PDF document you can set PdfConverter.PdfDocumentOptions.LiveUrlsEnabled = false.

5.6 Merge Capabilities

The HTML to PDF Converter provides you with the possibility to append a PDF file or a list of PDF files to the conversion result. A *Document* object can be generated by the converter. A *Document* object can also be created from an external PDF document from disk or from a stream. The *Document* class has an *AppendDocument()* method which can be used to append other PDF documents created by the library or loaded from an external source.

For more details please take a look a the 'Append External PDF Documents' sample.

5.7 Enable/Disable JavaScript

The JavaScript execution is enabled by default during conversion. You can disable JavaScript by setting the *JavaScriptEnabled* property of the *PdfConverter* class on false. The property prototype is:

```
public bool JavaScriptsEnabled { get; set; }
```

5.8 Server Authentication

The converter offers support for HTTP authentication. For example the converter can handle *IIS authentication* types like Integrated *Windows Authentication* and *Basic Authentication*. The authentication is disabled by default. To enable authentication you have to set the ***AuthenticationOptions*** property of the PdfConverter object with the username and password of a Windows account on the machine hosting the web server. Below you can find sample code for setting the username and password for authentication when converting HTML to PDF:

```
                pdfConverter.AuthenticationOptions.Username = username;
                pdfConverter.AuthenticationOptions.Password = password;
```

The *ImgConverter* has similar properties.

5.9 Custom PDF Page Size

The converter can produce PDF documents with pages of any size. The page size is controlled by the ***PdfConverter.PdfDocumentOptions.PdfPageSize*** property of type PdfPageSize. You can set this property to standard values like A4,A3,etc or to a *PdfPageSize* object constructed with custom width and height values expressed in points. A point is 1/72 inch.

```
pdfConverter.PdfDocumentOptions.PdfPageSize = new PdfPageSize(200,300);
```

When a custom PDF page size is specified, the page orientation is automatically set to portrait or to landscape function of the width and height of the PDF page.

5.10 Bookmarks

The converter can produce bookmarks in the generated PDF document for a list of specified HTML tags. The bookmarking is controlled by the pdfConverter.PdfBookmarkOptions property and is enabled only when a list of HTML selectors is specified by the ***pdfConverter.PdfBookmarkOptions.HtmlElementSelectors*** property. For example, to enable bookmarking of the H1 and H2 tags you can use the following line of C# code:

```
pdfConverter.PdfBookmarkOptions.HtmlElementSelectors = new string[] { "H1", "H2" };
```

The tags to be bookmarked can be further filtered by CSS class name using the HTML selectors syntax. For example, to filter only the H1 and H2 tags having the CSS class bookmark, the following line of C# can be used:

```
pdfConverter.PdfBookmarkOptions.HtmlElementSelectors = new string[] { "H1[class=\"bookmark\"]", "H2[class=\"bookmark\"]" };
```

5.11 Internal Links in PDF

The converter automatically converts the HTML links with anchors found in the HTML document to internal links in PDF. This feature can be used to easily create table of contents in the generated PDF document.

A HTML link with anchor consists in two HTML elements : a link defined with by a ***<a href="#target">Internal Link</a>*** tag and the target of the link defined by a ***<a name="target">Link Target</a>*** tag. When the HTML to PDF converter finds this construction it automatically generates an internal link in PDF from "Internal Link" to "Link Target".

The generation of internal links can be disabled using the *PdfConverter.PdfDocumentOptions.InternalLinksEnabled = false*.

There are a few things to ensure in order to get the internal links correctly generated in the PDF document. When converting an URL to PDF the URL must be fully qualified. For example if a website MyWebsite has a default.aspx page with internal links which is automatically served by the web server when the address http://MyWebsite is typed in the web browser address bar, then converting directly the http://MyWebsite url might not produce the correct internal links because the converter is unable to determine the web page automatically served by the web server. Instead, when converting http://MyWebsite/default.aspx the internal links will always be correctly generated.

5.12 JPEG Compression of Images in PDF

The converter automatically compresses the images generated in PDF using the JPEG compression algorithm to highly reduce the size of the generated PDF document. The drawback of the JPEG compression is that it reduces the quality of the images. When the JPEG compression level is increased the quality of the images in the PDF decreases.

The PdfConverter.PdfDocumentOptions.JpegCompressionLevel property defines the current level used for JPEG compression on a scale from 0 to 100. When the compression level is 0 the compression is the worst and the image quality is the best. The default JPEG compression level used by the converter is 10 which offers a good balance between the images quality and the size of the generated PDF document.

If you want to obtain the best image quality it is recommended to set te JpegCompressionLevel to 0. It is also possible to completely disable the JPEG compression of the images by setting the PdfConverter.PdfDocumentOptions.JpegCompressionEnabled to false, but this is not recommended in general because of the large size of the generated document.

5.13 Retrieve HTML Elements Mapping to PDF

This is a very powerful feature of the converter which allows you to obtain the position in the generated PDF document for any HTML element. Knowing the position in the generated PDF document of any element from the HTML document allows you to create bookmarks for elements from the HTML document, create internal links between HTML elements, place texts or images over the HTML elements or assign a digital signature to a certain element from HTML.

This feature can accessed using the PdfConverter.HtmlElementsMappingOptions property. This property allows you defined a list with the HTML elements for which to retrieve position using a list of HTML selectors.

The HtmlElementsMappingOptions property must be set before calling the converter method.

The HTML elements mapping is returned in the PdfConverter.HtmlElementsMappingOptions.HtmlElementsMappingResult. The HtmlElementsMapping result is a collection of HtmlElementMapping objects which offers the PDF page index where the element was mapped by the converter and the rectangle where the element was rendered inside that page, the element HTML ID, the element tag name, the element text and the element outer HTML code.

A code sample showing how to use this feature to highlight a specified list of HTML elements in the generated PDF document is presented in the Advanced Post Convert Customization of the Generated PDF Document section.

5.14 Repeat HTML Table Header a o5.14 Repeat HTML Table Header or Footer on Each PDF Page

The converter can be instructed to repeat the header of a HTML table on each PDF page where this table is rendered using the **display:table-header-group the thead element of a HTML tablethead** element of a HTML table and can be instructed to repeat the footer of a HTML table on each PDF page where this table is rendered using the **display:table-footer-group** style for the **tfoot** element of a HTML table.

You can repeat the table head of any HTML table having a thead element if you define the following CSS style in your document:

```
thead {display: table-header-group;}
```

This feature is useful for long HTML tables spanning on many PDF pages such that the reader of the document can always know to which column belongs the data in a PDF page. The *RepeatHtmlTableOnEachPage* sample contains a complete example for this feature. You can have anything in the repeated table head, from text and images to any other HTML code.

5.15 Select PDF Standard (PDF/A, 5.15 Select PDF Standard (PDF/A, PDF/X)

By default the converter can generate PDF documents in conformance with PDF 1.4 standard. This standard is accepted by Adobe Reader 5.0 and the later versions of the Adobe Reader.
Using the *PdfConverter.PdfStandardSubset* property the converter can be in property the converter can be instructed to generate PDF documents in conformance with PDF/A and PDF/X standards. These standards impose additional restrictions to the generate document.

The PDF/A-1b standard (ISO 19005-1), used for long-term archiving of PDF documents, requires that all the true type fonts used by the document to be embedded in the document, the http links are disabled, the document does not use transparent objects, the document information properties are disabled.

5.16 Select Color Space (RGB, CMYK, Grayscale)

By default the converter uses the RGB color space to draw graphics and images. Using the *PdfConverter.ColorSpace* property it is possible to instruct the converter to use the CMYK or Grayscale color space when the PDF document is rendered.

5.17 Conversion Summary

The *PdfConverter.ConversionSummary* property returns an object populated after the call of one of the render methods described in PdfConverter Render Methods section with a set of data that can be used to display details about the conversion that has just finished or to further customize the generated PDF document as described in the Post Convert Customization of the Generated PDF Document section.

The most important properties of the ConversionSummary object are the *PdfPageCount* which represents the total number of pages rendered by the converter and the *RenderedPagesRectangles* which gives for each rendered PDF page the bounding rectangle for the content rendered in that page.

For example, if you want to add more elements at the end of the generated PDF document after a conversion, you can find where the conversion ended from the last rectangle of the *RenderedPagesRectangles* array. For this important scenario two explicit properties were also added to the conversion summary: *LastPageIndex* and *LastPageRectangle* giving the zero based index of the last redered page and the bounding rectangle of the content rendered in this last page.

5.18 Select Media Type

A HTML document can define multiple styles to target different media types using the *@media rules*. For example, a document can have style to be used when displaying the document in browser on a screen defined by a "*@media screen*" rule and a different style to be used when printing the HTML document defined by a "*@media print*".

The HTML to PDF converter can be instructed to render for a given media type using the *PdfConverter.MediaType* property. For example, in order to use the style for printing defined in the HTML document by the "*@media print*" rule, the *PdfConverter.MediaType* property must be set with "*print*" value.

5.19 Post Convert Customization of the Generated PDF Document

The converter offers a set of render methods described in Post Convert Customization Overview section returning the internal *Document* object created by the converter. The object resulted after conversion is an instance of the *EvoPdf.Document* class. This Document object offers access to the collection of pages of PDF document. You can iterate over the document pages, add new pages to the document, append external PDF documents or add new elements like text and images to the document pages. After modification the document can be saved to a file or to a stream using one of the Save methods of the Document class.

The chaper Advanced Post Convert Customization of the Generated PDF Document contains a detailed description of the classes and methods available in the *EvoPdf* namespace.

Below is a complete example of how to convert many HTML document into the same PDF document using post convert customization of the Document object. The code is taken from the *WinForms_MultipleHtmlConversions* sample. Each additional HTML to PDF conversion is represented by a HtmlToPdfElement object that can be added in any position in a PDF page. The first conversion is performed by the call to *pdfConverter.GetPdfDocumentObjectFromUrl*, the next two conversions (the conversion of another URL and the conversion of a HTML string) are achieved by adding two *HtmlToPdfElement* objects to the document pages:

```
using System;
using System.Data;
using System.Configuration;
```

```csharp
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

using System.Drawing;

using EvoPdf;

namespace EvoHtmlToPdfWebDemo
{
    public partial class ConvertMultipleURLs : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            ctrlDemoLinksBox.LoadDemo("ConvertMultipleUrls");
        }

        protected void btnConvert_Click(object sender, EventArgs e)
        {
            PdfConverter pdfConverter = new PdfConverter();

            // add header and footer
            if (cbAddHeader.Checked)
                AddHeader(pdfConverter);
            if (cbAddFooter.Checked)
                AddFooter(pdfConverter);

            // call the converter and get a Document object from URL
            Document pdfDocument = pdfConverter.GetPdfDocumentObjectFromUrl(
                        textBoxURL1.Text.Trim());

            // get the conversion summary object from the event arguments
            ConversionSummary conversionSummary = pdfConverter.ConversionSummary;

            // the PDF page where the previous conversion ended
            PdfPage lastPage = pdfDocument.Pages[conversionSummary.LastPageIndex];
            // the last rectangle in the last PDF page where the previous conversion ended
            RectangleF lastRectangle = conversionSummary.LastPageRectangle;

            // the result of adding an element to a PDF page
            // ofers the index of the PDF page where the rendering ended
            // and the bounding rectangle of the rendered content in the last page
            AddElementResult addResult = null;

            // the converter for the second URL
            HtmlToPdfElement htmlToPdfURL2 = null;

            if (cbStartOnNewPage.Checked)
            {
                // render the HTML from the second URL on a new page after the first URL
                PdfPage newPage = pdfDocument.Pages.AddNewPage();
                htmlToPdfURL2 = new HtmlToPdfElement(0, 0, textBoxURL2.Text);
                addResult = newPage.AddElement(htmlToPdfURL2);
            }
            else
            {
                // render the HTML from the second URL immediately after the first URL
                htmlToPdfURL2 = new HtmlToPdfElement(lastRectangle.Left,
                        lastRectangle.Bottom, textBoxURL2.Text);
                addResult = lastPage.AddElement(htmlToPdfURL2);
            }

            // the PDF page where the previous conversion ended
            lastPage = pdfDocument.Pages[addResult.EndPageIndex];

            // add a HTML string after all the rendered content
            HtmlToPdfElement htmlStringToPdf = new HtmlToPdfElement(addResult.EndPageBounds.Left,
                    addResult.EndPageBounds.Bottom,
                "<b><i>The rendered content ends here</i></b>", null);

            lastPage.AddElement(htmlStringToPdf);

            byte[] pdfBytes = null;

            try
            {
                pdfBytes = pdfDocument.Save();
            }
            finally
            {
                // close the Document to realease all the resources
                pdfDocument.Close();
            }

            // send the generated PDF document to client browser

            // get the object representing the HTTP response to browser
            HttpResponse httpResponse = HttpContext.Current.Response;

            // add the Content-Type and Content-Disposition HTTP headers
            httpResponse.AddHeader("Content-Type", "application/pdf");
            httpResponse.AddHeader("Content-Disposition",
                    String.Format("attachment; filename=ConvertMultipleUrls.pdf; size={0}",
                    pdfBytes.Length.ToString()));

            // write the PDF document bytes as attachment to HTTP response
            httpResponse.BinaryWrite(pdfBytes);

            // Note: it is important to end the response, otherwise the ASP.NET
            // web page will render its content to PDF document stream
            httpResponse.End();
        }
        private void AddHeader(PdfConverter pdfConverter)
        {
            string thisPageURL = HttpContext.Current.Request.Url.AbsoluteUri;
            string headerAndFooterHtmlUrl = thisPageURL.Substring(0,
                thisPageURL.LastIndexOf('/')) + "/HeaderFooter/HeaderAndFooterHtml.htm";

            //enable header
            pdfConverter.PdfDocumentOptions.ShowHeader = true;
            // set the header height in points
            pdfConverter.PdfHeaderOptions.HeaderHeight = 60;
            // set the header HTML area
            HtmlToPdfElement headerHtml = new HtmlToPdfElement(0, 0, 0,
                    pdfConverter.PdfHeaderOptions.HeaderHeight,
                    headerAndFooterHtmlUrl, 1024, 0);
            headerHtml.FitHeight = true;
            pdfConverter.PdfHeaderOptions.AddElement(headerHtml);
        }

        private void AddFooter(PdfConverter pdfConverter)
        {
            string thisPageURL = HttpContext.Current.Request.Url.AbsoluteUri;
            string headerAndFooterHtmlUrl = thisPageURL.Substring(0,
                    thisPageURL.LastIndexOf('/')) + "/HeaderFooter/HeaderAndFooterHtml.htm";

            //enable footer
            pdfConverter.PdfDocumentOptions.ShowFooter = true;
            // set the footer height in points
            pdfConverter.PdfFooterOptions.FooterHeight = 60;
            //write the page number
            TextElement footerText = new TextElement(0, 30, "This is page &p; of &P;  ",
                new System.Drawing.Font(new System.Drawing.FontFamily("Times New Roman"),
                    10, System.Drawing.GraphicsUnit.Point));
            footerText.EmbedSysFont = true;
            footerText.TextAlign = HorizontalTextAlign.Right;
            pdfConverter.PdfFooterOptions.AddElement(footerText);

            // set the footer HTML area
            HtmlToPdfElement footerHtml = new HtmlToPdfElement(0, 0, 0,
                    pdfConverter.PdfFooterOptions.FooterHeight,
                    headerAndFooterHtmlUrl, 1024, 0);
            footerHtml.FitHeight = true;
            pdfConverter.PdfFooterOptions.AddElement(footerHtml);
        }
    }
}
```

5.20 Multithreading Support

The multithreading support was one of the key factors in the design of the HTML to PDF library because most of the applications today are multithreaded applications. An ASP.NET application is a very good example of multithreaded application because at each request a user makes for a web page the ASP.NET subsystem from IIS creates a new thread in the worker process to execute the page and to return the result to the browser. Because many of the web applications today are ASP.NET applications you can easily understand how important is for a library to perform correctly and reliably in a multithreaded environment.

The render and save methods of the PdfConverter object can be safely called from multiple threads of an application and all the memory and resources used by the converter are automatically disposed after each conversion. The library was designed and carefully tested to not leak any memory or system resources after a conversion. Because of this the library can be safely used in services running a very long period without interruption and in highly loaded systems.

6. Advanced Post Convert Customization of the Generated PDF Document

6.1 Post Convert Customization Overview

The *PdfConverter* class offers a set of render methods producing the internal *Document* object created by the converter.

```
public Document GetPdfDocumentObjectFromUrl(string url)
public Document GetPdfDocumentObjectFromUrl(string url, string internalLinksDocUrl)
public Document GetPdfDocumentObjectFromHtmlFile(string htmlFilePath)
public Document GetPdfDocumentObjectFromHtmlFile(string htmlFilePath, string internalLinksDocUrl)
public Document GetPdfDocumentObjectFromHtmlString(string htmlString)
public Document GetPdfDocumentObjectFromHtmlString(string htmlString, string urlBase)
public Document GetPdfDocumentObjectFromHtmlString(string htmlString, string urlBase,
       string internalLinksDocUrl)
public Document GetPdfDocumentObjectFromHtmlStream(System.IO.Stream htmlStream, Encoding streamEncoding)
public Document GetPdfDocumentObjectFromHtmlStream(System.IO.Stream htmlStream, Encoding streamEncoding,
       string urlBase)
public Document GetPdfDocumentObjectFromHtmlStream(System.IO.Stream htmlStream, Encoding streamEncoding,
       string urlBase, string internalLinksDocUrl)
```

The object resulted after conversion is an instance of the *EvoPdf.Document* class. This *Document* object offers access to the collection of pages of PDF document. You can iterate over the document pages, add new pages to the document, append external PDF documents or add new elements like text and images to the document pages. After modification the document can be saved to a file or to a stream using one of the Save methods of the Document class. The classes and methods that can be used to customize the generated PDF document are available in the *EvoPdf* namespace.

The most important class defined in the in the *EvoPdf* is the *Document* class. When the *Document* object is returned by one of the render methods above it already contains the PDF pages generated by the converter from the URL or from the HTML string being converted. The collection of PDF pages can be accessed with *Document.Pages* property. New pages with the desired size, orientation and margins can be added to the collection of pages and new elements can be added to any page in the collection.

Below in this chapter we offer a detailed description of the elements that can be added to a page of the PDF document. You can add a *HtmlToPdfElement* which makes possible multiple conversions in the same PDF document, a *HtmlToImage* element which embeds the image of a HTML document into the PDF document, you can add new texts and images, shapes, digital signatures, bookmarks, templates, watermarks, file attachments and notes.

In the code sample below taken from the *HTML Elements Location in PDF* demo application, all the *H1* and *IMG* elements and the elements with the ID *ID1* and *ID2* will be highlighted with a green rectangle in the generated PDF:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using EvoPdf;

namespace WinForms_HtmlElementsLocationInPdf
{
    public partial class HtmlElementsLocationInPdf : Form
    {
        public HtmlElementsLocationInPdf()
        {
            InitializeComponent();
        }

        private void btnConvert_Click(object sender, EventArgs e)
        {
            try
            {
                PdfConverter pdfConverter = new PdfConverter();

                // inform the converter about the HTML elements for which we want
                    the location in PDF
                // in this sample we want the location of IMG, H1 and H2 elements
                    and the elements having ID 'id1' or 'id2'
                pdfConverter.HtmlElementsMappingOptions.HtmlElementSelectors =
                                    new string[] { "IMG", "H1", "H2", "#id1", "#id2" };

                // call the converter and get a Document object from URL
                Document pdfDocument = pdfConverter.GetPdfDocumentObjectFromUrl(textBoxURL.Text.Trim());

                // iterate over the HTML elements locations and hightlight each element
                    with a green rectangle
                foreach (HtmlElementMapping elementMapping in pdfConverter.HtmlElementsMappingOptions.HtmlElementsMappingResult)
                {
                    // because a HTML element can span over many PDF pages the mapping

                    // of the HTML element in PDF document consists in a list of rectangles,
                    // one rectangle for each PDF page where this element was rendered
                    foreach (HtmlElementPdfRectangle elementLocationInPdf in elementMapping.PdfRectangles)
                    {
                        // get the PDF page
                        PdfPage pdfPage = pdfDocument.Pages[elementLocationInPdf.PageIndex];
                        RectangleF pdfRectangleInPage = elementLocationInPdf.Rectangle;

                        // create a RectangleElement to highlight the HTML element
                        RectangleElement highlightRectangle = new RectangleElement(pdfRectangleInPage.X, pdfRectangleInPage.Y,
                            pdfRectangleInPage.Width, pdfRectangleInPage.Height);
                        highlightRectangle.ForeColor = Color.Green;

                        pdfPage.AddElement(highlightRectangle);
                    }
                }

                // save the PDF bytes in a file on disk
                string outFilePath = System.IO.Path.Combine(Application.StartupPath, "Result.pdf");

                try
                {
                    pdfDocument.Save(outFilePath);
                }
                finally
                {
                    // close the Document to realease all the resources
                    pdfDocument.Close();
                }

                // open the generated PDF document in an external viewer
                DialogResult dr = MessageBox.Show("Open the rendered file in an external
                    viewer?", "Open Rendered File",
                    MessageBoxButtons.YesNo);
                if (dr == DialogResult.Yes)
                {
                    System.Diagnostics.Process.Start(outFilePath);
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                return;
            }
        }
    }
}
```

6.2 Coordinates System and Graphic Units

The PDF document uses the point (1/72 inch) as a graphic unit. All the coordinates and sizes are expected to be specified in points. Sometimes, when the size of an element (e.g. image) is known in pixels it is useful to convert pixels in points. The HTML to PDF Converter API offers the *UnitsConverter* class to convert pixels in points and points in pixels. Here are the prototypes of these static functions from the UnitsConverter class:

```
           public static float PointsToPixels(float points)
           public static float PixelsToPoints(float pixels)
```

The coordinates system origin is located in the top left corner. The positive X coordinates go to right and the positive Y coordinates go down like in the .NET Windows Forms graphics.

6.3 Document Class

This class represents a PDF document in the HTML to PDF Converter framework. A Document object can be received as result of one of the render methods described in the . When creating a Document object from an existing PDF document, the existing PDF document is passed to the Document class constructor as the path of the .pdf file on disk or as a stream containing the PDF document image. In either case the resulted Document object can be further modified. There are also *Document* constructors accepting a password when creating a *Document* object form a password protected PDF document. Here are the *Document* class constructors:

```
        public Document(Stream pdfStream)
        public Document(string pdfFileName)
        public Document(Stream pdfStream, string pdfPassword)
        public Document(string pdfFileName, string pdfPassword)
```

The *Document* class offers access to the PDF pages collection, PDF fonts collection, bookmarks, templates and viewer preferences. For example you can add a new page to the document pages collection using the *AddPage()* method, adding a new font to the fonts collection using the *AddFont()* methodor adding a new template using the *AddTemplate*() method.

6.4 PDF Renderers

A *PDF renderer* is an object capable of rendering PDF elements into the final PDF document. Currently the HTML to PDF Converter API defines two types of renderers derived from the ElementsRenderer abstract class: the *PdfPage* class which renders elements in a PDF page and the *Template* class which renders the elements in a template area repeated on each page of the PDF document. Both renderers are capable to render the same graphic elements types like HTML to PDF converter elements, shape elements texts and images but additionally the PdfPage renderer can render other interactive elements like PDF links, digital signature elements, attachments, text notes that cannot be rendered by a *Template* renderer.

6.4.1 PdfPage Class

The *PdfPage class* represents a page in the PDF document. A PdfPage class object can be instantiated using the *Document.AddPage()* interface. When calling the *AddPage()* method without parameters a page with the default margins, size and orientation is created. The margins are inherited from the Document.Margins property. The default orientation is portrait when adding the first page to the document or the previous page orientation is inherited if there was a previous page in document. The default size is A4 when adding the first page or the previous page size if there was a previous page in document. However, any of these properties can be specified when using other overloadings of the AddPage() method.

6.4.2 Template Class

The *Template class* represents a content which is repeated on each page of the PDF document. The templates can be used to create headers and footers, watermarks or any other content that needs to be repeated on each page of the PDF document.

A Template can render all the graphic elements (e.g HTML to PDF converter, texts, images, shapes) a PdfPage can render but it cannot render interactive elements like PDF links, attachments, text notes. A Template class object can be instantiated using the *Document.AddTemplate()* interface.

There two logical types of templates in HTML to PDF Converter: predefined templates like header and footer and custom templates. Both predefined and custom are instances of the Template class but the predefined templates dimensions are for example considered when calculating the available client area in a PDF page.

6.4.2.1 Predefined Templates

The predefined templates are the document header, footer. The predefined templates are automatically docked to the corresponding side of the PDF page.

For example the predefined header is automatically docked to the top of the PDF page which means the header location is the top left corner and the width of the header template is the width of the page. Also if a predefined header was set, the available client area in PDF page will start right under the header. Similar, if a predefined footer was set, it will be docked to the bottom of page and the available client area in PDF will end right above the footer template. It is recommended to set the predefined templates before starting to add elements to the document.

The HTML to PDF Converter API allows you to access the headers and footers for of the PDF document created by the converter for customizations but it doesn't allow you to change the header or footer template. The headers and footers can be created using the PdfConverter class interface.

6.4.2.2 Custom Templates

The custom templates can be used to add stamps or any other content that must be repeated in the same position on each page. In the code sample below, taken from the PDF Stamps sample, a watermark containing a text and an image is added to an existing document:

```
private void btnCreatePdf_Click(object sender, EventArgs e)
{
    string pdfToModify = textBoxPdfFilePath.Text.Trim();

    // create a PDF document
    Document document = new Document(pdfToModify);

    // set the license key
    document.LicenseKey = "ORIJGQoKGQkZCxcJGQoIFwgLFwAAAAA=";

    // get the first page the PDF document
    PdfPage firstPage = document.Pages[0];

    string logoTransImagePath = System.IO.Path.Combine(Application.StartupPath, @"..\..\img\evologo-100-trans.png");
    string logoOpaqueImagePath = System.IO.Path.Combine(Application.StartupPath, @"..\..\img\evologo-100.jpg");

    // add an opaque image stamp in the top left corner of the first page
    // and make it semitransparent when rendered in PDF
    ImageElement imageStamp = new ImageElement(1, 1, logoOpaqueImagePath);
    imageStamp.Transparency = 50;
    AddElementResult addResult = firstPage.AddElement(imageStamp);
    // add a border for the image stamp
    RectangleElement imageBorderRectangleElement = new RectangleElement(1, 1, addResult.EndPageBounds.Width,
                        addResult.EndPageBounds.Height);
    firstPage.AddElement(imageBorderRectangleElement);

    // add a template stamp to the document repeated on each document
        page
    // the template contains an image and a text

    System.Drawing.Image logoImg = System.Drawing.Image.FromFile(logoTransImagePath);

    // calculate the template stamp location and size
    System.Drawing.SizeF imageSizePx = logoImg.PhysicalDimension;

    float imageWidthPoints = UnitsConverter.PixelsToPoints(imageSizePx.Width);
    float imageHeightPoints = UnitsConverter.PixelsToPoints(imageSizePx.Height);

    float templateStampXLocation = (firstPage.ClientRectangle.Width - imageWidthPoints) / 2;
    float templateStampYLocation = firstPage.ClientRectangle.Height / 4;

    // the stamp size is equal to image size in points
    Template templateStamp = document.AddTemplate(new System.Drawing.RectangleF(templateStampXLocation, templateStampYLocation,
            imageWidthPoints, imageHeightPoints + 20));

    // set a semitransparent background color for template
    RectangleElement background = new RectangleElement(0, 0, templateStamp.ClientRectangle.Width, templateStamp.ClientRectangle.Height);
    background.BackColor = Color.White;
    background.Transparency = 25;
    templateStamp.AddElement(background);

    // add a true type font to the document
    System.Drawing.Font ttfFontBoldItalic = new System.Drawing.Font("Times New Roman", 10,
                System.Drawing.FontStyle.Bold | System.Drawing.FontStyle.Italic, System.Drawing.GraphicsUnit.Point);
    PdfFont templateStampTextFont = document.AddFont(ttfFontBoldItalic, true);

    // Add a text element to the template. You can add any other types
        of elements to a template like a HtmlToPdfElement.
    TextElement templateStampTextElement = new TextElement(3, 0, "This is the Stamp Text", templateStampTextFont);
    templateStampTextElement.ForeColor = System.Drawing.Color.DarkBlue;
    templateStamp.AddElement(templateStampTextElement);

    // Add an image with transparency to the template. You can add any
        other types of elements to a template like a HtmlToPdfElement.
    ImageElement templateStampImageElement = new ImageElement(0, 20, logoImg);
    // instruct the library to use transparency information
    templateStampImageElement.RenderTransparentImage = true;
    templateStamp.AddElement(templateStampImageElement);

    // add a border to template
    RectangleElement templateStampRectangleElement = new RectangleElement(0, 0, templateStamp.ClientRectangle.Width,
                templateStamp.ClientRectangle.Height);
    templateStamp.AddElement(templateStampRectangleElement);

    // dispose the image
    logoImg.Dispose();

    string outFilePath = System.IO.Path.Combine(Application.StartupPath, "PdfStamps.pdf");

    // save the PDF document to disk
    try
    {
        document.Save(outFilePath);
    }
    finally
    {
        document.Close();
    }

    DialogResult dr = MessageBox.Show("Open the saved file in an external
        viewer?", "Open Rendered File", MessageBoxButtons.YesNo);
    if (dr == DialogResult.Yes)
    {
        try
        {
            System.Diagnostics.Process.Start(outFilePath);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            return;
        }
    }
}
```

6.5 PDF Page Elements

A PDF page element represents anything that can be added to a renderer and it implements the *PageElement* abstract class. The HTML to PDF Converter defines two types of page elements: graphic elements that are inheriting the *PageGraphicElement* class and interactive page elements inheriting directly the *PageElement* class.

The graphic elements have common properties like fore color, back color, transparency, rotation angle. Examples of graphic elements are HTML to PDF Converter element, RTF to PDF Converter element, text element, image element, shapes elements. Examples of interactive page elements are internal link element, url link element, text note element, file attachment element.

6.5.1 Page Graphic Elements

The page graphic elements inherit the *PageGraphicElement* class. This class defines the following properties and methods that can be used to change the aspect of the element:

ForeColor - this properties defines the color used to draw a shape for example the color of a line when drawing a line element or the text color when drawing a text
BackColor - this property defines the background color of a graphic element, for example the background color of rectangle when rendering a rectangle element
Gradient - this property defines a gradient color to be used when filling a shape like a rectangle element
Rotate(rotateAngle) - this method is used to rotate clockwise the coordinates system with a specified angle before rendering the graphic element.
Transparency - this property is used to set the graphic element transparency. It is expected a value between 0 and 100 for the transparency. 0 means completely transparent and 100 means completely opaque.
LineStyle - property is used to set the line width, dash style, line join style, line cap style for the graphic elements rendering lines like rectangle element, ellipse element, line element.

In the next sections each of the graphic elements will be described in detail.

6.5.1.1 HTML to PDF Converter Element

The integrated HTML to PDF Converter is implemented by the *HtmlToPdfElement* graphic element. It offers the possibility to specify the position and the size of the PDF content rendered from HTML and the possibility to add many HTML to PDF conversions to same document.

A very useful feature is the possibility to know the size of the rendered content in each page when the rendered content spans on many pages. The information about the last rendered page can be taken from the *AddElementResult* object returned after adding the element to a renderer like a page or template.

The HtmlToPdfElement offer many constructors that basically calls the following two constructors with more or less default values for converting a URL or a HTML string to PDF:

```
       public HtmlToPdfElement(float x, float y, float width, float height,
            string urlToConvert, int htmlViewerWidth, int htmlViewerHeight)

       public HtmlToPdfElement(float x, float y, float width, float height,
            string htmlStringToConvert, string htmlStringBaseURL,
            int htmlViewerWidth, int htmlViewerHeight)
```

The variours constructor parameters are explined below.

The first constructor creates a URL to PDF converter element at the specified *x* and *y* coordinates with the specified *width* and *height*. The virtual browser width and height in pixels are specified by the *htmlViewerWidth* and *htmlViewerHeight* parameters.

x - The x position in points where the rendered content will be placed
y- The y position in points where the rendered content will be placed
width - The destination width in points for the rendered content. If the specified with is negative, the destination width will be given by the available width in page or template
height - The destination height in points for the rendered content. If the specified height is negative, the destination height will be auto determined so all the content can be rendered. Please note that the specified height is the effective height that will be rendered in the PDF document and does not include for example the empty spaces introduced by custom or automatic page breaks.
urlToConvert - The URL to convert to PDF
htmlViewerWidth - The virtual browser width in pixels. The default value is 1024 pixels. The effect of this parameter is similar with viewing the HTML page in a browser window with the specified width. When this parameter is negative, the converter will try to auto-determine the HTML page width from the HTML body element width.
htmlViewerHeight - The virtual browser height in pixels. The default value is 0 which means the height will be auto-determined. The effect of this parameter is similar with viewing the HTML page in a browser window with the specified width and height. When this paramter is negative, the converter will try to auto-determine the HTML page height from the HTML body element height.

The second constructor creates a HTML string to PDF converter element at the specified *x* and *y* coordinates with the specified *width* and *height*. The virtual browser width and height in pixels is specified by the *htmlViewerWidth* and *htmlViewerHeight* parameters.

htmlStringToConvert - The HTML string convert to PDF.
htmlStringBaseURL - The full URL of the page from where this string was taken used to resolve the images and CSS files referenced by a relative URL in the HTML string. This parameter is optional and the default value is NULL. When this parameter is NULL no base URL will be used. This paramter has effect only if the specified HTML string contains a HEAD element.

6.5.1.1.1 Page Breaks, Keep Together in HtmlToPdfElement

The converter supports the following CSS styles to control the page breaks: page-break-before:always, page-break-after:always and page-break-inside:avoid. For example, with the page-break-after:always style applied to a HTML element (image, text, etc) you instruct the converter to insert a page break right after that element is rendered.

By default the converter always tries to avoid breaking the text between PDF pages. You can disable this behavior using the PdfConverter.AvoidTextBreak property. Also you can enable the converter to avoid breaking the images between PDF pages using the PdfConverter.AvoidImageBreak . By default this property is false.

An advanced and very useful feature when creating PDF reports is the Keep Together feature which can be implemented with the page-break-inside:avoid style. This instructs the converter to avoid breaking the content of a group of HTML elements you want to keep together on the same page. If you think you can apply this style to a table, a table row or a div element you can easily understand the utility of this feature.

Below is an example of using the page-break-inside:avoid style. The table contains a large number of rows, each row containing an image in the left and a text in the right and we don't want a row to span on two pages.

```
<table>
        <tr style="page-break-inside : avoid">
            <td>
                <img width="100" height="100" src="img1.jpg">
            </td>
            <td>
                My text 1
            </td>
        </tr>

        <tr style="page-break-inside : avoid">
            <td>
                <img width="100" height="100" src="img2.jpg">
            </td>
            <td>
                My text 2
            </td>
        </tr>
</table>
```

6.5.1.1.2 Live HTTP Links in HtmlToPdfElement

The converter can convert any HTTP link from the HTML document into a link in the PDF document. This works on links containing text, image or any other combination supported by the HTML code. This is the default behavior of the converter. If you don't want to get active links in the generated PDF document you can set *HtmlToPdfElement.LiveUrlsEnabled* = *false*.

6.5.1.1.3 Enable/Disable JavaScript

The JavaScript execution is enabled by default during conversion. You can disable JavaScript by setting the *JavaScriptEnabled* property of the *PdfConverter* class on false. The property prototype is:

```
public bool JavaScriptsEnabled { get; set; }
```

6.5.1.1.4 Server Authentication in HtmlToPdfElement

The converter offers support for HTTP authentication. For example the converter can handle *IIS authentication* types like Integrated *Windows Authentication* and *Basic Authentication*. The authentication is disabled by default. To enable authentication you have to set the ***AuthenticationOptions*** property of the HtmlToPdfElement object with the username and password of a Windows account on the machine hosting the web server. Below you can find sample code for setting the username and password for authentication when converting HTML to PDF:

```
                htmlToPdfElement.AuthenticationOptions.Username = username;
                htmlToPdfElement.AuthenticationOptions.Password = password;
```

6.5.1.1.5 Bookmarks in HtmlToPdfElement

The HtmlToPdfElement can produce bookmarks in the generated PDF document for a list of specified HTML tags. The bookmarking is controlled by the htmlToPdfElement.PdfBookmarkOptions property and is enabled only when a list of HTML selectors is specified by the ***htmlToPdfElement.PdfBookmarkOptions.HtmlElementSelectors*** property. For example, to enable bookmarking of the H1 and H2 tags you can use the following line of C# code:

```
htmlToPdfElement.PdfBookmarkOptions.HtmlElementSelectors = new string[] { "H1", "H2" };
```

The tags to be bookmarked can be further filtered by CSS class name using the HTML selectors syntax. For example, to filter only the H1 and H2 tags having the CSS class bookmark, the following line of C# can be used:

```
htmlToPdfElement.PdfBookmarkOptions.HtmlElementSelectors = new string[] { "H1[class=\"bookmark\"]",  "H2[class=\"bookmark\"]" };
```

6.5.1.1.6 Internal Links in HtmlToPdfElement

The converter automatically converts the HTML links with anchors found in the HTML document to internal links in PDF. This features can be used to easily create table of contents in the generated PDF document.

A HTML link with anchor consists in two HTML elements : a link defined with by a ***<a href="#target">Internal Link</a>*** tag and the target of the link defined by a ***<a name="target">Link Target</a>*** tag. When the HTML to PDF converter finds this construction it automatically generates an internal link in PDF from "Internal Link" to "Link Target".

The generation of internal links can be disabled using the HtmlToPdfElement.InternalLinksEnabled = false.

There are a few things to ensure in order to get the internal links correctly generated in the PDF document. When converting an URL to PDF the URL must be fully qualified. For example if a website MyWebsite has a default.aspx page with internal links which is automatically served by the web server when the address http://MyWebsite is typed in the web browser address bar, then converting directly the http://MyWebsite url might not produce the correct internal links because the converter is unable to determine the web page automatically served by the web server. Instead, when converting http://MyWebsite/default.aspx the internal links will always be correctly generated.

6.5.1.1.7 Retrieve HTML Elements Mapping to PDF from HtmlToPdfElement

This is a very powerful feature of the converter which allows you to obtain the position in the generated PDF document for any HTML element. Knowing the position in the generated PDF document of any element from the HTML document allows you to create bookmarks for elements from the HTML document, create internal links between HTML elements, place texts or images over the HTML elements or assign a digital signature to a certain element from HTML.

This feature can accessed using the HtmlToPdfElement.HtmlElementsMappingOptions property. This property allows you define a list with the HTML element selectors for which to retrieve position using a list of HTML selectors.

The HtmlElementsMappingOptions property must be set before adding the HtmlToPdfElement to a PDF page.

The HTML elements mapping is returned in the HtmlToPdfElement.HtmlElementsMappingOptions.HtmlElementsMappingResult. The HtmlElementsMapping result is a collection of HtmlElementMapping object which offers the PDF page index where the element was mapped by the converter and the rectangle where the element was rendered inside that page, the element HTML ID, the element tag name, the element text and the element outer HTML code.

In the code sample below taken from the *HTML Elements Location in PDF* demo application, all the *H1* and *IMG* elements and the elements with the ID *ID1* and *ID2* will be highlighted with a green rectangle in the generated PDF:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using EvoPdf;

namespace WinForms_HtmlElementsLocationInPdf
{
    public partial class HtmlElementsLocationInPdf : Form
    {
        public HtmlElementsLocationInPdf()
        {
            InitializeComponent();
        }

        private void btnConvert_Click(object sender, EventArgs e)
        {
            try
            {
                PdfConverter pdfConverter = new PdfConverter();

                // inform the converter about the HTML elements for which we want
                    the location in PDF
                // in this sample we want the location of IMG, H1 and H2 elements
                    and the elements having ID 'id1' or 'id2'
                pdfConverter.HtmlElementsMappingOptions.HtmlElementSelectors =
                                    new string[] { "IMG", "H1", "H2", "#id1", "#id2" };

                // call the converter and get a Document object from URL
                Document pdfDocument = pdfConverter.GetPdfDocumentObjectFromUrl(textBoxURL.Text.Trim());

                // iterate over the HTML elements locations and hightlight each element
                    with a green rectangle
                foreach (HtmlElementMapping elementMapping in pdfConverter.HtmlElementsMappingOptions.HtmlElementsMappingResult)
                {
                    // because a HTML element can span over many PDF pages the mapping

                    // of the HTML element in PDF document consists in a list of rectangles,
                    // one rectangle for each PDF page where this element was rendered
                    foreach (HtmlElementPdfRectangle elementLocationInPdf in elementMapping.PdfRectangles)
                    {
                        // get the PDF page
                        PdfPage pdfPage = pdfDocument.Pages[elementLocationInPdf.PageIndex];
                        RectangleF pdfRectangleInPage = elementLocationInPdf.Rectangle;

                        // create a RectangleElement to highlight the HTML element
                        RectangleElement highlightRectangle = new RectangleElement(pdfRectangleInPage.X, pdfRectangleInPage.Y,
                            pdfRectangleInPage.Width, pdfRectangleInPage.Height);
                        highlightRectangle.ForeColor = Color.Green;

                        pdfPage.AddElement(highlightRectangle);
                    }
                }

                // save the PDF bytes in a file on disk
                string outFilePath = System.IO.Path.Combine(Application.StartupPath, "Result.pdf");

                try
                {
                    pdfDocument.Save(outFilePath);
                }
                finally
                {
                    // close the Document to realease all the resources
                    pdfDocument.Close();
                }

                // open the generated PDF document in an external viewer
                DialogResult dr = MessageBox.Show("Open the rendered file in an external
                    viewer?", "Open Rendered File",
                    MessageBoxButtons.YesNo);
                if (dr == DialogResult.Yes)
                {
                    System.Diagnostics.Process.Start(outFilePath);
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                return;
            }
        }
    }
}
```

6.5.1.2 HTML to Image Converter Element

The integrated HTML to Image Converter is implemented by the *HtmlToImageElement* graphic element. It basically includes the functionality of the HTML to PDF Converter product when converting to a PDF with embedded image and additionally offers the possibility to specify the position and the size of the rendered image and the possibility to add many HTML to Image conversions to same document.

A very useful feature is the possibility to know the size of the rendered content in each page when the rendered content spans on many pages. The information about the last rendered page can be taken from the *AddElementResult* object returned after adding the element to a renderer like a page or template.

The *HtmlToImageElement* offer many constructors that basically calls the following two constructors with more or less default values for converting a URL or a HTML string to image:

```
        public HtmlToImageElement(float x, float y, float width, float height,
              string urlToConvert, int htmlViewerWidth, int htmlViewerHeight)
        public HtmlToImageElement(float x, float y, float width, float height,
              string htmlStringToConvert, string htmlStringBaseURL,
              int htmlViewerWidth, int htmlViewerHeight)
```

The various constructor parameters are explained below.

The first constructor creates a URL to Image converter element at the specified *x* and *y* coordinates with the specified *width* and *height*. The virtual browser width and height in pixels are specified by the *htmlViewerWidth* and *htmlViewerHeight* parameters.

x - The x position in points where the rendered content will be placed
y- The y position in points where the rendered content will be placed
width - The destination width in points for the rendered content. If the specified with is negative, the destination width will be given by the available width in page or template
height - The destination height in points for the rendered content. If the specified height is negative, the destination height will be auto determined so all the content can be rendered. Please note that the specified height is the effective height that will be rendered in the PDF document and does not include for example the empty spaces introduced by custom or automatic page breaks.
urlToConvert - The URL to convert to PDF
htmlViewerWidth - The virtual browser width in pixels. The default value is 1024 pixels. The effect of this parameter is similar with viewing the HTML page in a browser window with the specified width. When this parameter is negative, the converter will try to auto-determine the HTML page width from the HTML body element width.
htmlViewerHeight - The virtual browser height in pixels. The default value is 0 which means the height will be auto-determined. The effect of this parameter is similar with viewing the HTML page in a browser window with the specified width and height. When this paramter is negative, the converter will try to auto-determine the HTML page height from the HTML body element height.

The second constructor creates a HTML string to Image converter element at the specified *x* and *y* coordinates with the specified *width* and *height*. The virtual browser width and height in pixels is specified by the *htmlViewerWidth* and *htmlViewerHeight* parameters.

htmlStringToConvert - The HTML string convert to PDF.
htmlStringBaseURL - The full URL of the page from where this string was taken used to resolve the images and CSS files referenced by a relative URL in the HTML string. This parameter is optional and the default value is NULL. When this parameter is NULL no base URL will be used. This paramter has effect only if the specified HTML string contains a HEAD element.

The sample code for the HtmlToPdfElement also contains sample code for the HtmlToImageElement:

6.5.1.3 Text Element And Fonts

The Text Element allows adding texts to a PDF document and is implemented by the *TextElement* class. The *TextElement* offers many constructors that basically calls the following constructor with more or less default values:

```
    public TextElement(float x, float y, float width, float height,
          string text, PdfFont font, PdfColor textColor)
```

The constructor creates a paginable text element that will be rendered in the specified rectangle using the specified width, height, font and color.

If the text pagination is not allowed (*Paginate* property of the *TextElement* is false), the text will be written on current page, the rendered text height being given by the minimum between the specified height and the available height on page. The remaining text and the text bounds inside the current page are returned in the *AddTextElementResult* object when the element is added to a renderer.

If the text pagination is allowed (*Paginate* property of the *TextElement* is true) and the text needs pagination (the specified height is bigger than the available space on page), the height parameter will be ignored and the text will be rendered to the end using the necessary height. The text bounds inside the last page and the last page index are returned in the AddTextElementResult object when the element is added to a renderer.

If the text pagination is allowed but the text does not need pagination (the specified height is less than the available space on page), the rendered text will be truncated to fit the specified height. The text bounds inside the current page and the last page index are returned in the *AddTextElementResult* object when the element is added to a renderer.

x - The start x coordinate where the text will be rendered
y- The start y coordinate where the text
width - The width of the destination rectangle
height - The height of the destination rectangle
text - The text to be rendered
font - The text font
textColor - The text color

Below is a sample code showing how to add text and fonts to a PDF document:

```
private void btnCreatePdf_Click(object sender, EventArgs e)
{
    // create a PDF document
    Document document = new Document();

    // set the license key
    document.LicenseKey = "ORIJGQoKGQkZCxcJGQoIFwgLFwAAAAA=";

    // add a page to the PDF document
    PdfPage firstPage = document.AddPage();

    // Create a Times New Roman .NET font of 10 points
    System.Drawing.Font ttfFont = new System.Drawing.Font("Times New Roman", 10, System.Drawing.GraphicsUnit.Point);
    // Create a Times New Roman Italic .NET font of 10 points
    System.Drawing.Font ttfFontItalic = new System.Drawing.Font("Times New Roman", 10,
                System.Drawing.FontStyle.Italic, System.Drawing.GraphicsUnit.Point);
    // Create a Times New Roman Bold .NET font of 10 points
    System.Drawing.Font ttfFontBold = new System.Drawing.Font("Times New Roman", 10,
                System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point);
    // Create a Times New Roman Bold .NET font of 10 points
    System.Drawing.Font ttfFontBoldItalic = new System.Drawing.Font("Times New Roman", 10,
                System.Drawing.FontStyle.Bold | System.Drawing.FontStyle.Italic, System.Drawing.GraphicsUnit.Point);

    // Create a Sim Sun .NET font of 10 points
    System.Drawing.Font ttfCJKFont = new System.Drawing.Font("SimSun", 10,
                System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point);
    // Create the PDF fonts based on the .NET true type fonts
    PdfFont newTimesFont = document.AddFont(ttfFont);
    PdfFont newTimesFontItalic = document.AddFont(ttfFontItalic);
    PdfFont newTimesFontBold = document.AddFont(ttfFontBold);
    PdfFont newTimesFontBoldItalic = document.AddFont(ttfFontBoldItalic);

    // Create the embedded PDF fonts based on the .NET true type fonts
    PdfFont newTimesEmbeddedFont = document.AddFont(ttfFont, true);
    PdfFont newTimesItalicEmbeddedFont = document.AddFont(ttfFontItalic, true);
    PdfFont newTimesBoldEmbeddedFont = document.AddFont(ttfFontBold, true);
    PdfFont newTimesBoldItalicEmbeddedFont = document.AddFont(ttfFontBoldItalic, true);

    PdfFont cjkEmbeddedFont = document.AddFont(ttfCJKFont, true);

    // Create a standard Times New Roman Type 1 Font
    PdfFont stdTimesFont = document.AddFont(StdFontBaseFamily.TimesRoman);
    PdfFont stdTimesFontItalic = document.AddFont(StdFontBaseFamily.TimesItalic);
    PdfFont stdTimesFontBold = document.AddFont(StdFontBaseFamily.TimesBold);
    PdfFont stdTimesFontBoldItalic = document.AddFont(StdFontBaseFamily.TimesBoldItalic);

    // Create CJK standard Type 1 fonts
    PdfFont cjkJapaneseStandardFont = document.AddFont(StandardCJKFont.HeiseiKakuGothicW5);
    PdfFont cjkChineseTraditionalStandardFont = document.AddFont(StandardCJKFont.MonotypeHeiMedium);

    // Add text elements to the document

    TextElement trueTypeText = new TextElement(0, 10, "True Type Fonts Demo:", newTimesFontBold);
    AddElementResult addResult = firstPage.AddElement(trueTypeText);

    // Create the text element
    TextElement textElement1 = new TextElement(20, addResult.EndPageBounds.Bottom + 10, "Hello World !!!!", newTimesFont);
    // Add element to page. The result of adding the text element is stored
        into the addResult object
    // which can be used to get information about the rendered size in
        PDF page.
    addResult = firstPage.AddElement(textElement1);

    // Add another element 5 points below the text above. The bottom of
        the text above is taken from the AddElementResult object
    // set the font size
    newTimesFontItalic.Size = 15;
    TextElement textElement2 = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "Hello World !!!!", newTimesFontItalic);
    textElement2.ForeColor = System.Drawing.Color.Green;
    addResult = firstPage.AddElement(textElement2);

    newTimesFontBoldItalic.Size = 20;
    TextElement textElement3 = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "Hello World !!!!", newTimesFontBoldItalic);
    textElement3.ForeColor = System.Drawing.Color.Blue;
    addResult = firstPage.AddElement(textElement3);

    TextElement stdTypeText = new TextElement(0, addResult.EndPageBounds.Bottom + 10, "Standard PDF Fonts Demo:", newTimesFontBold);
    addResult = firstPage.AddElement(stdTypeText);

    TextElement textElement4 = new TextElement(20, addResult.EndPageBounds.Bottom + 10, "Hello World !!!!", stdTimesFont);
    addResult = firstPage.AddElement(textElement4);

    stdTimesFontItalic.Size = 15;
    TextElement textElement5 = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "Hello World !!!!", stdTimesFontItalic);
    textElement5.ForeColor = System.Drawing.Color.Green;
    addResult = firstPage.AddElement(textElement5);

    stdTimesFontBoldItalic.Size = 20;
    TextElement textElement6 = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "Hello World !!!!", stdTimesFontBoldItalic);
    textElement6.ForeColor = System.Drawing.Color.Blue;
    addResult = firstPage.AddElement(textElement6);

    // embedded true type fonts

    TextElement embeddedTtfText = new TextElement(0, addResult.EndPageBounds.Bottom + 10, "Embedded True Type Fonts Demo:", newTimesFontBold);
    addResult = firstPage.AddElement(embeddedTtfText);

    // russian text
    TextElement textElement8 = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "Появление на свет!!", newTimesEmbeddedFont);
    addResult = firstPage.AddElement(textElement8);

    string outFilePath = System.IO.Path.Combine(Application.StartupPath, "TextsAndFontsDemo.pdf");

    // save the PDF document to disk
    document.Save(outFilePath);

    // close the PDF document to release the resources
    document.Close();

    DialogResult dr = MessageBox.Show("Open the saved file in an external
        viewer?", "Open Rendered File", MessageBoxButtons.YesNo);
    if (dr == DialogResult.Yes)
    {
        try
        {
            System.Diagnostics.Process.Start(outFilePath);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            return;
        }
    }
}
```

6.5.1.4 Image Element

The Image Element allows adding images to a PDF document and is implemented by the *ImageElement* class. The *ImageElement* offers many constructors that basically calls the following two constructors with more or less default values:

```
      public ImageElement(float x, float y, float destWidth, float destHeight,
                  string filePath)
      public ImageElement(float x, float y, float destWidth, float destHeight,
                  System.Drawing.Image imageObj)
```

The first constructor creates an *ImageElement* from the specified file that will be rendered at the position (x,y) with the (destWidth,destHeight) size.
The second constructor creates an *ImageElement* from the specified *System.Drawing.Image* object that will be rendered at the position (x,y) with the (destWidth,destHeight) size.

x - The X location where this element will be rendered
y- The Y location where this element will be rendered
destWidth - The destination rectangle width
destHeight- The destination rectangle height
filePath - The image file path
imageObj - The System.Drawing.Image object

Below is a sample code showing how to add images to a PDF document:

```
        // add a page to the PDF document
        PdfPage firstPage = document.AddPage();

        string imagesPath = System.IO.Path.Combine(Server.MapPath("~"), "Images");

        // display image in the available space in
                                        page and with a auto determined height to keep the aspect ratio
        ImageElement imageElement1 = new ImageElement(0, 0, System.IO.Path.Combine(imagesPath, "html-to-pdf-box-250.PNG"));
        AddElementResult addResult = firstPage.AddElement(imageElement1);

        // display image with the specified width
                                        and the height auto determined to keep the aspect ratio
        // the images is displayed to the right of the previous
                                        image and the bounds of the image inside the current page
        // are taken from the AddElementResult object
        ImageElement imageElement2 = new ImageElement(addResult.EndPageBounds.Right + 10, 0, 100,
                System.IO.Path.Combine(imagesPath,
                                        "html-to-pdf-box-250.PNG"));
        addResult = firstPage.AddElement(imageElement2);

        // Display image with the specified width and the specified
                                        height. It is possible for the image to not preserve the aspect ratio
        // The images is displayed to the right of
                                        the previous image and the bounds of the image inside the current page
        // are taken from the AddElementResult object
        ImageElement imageElement3 = new ImageElement(addResult.EndPageBounds.Right + 10, 0, 200, 100,
                System.IO.Path.Combine(imagesPath, "html-to-pdf-box-250.PNG"));
        addResult = firstPage.AddElement(imageElement3);
```

6.5.1.5 Shape Elements

The HTML to PDF Converter defines the following shape elements: *LineElement*, *RectangleElement*, *EllipseElement*, *EllipseArcElement*, *EllipseSliceElement*, *BezierCurveElement*, *PolygonElement*.

Below is a sample code showing how to add shapes to a PDF document:

```
        // add a page to the PDF document
        PdfPage firstPage = document.AddPage();

        // draw rectangle
        RectangleElement rectangle1 = new RectangleElement(10, 10, 150, 100);
        rectangle1.ForeColor = System.Drawing.Color.Blue;
        rectangle1.LineStyle.LineWidth = 5; // a 5 points line
                                        width
        rectangle1.LineStyle.LineJoinStyle = LineJoinStyle.RoundJoin;
        firstPage.AddElement(rectangle1);

        // draw colored rectangle
        RectangleElement rectangle2 = new RectangleElement(200, 10, 150, 100);
        rectangle2.ForeColor = System.Drawing.Color.Blue;
        rectangle2.BackColor = System.Drawing.Color.Green;
        firstPage.AddElement(rectangle2);

        // draw gradient colored rectangle
        RectangleElement rectangle3 = new RectangleElement(400, 25, 100, 50);
        rectangle3.ForeColor = System.Drawing.Color.Blue;
        rectangle3.Gradient = new GradientColor( GradientDirection.Vertical, System.Drawing.Color.Green, System.Drawing.Color.Blue);
        firstPage.AddElement(rectangle3);

        // draw ellipse
        EllipseElement ellipse1 = new EllipseElement(75, 200, 70, 50);
        ellipse1.ForeColor = System.Drawing.Color.Blue;
        ellipse1.LineStyle.LineDashStyle = LineDashStyle.Dash;
        firstPage.AddElement(ellipse1);

        // draw ellipse
        EllipseElement ellipse2 = new EllipseElement(275, 200, 70, 50);
        ellipse2.ForeColor = System.Drawing.Color.Blue;
        ellipse2.BackColor = System.Drawing.Color.Green;
        firstPage.AddElement(ellipse2);

        // draw ellipse
        EllipseElement ellipse3 = new EllipseElement(450, 200, 50, 25);
        ellipse3.ForeColor = System.Drawing.Color.Blue;
        ellipse3.Gradient = new GradientColor(GradientDirection.Vertical, System.Drawing.Color.Green, System.Drawing.Color.Blue);
        firstPage.AddElement(ellipse3);


        BezierCurveElement bezierCurve1 = new BezierCurveElement(10, 350, 100, 300, 200, 400, 300, 350);
        bezierCurve1.ForeColor = System.Drawing.Color.Blue;
        bezierCurve1.LineStyle.LineWidth = 3;
        bezierCurve1.LineStyle.LineJoinStyle = LineJoinStyle.RoundJoin;
        firstPage.AddElement(bezierCurve1);
        BezierCurveElement bezierCurve2 = new BezierCurveElement(10, 350, 100, 400, 200, 300, 300, 350);
        bezierCurve2.ForeColor = System.Drawing.Color.Green;
        bezierCurve2.LineStyle.LineWidth = 3;
        bezierCurve2.LineStyle.LineJoinStyle = LineJoinStyle.RoundJoin;
        firstPage.AddElement(bezierCurve2);
```

6.5.2 Page Interactive Elements

The page interactive elements inherit directly from the *PageElement* class. They add interactive features to the PDF document like internal and URL links, file attachments and links, popup text notes, digital signatures, etc.

In the next sections each of the graphic elements will be described in detail.

6.5.2.1 Digital Signature Element

Digital signatures can be added to a PDF document using the DigitalSignatureElement. Adding a digital signature requires a PKCS#12 certificate provided as a .pfx or a .p12 file.

Below is a sample showing how to add a digital signature to a PDF document:

```
private void btnCreatePdf_Click(object sender, EventArgs e)
{
    // create a PDF document
    Document document = new Document();

    // set the license key
    document.LicenseKey = "ORIJGQoKGQkZCxcJGQoIFwgLFwAAAAA=";

    // add a page to the PDF document
    PdfPage firstPage = document.AddPage();

    string logoImagePath = System.IO.Path.Combine(Application.StartupPath, @"..\..\img\evologo-250.png");
    string certificateFilePath = System.IO.Path.Combine(Application.StartupPath, @"..\..\certificates\evopdf.pfx");

    PdfFont pdfFont = document.Fonts.Add(new Font("Arial", 10, FontStyle.Regular, GraphicsUnit.Point));
    TextElement descriptionTextElement = new TextElement(0, 0,
        "A digital signature was applied on the logo image below. Click on
            the image to see the signature details", pdfFont);
    AddElementResult addResult = firstPage.AddElement(descriptionTextElement);

    // create the area where the digital signature will be displayed in
        the PDF document
    // in this sample the area is a logo image but it could be anything
        else
    ImageElement logoElement = new ImageElement(0, addResult.EndPageBounds.Bottom + 10, 100, logoImagePath);
    addResult = firstPage.AddElement(logoElement);

    //get the #PKCS 12 certificate from file
    DigitalCertificatesCollection certificates = DigitalCertificatesStore.GetCertificates(certificateFilePath, "evopdf");
    DigitalCertificate certificate = certificates[0];

    // create the digital signature over the logo image element
    DigitalSignatureElement signature = new DigitalSignatureElement(addResult.EndPageBounds, certificate);
    signature.Reason = "Protect the document from unwanted changes";
    signature.ContactInfo = "The contact email is support@evopdf.com";
    signature.Location = "Development server";
    firstPage.AddElement(signature);

    string outFilePath = System.IO.Path.Combine(Application.StartupPath, "DigitalSignature.pdf");

    // save the PDF document to disk
    document.Save(outFilePath);

    // close the PDF document to release the resources
    document.Close();

    DialogResult dr = MessageBox.Show("Open the saved file in an external
        viewer?", "Open Rendered File", MessageBoxButtons.YesNo);
    if (dr == DialogResult.Yes)
    {
        try
        {
            System.Diagnostics.Process.Start(outFilePath);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            return;
        }
    }
}
```

6.5.2.2 Other Interactive Page Elements

The HTML to PDF Converter also defines the following interactive elements that can be added to a page: *LinkUrlElement*, *InternalLinkElement*, *FileLinkElement*, *FileAttachmentElement*, *TextNoteElement* that can be used to add interactive features to the PDF document.

6.6 Bookmarks

The HTML to PDF Converter offers an API for adding bookmarks to a PDF document. Using the *AddBookmark()* methods of the *Document* class you can add root or child bookmarks. The code sample shows how to do this:

```
        // add a page to the PDF document
        PdfPage firstPage = document.AddPage();

        //add a root bookmark to the first page
        Bookmark firstPageBookmark = document.AddBookmark("First
                                        Page", new ExplicitDestination(firstPage));

        // add a child bookmark
        document.AddBookmark("HTML to PDF Converter", new ExplicitDestination(firstPage, new System.Drawing.PointF(0,0)),
                          firstPageBookmark);

        // add a child bookmark
        document.AddBookmark("HTML to Img Converter", new ExplicitDestination(firstPage, new System.Drawing.PointF(0, 200)),
                          firstPageBookmark);

        //add a new page to document
        PdfPage secondPage = document.AddPage();

        //add a root bookmark to the second page
                                        //and set a 150% zoom when visiting this bookmark
        ExplicitDestination secondPageDestination = new ExplicitDestination(secondPage);
        secondPageDestination.ZoomPercentage = 150;
        Bookmark secondPageBookmark = document.AddBookmark("Second
                                        Page", secondPageDestination);
```

7. Licensing

A unique license key string is generated for each purchase. In order to unlock the EVO HTML to PDF Converter product you have to set the ***LicenseKey*** property of the PdfConverter class (when converting to PDF) or of the ImgConverter class (when converting to image) with the license key string you have received after the product purchase.

The license key contains the information about the purchased product like the product version and license type and is uniquely associated with an order ID. More details about the license types and pricing can be found on the Buy Now page of our website.