

ExpertPDF HTML to PDF Converter for .NET

Developer's Manual



Table of Contents

1. Introduction

2. Installation

3. Requirements

4. Converter API

4.1 PdfConverter Class

4.1.1 PdfConverter Render Methods

4.1.2 PdfConverter Configuration Properties

4.2 ImgConverter Class

4.2.1 ImgConverter Render Methods

4.2.2 ImgConverter Configuration Properties

5. Features

5.1 Headers and Footers

5.2 Security Options

5.3 Document Description

5.4 Automatic and Custom Page Breaks, Keep Together

5.5 Live HTTP Links

5.6 Merge Capabilities

5.7 Enable/Disable Client Scripts and ActiveX from HTML Page

5.8 Server Authentication

5.9 Custom PDF Page Size

5.10 Bookmarks

5.11 Internal Links in PDF

5.12 Jpeg Compression of Images in PDF

5.13 Retrieve HTML Elements Mapping to PDF

5.14 PostConvertActionEvent

5.15 Select PDF Standard

5.16 Select Color Space

6. Advanced Features with PostConvertActionEvent

6.1 Overview

6.2 Coordinates System and Graphic Units

6.3 Document Class

6.4 PDF Renderers

6.4.1 PdfPage Class

6.4.2 Template Class

6.4.2.1 Predefined Templates

6.4.2.2 Custom Templates

6.5 PDF Page Elements

6.5.1 Page Graphic Elements

6.5.1.1 HTML to PDF Converter Element

6.5.1.1.1 Page Breaks, Keep Together

6.5.1.1.2 Live HTTP Links

6.5.1.1.3 Enable/Disable Client Scripts and ActiveX/Flash from HTML Page

6.5.1.1.4 Server Authentication

6.5.1.1.5 Bookmarks

6.5.1.1.6 Internal Links in PDF

6.5.1.1.7 Retrieve HTML Elements Mapping to PDF

6.5.1.2 HTML to Image Converter Element

6.5.1.3 RTF to PDF Converter Element

6.5.1.4 Text Element And Fonts

6.5.1.5 Image Element

6.5.1.6 Shape Elements

6.5.2 Page Interactive Elements

6.5.2.1 Digital Signature Element

6.5.2.2 Other Interactive Page Elements

6.6 Bookmarks

7. Licensing

1. Introduction

The ExpertPDF HTML to PDF Converter for .NET 2.0 and .NET 3.5 consists in a .NET library that can be used directly in any .NET application (ASP.NET, Windows Forms, Console, Web Services, Windows Services, etc) and a ASP.NET 2.0 and ASP.NET 3.5 server control built as wrapper of the library to help you benefit from the design time capabilities of the Microsoft Visual Studio 2005 and 2008.

The converter does not require any installation and it does not use any printer driver to perform conversion. It's just an assembly that you can directly link with your .NET application. 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 general purpose tool for converting web pages and HTML code to PDF or as part of our Reporting Toolkit for .NET to easily create PDF reports directly from ASP.NET pages. If you think that the converted ASP.NET page can contain your preferred server controls like charts, barcodes, data bound control like data grids and repeaters you can realize how powerful this tool can be.

The converter API offers methods to convert a web page from a specified URL to PDF or a specified HTML string. Additionally you can convert web pages and HTML code **to images in any format supported by .NET 2.0 and .NET 3.5 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 ***ephhtmltopdf.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:

```
1: using ExpertPdf; HtmlToPdf;
2: 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 ExpertPDF HTML to PDF Converter for .NET 2.0 and .NET 3.5 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 2.0/3.5 assemblies you can use in your application.

ephhtmltopdf.dll - is the HTML to PDF converter library that you can link in any .NET application, either Windows Forms or ASP.NET.

ephhtmltopdf.xml - is the HTML to PDF converter API reference in XML format for Visual Studio IntelliSense.

2.1 Doc Folder

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

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

Manual.pdf and *Manual.html* - contain this manual in PDF and HTML formats

2.1 Samples Folder

Samples folder contains C# and VB.NET full sample applications to offer you ready to use code for ASP.NET, Windows Forms and console applications. Each sample has a solution file .sln that you can directly open in Visual Studio 2005 or in Visual Studio 2008.

AspNet_GettingStarted - is a ASP.NET 2.0/3.5 application which shows how to convert web pages and HTML code to PDF and images. The application uses the HTML to PDF Converter library with the default settings.

AspNet_InvoicesDemo - is a ASP.NET 2.0/3.5 application which shows how to dynamically generate PDF invoices from a ASP.NET page. The application uses the HTML to PDF Converter library to convert a HTML string to PDF.

AspNet_HtmlConvertDemo - is a ASP.NET 2.0/3.5 application which shows how to convert web pages and HTML code to PDF and images. The application uses the HTML to PDF Converter library and shows you how to set various conversion parameters like the headers and footers, page size, page orientation, compression level, etc.

WinForms_HtmlConverterFeaturesDemo - is a Windows Forms application which shows how to convert web pages and HTML code to PDF and images. The application uses the HTML to PDF Converter library and shows you how to set various conversion parameters like the headers and footers, page size, page orientation, compression level, etc.

WinForms_NavigateAndConvertDemo - is a Windows Forms application which shows how to convert web pages and HTML code to PDF while you are navigating in Internet. The application uses the HTML to PDF Converter library. Using this application it is possible to convert pages from websites requiring authentication at application level implemented with a custom login page.

Console_HtmlConvertDemo_CS - is a Console application which shows how to convert web pages and HTML code to PDF and images. The application uses the HTML to PDF Converter library and shows you how to set various conversion parameters like the headers and footers, page size, page orientation, compression

level, etc.

Console_BatchConversion_CS - is a Console application which shows how to use the converter to convert to PDF documents all the HTML files from a specified folder and its subfolders

WinForms_HtmlInHeaderAndFooter - is a Windows Forms application which shows how to add HTML in header and footer of the rendered PDF document. Here you can also find code to add page numbers in footer.

WinForms_PrependAppendExternalPdfs - is a C# Windows Forms application which shows how to convert many URLs into the same PDF document.

3. Requirements and Recommendations

The recommended hardware and software resources for successfully running the ExpertPDF 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

Hardware Architecture: 32-bit, 64-bit

Free RAM: 1GB

Microsoft .NET Framework 2.0 or 3.5

Full trust level when used in ASP.NET applications

4. Converter API

The converter API is fully documented in the *HtmlToPdf.chm*. In order to use the converter library you have to include the *ExpertPdf.HtmlToPdf* 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 Render Methods

The method below retrieves the PDF bytes from a URL. 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[] GetPdfFromUrlBytes ( 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 a hint for the converter which is used 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)
```

In the full API reference document you'll notice some other similar methods for converting a HTML stream to PDF or a HTML file to file but they are derived from the methods described above and in the most of the cases you won't need them.

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 *PageWidth* and *PageHeight* properties allows you to set the width and height of the virtual browser windows. The web page content is rendered based on the virtual browser width specified as a 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 *PageWidth* property is 1024 pixels. The default value of the *PageHeight* property is 0 pixels which means the height will be automatically determined. 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. You can also choose to let the converter autodetermine both the width and height of the virtual browser by setting both *PageWidth* and *Page Height* properties to 0.

```
public int PageWidth { get; set; }
public int PageHeight { 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 a image 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 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 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[] GetImageFromUrlBytes (string url, ImageFormat format)
public Image GetImageFromUrl (string url, ImageFormat format)
```

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 a hint for the converter which is used 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 and in the most of the cases you won't need them.

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 *PageWidth* and *PageHeight* properties allows you to set the width and height of the virtual browser windows. The web page content is rendered based on the virtual browser width specified as a 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 *PageWidth* property is 1024 pixels. The default value of the *PageHeight* property is 0 pixels which means the height will be automatically determined. 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. You can also choose to let the converter auto determine both the width and height of the virtual browser by setting both *PageWidth* and *PageHeight* properties to 0.

```
public int PageWidth { get; set; }
public int PageHeight { get; set; }
```

5. Features

In this section will be described the main features of the converter and 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.

For the header you can set a title and a subtitle, add a image in a specified position and draw a horizontal line under the header, change the text font and size, change the background color of the header. Below you can see a sample code to set the header options:

```
pdfConverter.PdfHeaderOptions.HeaderBackColor = Color.WhiteSmoke;
pdfConverter.PdfHeaderOptions.HeaderHeight = 50;
pdfConverter.PdfHeaderOptions.HeaderText =
    "Title";
pdfConverter.PdfHeaderOptions.HeaderTextColor = Color.Black;
pdfConverter.PdfHeaderOptions.HeaderTextFontType = PdfFontType.Helvetica;
pdfConverter.PdfHeaderOptions.HeaderTextFontSize = 18;
pdfConverter.PdfHeaderOptions.HeaderTextYLocation = 5;
pdfConverter.PdfHeaderOptions.HeaderSubtitleText =
    "Subtitle";
pdfConverter.PdfHeaderOptions.HeaderSubtitleTextColor = Color.Black;
pdfConverter.PdfHeaderOptions.HeaderSubtitleTextFontType = PdfFontType.Helvetica;
pdfConverter.PdfHeaderOptions.HeaderSubtitleTextFontSize = 12;
pdfConverter.PdfHeaderOptions.HeaderTitleSubtitleYSpacing = 7;
pdfConverter.PdfHeaderOptions.HeaderImageLocation = new PointF(0, 0);
pdfConverter.PdfHeaderOptions.HeaderImage = Image.FromFile(LogoImageFullPath);
pdfConverter.PdfHeaderOptions.DrawHeaderLine = true;
```

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.

For the footer you can set the text, to show or not the page numbering, the text that appears before the page number, the font text and color, the footer background color and to draw or not a line above the footer. Below you can see a sample code to set the footer options:

```
pdfConverter.PdfFooterOptions.FooterText =
    "Footer text";
pdfConverter.PdfFooterOptions.FooterBackColor = Color.WhiteSmoke;
pdfConverter.PdfFooterOptions.FooterHeight = 40;
pdfConverter.PdfFooterOptions.FooterTextColor = Color.Black;
pdfConverter.PdfFooterOptions.FooterTextFontType = PdfFontType.HelveticaOblique;
pdfConverter.PdfFooterOptions.FooterTextFontSize = 8;
pdfConverter.PdfFooterOptions.DrawFooterLine = true;
pdfConverter.PdfFooterOptions.PageNumberText = "Page";
pdfConverter.PdfFooterOptions.PageNumberTextColor = Color.Black;
pdfConverter.PdfFooterOptions.PageNumberTextFontType = PdfFontType.HelveticaBold;
pdfConverter.PdfFooterOptions.PageNumberTextFontSize = 10;
pdfConverter.PdfFooterOptions.ShowPageNumber = true;
```

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.

Starting with the version 3.5 of the converter you have the possibility to add HTML in header footer. This offers maximum flexibility when designing the header and footer of the rendered PDF document. For a complete sample of adding HTML in header and footer please take a look at the WinForms_HeaderAndFooterHtml sample application.

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 set a user password the PDF document is encrypted and that password will be asked by the PDF viewer in order to open the PDF document. When you set the owner password that password will be required when someone wants to change the PDF permissions. 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 = "ExpertPDF";
pdfConverter.PdfSecurityOptions.OwnerPassword = "";
```

5.3 Document Description

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

```
pdfConverter.PdfDocumentInfo.AuthorName =
    "ExpertPDF";
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>
```



```

        
    </td>
    <td>
        My text 1
    </td>
</tr>

<tr style="page-break-inside: avoid">
    <td>
        
    </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. This possibility is available with the AppendPDFFile and AppendPDFFileArray properties from PdfDocumentOptions class. The properties must be set before calling the PDF render method. There also available similar properties to append PDF streams instead of files. The prototypes of these properties are:

```

public string AppendPDFFile { get; set; }
public string[] AppendPDFFileArray { get; set; }
public Stream AppendPDFStream { get; set; }
public Stream[] AppendPDFStreamArray { get; set; }

```

For more details please take a look at the WinForms_ConvertAndMergePdf sample application.

5.7 Enable/Disable Client Scripts and ActiveX from HTML Page

The JavaScript code and ActiveX controls are disabled by default in the converted page during conversion to a PDF with selectable texts and objects and enabled when converting to image. If you have JavaScript code that modifies the web page on the client you can instruct the converter to execute that JavaScript code. You can activate scripts both when rendering an image or a PDF document. The properties from PdfConverter class which allow you to activate the scripts when converting to PDF file are:

```

public bool ScriptsEnabled { get; set; }
public bool ScriptsEnabledInImage { get; set; }

```

5.8 Server Authentication

The converter offers support for any type of server 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. 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 properties of *ImgConverter* class which allow you to handle the authentication when converting HTML to images are:

```

public string AuthenticationPassword { get; set; }
public string AuthenticationUsername { get; set; }

```

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 Custom. In this case the PDF page size will be given by the **PdfConverter.PdfDocumentOptions.CustomPdfPageSize** property. Below is a sample code for setting the converter to produce PDF pages with the width of 200 points and height of 300 points. A point is 1/72 inch.

```

pdfConverter.PdfDocumentOptions.CustomPdfPageSize =
    new SizeF(200, 300);

```

By default the custom size is set to a width of 595 and a height of 842 points which is the size of the A4 portrait page. When the page orientation is set to landscape the width and height values are inverted.

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 tag names is specified by the `pdfConverter.PdfBookmarkOptions.TagNames` property. For example, to enable bookmarking of the H1 and H2 tags you can use the following line of C# code:

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

The tags to be bookmarked can be further filtered by CSS class name using the `pdfConverter.PdfBookmarkOptions.ClassNameFilter` property. For example, to filter only the H1 and H2 tags having the CSS class `bookmark`, the following line of C# can be added to the previous one:

```
pdfConverter.PdfBookmarkOptions.ClassNameFilter =  
    "bookmark";
```

The `ClassNameFilter` property is case sensitive and the string value set for this property must textually match the class attribute of the HTML tag to be bookmarked.

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 `Internal Link` tag and the target of the link defined by a `Link Target` 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.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. The PdfConverter has a property `InternalLinksDocUrl` which allows you to specify the fully qualified URL referenced by the internal links before calling the converter method.

When converting a HTML string to PDF it is recommended to always pass the `baseUrl` and `internalLinksDocUrl` parameters to the method used to convert the HTML string to PDF.

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 JPEG compression 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 possible to completely disable the JPEG compression of the images by setting the `PdfConverter.PdfDocumentOptions.JpegCompressionEnabled` to `false`.

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 be accessed using the `PdfConverter.HtmlElementsMappingOptions` property. This property allows you to define a list with HTML IDs of the HTML elements for which you want to retrieve position using the `HtmlElementIds` property or a list with the HTML tag names of the HTML elements for which you want to retrieve position using the `HtmlTagNames` property.

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

The HTML elements mapping is returned in the `PdfConverter.HtmlElementsMappingOptions.HtmlElementsMappingResult`. The `HtmlElementsMappingResult` 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 Features with PostConvertActionEvent - Overview](#) section.

5.14 PostConvertActionEvent Event

The PostConvertActionEvent event is raised by the PdfConverter object right before the PDF document is saved. The argument of the handler of this event offers a reference to the PdfConverter object, a reference to the Document object created by the converter and a reference to a ConversionSummary object. The most important object is the Document object defined in the ExpertPdf.HtmlToPdf.PdfDocument namespace which allows an advanced customization of the PDF document. A detailed description of the possibilities opened by this event can be found in [Advanced Features with PostConvertActionEvent](#) section.

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

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 instructed to generate PDF documents in conformance with PDF/A, PDF/X and PDF/SiQQ and 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.

PDF/X-1a:2003 standard (ISO 15930-4), used to facilitate graphics exchange, requires that all the true type fonts used by the document to be embedded in the document, the http links are disabled, all the graphics are in CMYK color space, the document does not use transparent objects.

PDF/SiQQ is a standard used by Adobe to ensure that a digitally signed document does not contain items that could alter its appearance when viewed in different environments. A PDF/SiQQ compliant document requires that all the true type fonts used by the document to be embedded in the document and the http links to be 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.

6. Advanced Features with PostConvertActionEvent

6.1 PostConvertActionEvent Overview

The PostConvertActionEvent event is raised by the PdfConverter object right before the PDF document is saved. The argument of the handler of this event offers a reference to the PdfConverter object, a reference to the Document object created by the converter and a reference to a ConversionSummary object.

The most important object is the Document object defined in the ExpertPdf.HtmlToPdf.PdfDocument namespace which allows an advanced customization of the PDF document. When the Document object is passed to the the PostConvertActionEvent even handler it already contains the PDF pages generated by the converter from the URL or 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 section 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, a RtfToPdfElement which embeds a RTF formatted text 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, all the *H1* and *IMG* elements and the elements with the ID *MyId1* and *MyId2* will be highlighted with a red rectangle in the generated PDF:

```
using ExpertPdf.HtmlToPdf;
using ExpertPdf.HtmlToPdf.PdfDocument;

private void btnConvert_Click(object sender, EventArgs e)
{
    // create the PdfConverter object
    PdfConverter pdfConverter = GetPDFConverter();

    // define the list with the HTML tags of the elements for which you
    // want to retrieve position
    pdfConverter.HtmlElementsMappingOptions.HtmlTagNames = new string[] { "H1", "IMG" };
    // define the list with the HTML IDs of the elements for which you
    // want to retrieve position
    pdfConverter.HtmlElementsMappingOptions.HtmlElementIds = new string[] { "MyId1", "MyId2" };

    // register as listener for the PostConvertActionEvent event
    pdfConverter.PostConvertActionEvent += new PostConvertActionHandler(pdfConverter_PostConvertActionEvent);

    // call the converter method to obtain the PDF document bytes and
    // write the
    // bytes into a file on disk
    string outFile = Path.Combine(Application.StartupPath, "RenderedPage.pdf");
    try
    {
        byte[] pdfBytes = pdfConverter.GetPdfBytesFromUrl("http://www.html-to-pdf.net");
        File.WriteAllBytes(outFile, pdfBytes);
    }
    finally
    {
    }
```

```

{
    // unregister from the PostConvertActionEvent event
    pdfConverter.PostConvertActionEvent -= new PostConvertActionHandler(pdfConverter_PostConvertActionEvent);
}

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

/// <summary>
/// The PostConvertActionEvent event handler.
/// </summary>
/// <param name="pcaEventArgs">The handler argument sent by
/// the PdfConverter object.</param>
void pdfConverter_PostConvertActionEvent(PostConvertEventArgs pcaEventArgs)
{
    PdfConverter pdfConverter = pcaEventArgs.PdfConverter;
    Document document = pcaEventArgs.PdfDocument;
    ConversionSummary conversionSummary = pcaEventArgs.ConversionSummary;

    // iterate over the HTML elements mappings and draw a red rectangle
    around the element in PDF
    foreach (HtmlElementMapping elementMapping in pdfConverter.HtmlElementMappingOptions.HtmlElementMappingResult)
    {
        // iterate over the positions of the HTML element in PDF because a
        HTML element
        // can span on many PDF pages
        foreach (HtmlElementPdfRectangle pdfRectangle in elementMapping.PdfRectangles)
        {
            // get the PDF page where the HTML element was rendered
            PdfPage elementPdfPage = document.Pages[pdfRectangle.PageIndex];
            RectangleF elementPdfRectangle = pdfRectangle.Rectangle;

            // get the rectangle inside the PDF page where the element was rendered
            RectangleElement elementHeightRectangle = new RectangleElement(elementPdfRectangle.X, elementPdfRectangle.Y,
                elementPdfRectangle.Width, elementPdfRectangle.Height);
            elementHeightRectangle.ForeColor = Color.Red;

            elementPdfPage.AddElement(elementHeightRectangle);
        }
    }
}
}

```

6.2 Coordinates System and Graphic Units in PostConvertActionEvent

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 in the PostConvertActionEvent handler argument or new one can be created from an external PDF document. 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 from 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()* method or 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 overloads 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 in PostConvertActionEvent

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 in PostConvertActionEvent

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

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

string logoImagePath = System.IO.Path.Combine(Server.MapPath("~/"), @"img\logo.jpg");

// 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, logoImagePath);
AddElementResult addResult = firstPage.AddElement(imageElement1);

// add image border
// add a border to watermark
RectangleElement imageBorderRectangleElement =
    new RectangleElement(1, 1, addResult.EndPageBounds.Width,
        addResult.EndPageBounds.Height);
firstPage.AddElement(imageBorderRectangleElement);

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

// calculate the watermark location

System.Drawing.SizeF imageSizePx = logoImg.PhysicalDimension;

// transform from pixels to points
float imageWidthPoints = UnitsConverter.PixelToPoints(imageSizePx.Width);
float imageHeightPoints = UnitsConverter.PixelToPoints(imageSizePx.Height);

float watermarkXLocation = (firstPage.ClientRectangle.Width - imageWidthPoints)/2;
float watermarkYLocation = firstPage.ClientRectangle.Height / 4;

// add a template watermark to the document repeated
```

```

        on each document page
        // the watermark size is equal to image size
        in points
        Template watermarkTemplate = document.AddTemplate(new System.Drawing.RectangleF(watermarkXLocation, watermarkYLocation,
        imageWidthPoints, imageHeightPoints + 20));

        // add a standard font to the document
        PdfFont watermarkTextFont = document.AddFont(StdFontBaseFamily.HelveticaBold);
        watermarkTextFont.Size = 10;

        // Add a text element to the watermark. You
        can add any other graphic element to a template
        TextElement watermarkTextElement = new TextElement(3, 0, "This is Watermark Text", watermarkTextFont);
        watermarkTextElement.ForeColor = System.Drawing.Color.Red;
        watermarkTextElement.Transparency = 100;
        watermarkTemplate.AddElement(watermarkTextElement);

        // add an image to the watermark
        ImageElement watermarkImageElement = new ImageElement(0, 20, logoImg);
        watermarkImageElement.Transparency = 100;
        watermarkTemplate.AddElement(watermarkImageElement);

        // add a border to watermark
        RectangleElement watermarkRectangleElement =
        new RectangleElement(0, 0, watermarkTemplate.ClientRectangle.Width,
        watermarkTemplate.ClientRectangle.Height);
        watermarkTemplate.AddElement(watermarkRectangleElement);

        // dispose the image
        logoImg.Dispose();

```

6.5 PDF Page Elements in PostConvertActionEvent

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 in PostConvertActionEvent

The integrated HTML to PDF Converter is implemented by the *HtmToPdfElement* 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 HtmToPdfElement 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 HtmToPdfElement(float x, float y, float width, float height,
    string urlToConvert, int htmlViewerWidth, int htmlViewerHeight)

public HtmToPdfElement(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 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 width 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 parameter 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 parameter 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>
      
    </td>
    <td>
      My text 1
    </td>
  </tr>

  <tr style="page-break-inside: avoid">
    <td>
      
    </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 Client Scripts and ActiveX/Flash from HTML Page

The JavaScript code and ActiveX/Flash controls are disabled by default in the converted page during conversion to a PDF. If you have JavaScript code that modifies the web page on the client you can instruct the converter to execute that JavaScript code. You can activate the *JavaScript* code . The properties from *HtmlToPdfElement* class which allow you to activate the scripts and ActiveX/Flash when converting to PDF file are:

```
public bool ScriptsEnabled { get; set; }
public bool ActiveXEnabled { get; set; }
```

6.5.1.1.4 Server Authentication in HtmlToPdfElement

The converter offers support for any type of web server 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. 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 converter 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 tag names is specified by the ***HtmlToPdfElement.PdfBookmarkOptions.TagNames*** property. For example, to enable bookmarking of the H1 and H2 tags you can use the following line of C# code:

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

The tags to be bookmarked can be further filtered by CSS class name using the ***HtmlToPdfElement.PdfBookmarkOptions.ClassNameFilter*** property. For example, to filter only the H1 and H2 tags having the CSS class *bookmark*, the following line of C# can be added to the previous one:

```
htmlToPdfElement.PdfBookmarkOptions.ClassNameFilter = "bookmark";
```

The *ClassNameFilter* property is case sensitive and the string value set for this property must textually match the class attribute of the HTML tag to be bookmarked.

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 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 ***Internal Link*** tag and the target of the link defined by a ***Link Target*** 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. The *HtmlToPdfElement* has an optional *internalLinksDocUrl* parameter in constructor which allows you to specify the fully qualified URL referenced by the internal links before adding the *HtmlToPdfElement* to a PDF page.

When converting a HTML string to PDF it is recommended to always use the *htmlStringBaseUrl* and *internalLinksDocUrl* parameters when the *HtmlToPdfElement* is constructed in order to correctly get the internal links in the generated PDF document.

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 be accessed using the *HtmlToPdfElement.HtmlElementsMappingOptions* property. This property allows you to define a list with HTML IDs of the HTML elements to for which you want to retrieve position using the *HtmlElementIds* property or a list with the HTML tag names of the HTML elements for which you want to retrieve position using the *HtmlTagNames* property.

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 *HtmlElementsMappingResult* 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, all the *H1* and *IMG* elements and the elements with the ID *MyID1* and *MyID2* will be highlighted with a red rectangle in the generated PDF:

```
// create the HtmlToPdfElement
htmlToPdfElement = new HtmlToPdfElement("http://www.html-to-pdf.net");
// define the list with the HTML tags of the elements for which you
// want to retrieve position
htmlToPdfElement.HtmlElementsMappingOptions.HtmlTagNames = new string[] { "IMG", "H1" };
// define the list with the HTML IDs of the elements for which you
```



```

        want to retrieve position
htmlToPdfElement.HtmlElementsMappingOptions.HtmlElementIds = new string[] { "MyID1", "MyID2" };

// add the element to a PDF page
page.AddElement(htmlToPdfElement);

// iterate over the HTML elements mappings and draw a red rectangle
// around the element in PDF
foreach (HtmlElementMapping elementMapping in htmlToPdfElement.HtmlElementsMappingOptions.HtmlElementsMappingResult)
{
    // iterate over the positions of the HTML element in PDF because a
    // HTML element
    // can span on many PDF pages
    foreach (HtmlElementPdfRectangle pdfRectangle in elementMapping.PdfRectangles)
    {
        // get the PDF page where the HTML element was rendered
        PdfPage elementPdfPage = document.Pages[pdfRectangle.PageIndex];
        RectangleF elementPdfRectangle = pdfRectangle.Rectangle;

        // get the rectangle inside the PDF page where the element was rendered
        Rectangle elementHeightRectangle = new Rectangle(elementPdfRectangle.X, elementPdfRectangle.Y,
            elementPdfRectangle.Width, elementPdfRectangle.Height);
        elementHeightRectangle.ForeColor = Color.Red;

        elementPdfPage.AddElement(elementHeightRectangle);
    }
}

```

6.5.1.2 HTML to Image Converter Element in PostConvertActionEvent

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 width 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 parameter 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 parameter 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 RTF to PDF Converter Element in PostConvertActionEvent

The integrated RTF to PDF Converter is implemented by the *RtfToPdfElement* graphic element. It basically includes the functionality of the RTF to PDF Converter product and additionally offers the possibility to specify the position and the size of the rendered content and the possibility to add many RTF 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 *RtfToPdfElement* offers many constructors that basically calls the following constructor with more or less default values

```
public RtfToPdfElement(float x, float y, float width, float height,
    string rtfStringToConvert, int rtfViewerWidth, int rtfViewerHeight)
```

The constructor creates a RTF to PDF Converter element at the specified *x* and *y* coordinates with the specified *width* and *height*. The virtual viewer width and height in pixels are specified by the *rtfViewerWidth* and *rtfViewerHeight* 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 width 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.

rtfStringToConvert - The RTF string to convert to PDF

rtfViewerWidth - The RTF viewer width in pixels. The default value is 800 pixels. The effect of this parameter is similar with viewing the RTF viewer window with the specified width. When this parameter is negative, the converter will try to auto-determine the RTF document width.

rtfViewerHeight - The RTF viewerheight 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 RTF document in a viewer window with the specified width and height. When this parameter is negative, the converter will try to auto-determine the RTF document height.

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

```
PdfPage page = document.Pages.AddNewPage(PageSize.A4, new Margins(10, 10, 0, 0), PageOrientation.Portrait);
```

```
// the code below can be used to create a
// page with default settings A4, document margins inherited, portrait orientation
//PdfPage page = document.Pages.AddNewPage();
```

```
// add a font to the document that can be used for the
// texts elements
```

```
PdfFont font = document.Fonts.Add(new Font(FontFamily("Times New Roman"), 10, GraphicsUnit.Point));
```

```
// add header and footer before rendering
// the content
```

```
if (cbAddHeader.Checked)
    AddHtmlHeader(document);
```

```
if (cbAddFooter.Checked)
    AddHtmlFooter(document, font);
```

```
// the result of adding an element to a PDF
// page
```

```
AddElementResult addResult;
```

```
// Get the specified location and size of the rendered
// content
```

```
// A negative value for width and height
```

```
// means to auto determine
```

```
// The auto determined width is the available width in
```

```
// the PDF page
```

```
// and the auto determined height is the
```

```
// height necessary to render all the content
```

```
float xLocation = float.Parse(textBoxXLocation.Text.Trim());
```

```
float yLocation = float.Parse(textBoxYLocation.Text.Trim());
```

```
float width = float.Parse(textBoxWidth.Text.Trim());
```

```
float height = float.Parse(textBoxHeight.Text.Trim());
```

```
// Create the RTF to PDF converter element
```

```
string rtfString = System.IO.File.ReadAllText(textBoxRtfFile.Text.Trim());
```

```
RtfToPdfElement rtfToPdfElement = new RtfToPdfElement(xLocation, yLocation, width, height, rtfString,
    RtfToPdfElement.DEFAULT_RTF_VIEWER_WIDTH_PX, -1);
```

```
//optional settings for the RTF to PDF converter
```

```
rtfToPdfElement.FitWidth = cbFitWidth.Checked;
```

```
rtfToPdfElement.EmbedFonts = cbEmbedFonts.Checked;
```

```
// add theHTML to PDF converter element to
// page
```

```
addResult = page.AddElement(rtfToPdfElement);
```

6.5.1.4 Text Element And Fonts in PostConvertActionEvent

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:

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

// Create a Times New Roman .NET font of 10 points
System.Drawing.Font ttffFont = 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 ttffFontItalic = 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 ttffFontBold = 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 ttffFontBoldItalic =
    new System.Drawing.Font("Times New Roman", 10,
        System.Drawing.FontStyle.Bold | System.Drawing.FontStyle.Italic, System.Drawing.GraphicsUnit.Point);

// Create a SimSun .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(ttffFont);
PdfFont newTimesFontItalic = document.AddFont(ttffFontItalic);
PdfFont newTimesFontBold = document.AddFont(ttffFontBold);
PdfFont newTimesFontBoldItalic = document.AddFont(ttffFontBoldItalic);

// Create the embedded PDF fonts based on the .NET true
    type fonts
PdfFont newTimesEmbeddedFont = document.AddFont(ttffFont, true);
PdfFont newTimesItalicEmbeddedFont = document.AddFont(ttffFontItalic, true);
PdfFont newTimesBoldEmbeddedFont = document.AddFont(ttffFontBold, true);
PdfFont newTimesBoldItalicEmbeddedFont = document.AddFont(ttffFontBoldItalic, 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);

// arabic right to left text
newTimesEmbeddedFont.Size = 15;
TextElement textElement7 = new TextElement(20, addResult.EndPageBounds.Bottom + 10, "مرحبا العالم!!!!", newTimesEmbeddedFont);
textElement7.RightToLeftLanguage = true;
addResult = firstPage.AddElement(textElement7);

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

//japanese text
TextElement textElementJapanese = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "こんにちは世界!", cjKEmbeddedFont);
addResult = firstPage.AddElement(textElementJapanese);

//chinese text
TextElement textElementChinese = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "你好世界! !", cjKEmbeddedFont);
addResult = firstPage.AddElement(textElementChinese);

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

// Hello World in Japanese
TextElement textElement10 = new TextElement(20, addResult.EndPageBounds.Bottom + 20, "こんにちは世界!", cjKJapaneseStandardFont);
addResult = firstPage.AddElement(textElement10);

// Hello World in Chinese Traditional
TextElement textElement11 = new TextElement(20, addResult.EndPageBounds.Bottom + 5, "你好世界! !", cjKChineseTraditionalStandardFont);
addResult = firstPage.AddElement(textElement11);

```

6.5.1.5 Image Element in PostConvertActionEvent

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 image 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 image 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);
```

6.5.1.6 Shape Elements in PostConvertActionEvent

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
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 in PostConvertActionEvent

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 in PostConvertActionEvent

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:

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

string logoImagePath = System.IO.Path.Combine(Server.MapPath("~/"), @"img\logo.jpg");
string certificateFilePath = System.IO.Path.Combine(Server.MapPath("~/"), "oss.pfx");

// 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, 0, logoImagePath);
AddElementResult addResult = firstPage.AddElement(logoElement);

//get the #PKCS 12 certificate from file
DigitalCertificatesCollection certificates = DigitalCertificatesStore.GetCertificates(certificateFilePath, "osspswd");
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 office@outsidesoftware.com";
signature.Location = "Development server";
firstPage.AddElement(signature);
```

6.5.2.2 Other Interactive Page Elements in PostConvertActionEvent

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

6.6 Bookmarks in PostConvertActionEvent

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 below is extracted from the *BookmarksDemo* sample application:

```
// 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));

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);

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

// display image in the available space in
// page and with a auto determined height to keep the aspect ratio
ImageElement imageElement2 = new ImageElement(0, 200, System.IO.Path.Combine(imagesPath, "rtftopdf-converter-250.jpg"));
addResult = firstPage.AddElement(imageElement2);

// add a child bookmark to this image
document.AddBookmark("RTF to PDF 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 200% zoom when visiting this bookmark
ExplicitDestination secondPageDestination = new ExplicitDestination(secondPage);
secondPageDestination.ZoomPercentage = 150;
Bookmark secondPageBookmark = document.AddBookmark("Second
Page", secondPageDestination);

// display image in the available space in page and with
// a auto determined height to keep the aspect ratio
ImageElement imageElement3 = new ImageElement(0, 0, System.IO.Path.Combine(imagesPath, "pdf-toolkit-pro-250.jpg"));
addResult = secondPage.AddElement(imageElement3);
```

7. Licensing

A unique license key string is generated for each purchase. In order to unlock the 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.