Installation manual and quick start guide.

Author: PerceiveIT Limited
Version: 0.8
Status: Beta Release
Date: 26 March 2013

http://www.scryber.co.uk

Congratualtions on downloading Scryber

Scryber has got to be the easiest and best way to generate dynamic pdf documents on the .NET platform.
You start by defining your document content in XML, add some styles to it, even bind to some external data, and once done render it to a file or out onto a stream with 2 lines of code.

It's now just as easy to create consistently styled documents as it is web pages, when the content changes, so can your documents.
Product catalogs, invoices, contracts, instruction manuals, articles and more. Everything can be generated as a document your users can download, save, add to their files or iPad reading list. This is not about printing this is about storing, preserving and reporting.

Scryber is Open Source under LGPL so you can include it in your own commercial applications / sites free of charge (See 'About the badge').
Scryber is also easy to extend, with full code support you can push in new components and alter the complete layout, before rendering.
And if that is not sufficient you can even write your own components, and include in the xml.

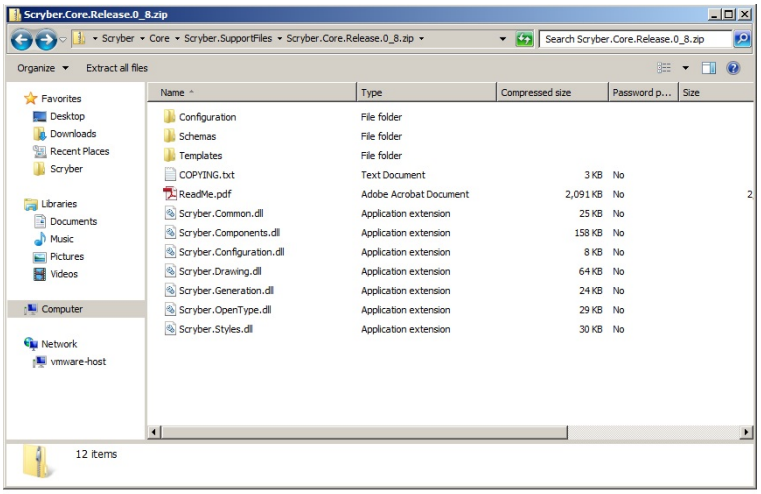
About this document

This document takes you through the installation of the Scryber libraries and getting you set up generating your first pdfs.
All the examples and screen shots are based around Visual Studio 2010, but Scryber is simply a bunch of libriaries, and you can use them on any .NET solution and development environment of your choice.
By the end of the guide you should be able to create new documents, add pages to the document, and add content to those pages with styles and layout options.
A complete set of guides is being created at http://www.scryber.co.uk and there is also a forum to ask all the questions you need.

Release status

Scryber is still, very definitely, in beta release. There are lots of things that may not give you the layout you are expecting, and you may also find bugs and errors occuring (although we hope there are not too many, and the messages are good).
Please be aware that if you do decide to take something to a fully loaded production platform - you may regret it, but if you get the results you want and it works for you, then enjoy!

Install files



You should have already extracted the zip archive that contains the installer, along with this guide.
If you haven't already done so, execute the installer
If you use Visual Studio 2010 - check the Visual Studio 2010 option,
If you use Visual Studio 2012 - check VS 2012
If you don't use VS then you do not need to choose either, but you will need to copy the templates to your solution folder when required.

Assemblies

There are 7 assemblies (dll's) that make up the core of Scryber, and all work together to generate PDF's. When you want to use Scryber in a project is it usually easier to just reference all the libraries.
They are fully signed and the installer will add them to the GAC by default.

All the assemblies are currently built against .NET 3.5. If you want to reference them in .NET 4+ then you may need to alter your application configuration to support dual mode execution.

Templates

The 'Templates' are added to *your* visual studio templates directory and the installation directory. They contain pre-formed xml documents that can be added to your projects quickly and easily. There are 4 in total each named appropriately.

Document.pdfx - a top level document
Page.ppfx - a page that can be referenced from any document
Component.pcfx - a single component that can be added to pages
Styles.psfx - an external styles collection that can be referenced from any document

Whilst the file extensions do not provide any extra capability currently, it is a convention that you should observe for future compatibility. The content of Document.pdfx is shown below - and this would render a perfect little 'Hello World'.

```
<?xml version="1.0" encoding="utf-8"?>
<pdf:Document auto-bind="true"
xmlns:pdf="Scryber.Components, Scryber.Components, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe"
xmlns:style="Scryber.Styles, Scryber.Styles, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe"
xmlns:data="Scryber.Data, Scryber.Components, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe" >
<Pages>
<pdf:Page id="MyFirstPage" >
<Content>
Hello World
</Content>
</pdf:Page>
</Pages>
</pdf:Document>
```

During this guide we will be using the templates exclusively to add new scryber components to the project.

Schema's

The XSDs are very important! There are 5 of them, and they work together to help you write structured documents, pages and components without knowing what fits where. All Xml editors should be able to read and understand these schemas so that when you try to add a component you can see where it should go.
This is a key feature of Scryber! The schemas are not just about intellisense, they also actually relate the xml to the components and their capabilities.
The namespace for the schemas defines the namespace for the libraries, and we will touch on this capability later on in the guide!

The visual studio installers add the files to the XML/Schemas directory so Visual Studio can find them, along with the installation path. The vanilla installer will only install the schemas in the installation path.
Now when you create a document, page, component or style sheet you will have intellisense to add the things you need where you need them.

Configuration

The Configuration directory of the instalation path contains a single .config file.
This is simply there as assistance to adding to your own project configuration file. If you want to alter the logging, include non-standard fonts, custom image types or your own components then you will need to edit the configuration.
We will not be using these sections for this guide, but there are other guides that cover the sections that you need to add.

Summary

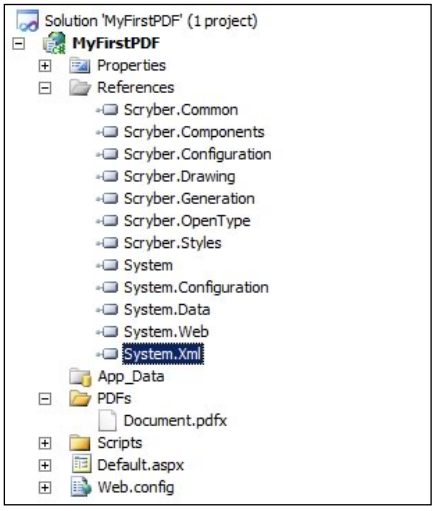
We should now have the dll's in a place where they can be easily added to projects, the templates accessible, and the schema files (XSDs) accessible to your development environment.

So we are now ready to move on to actually creating PDF documents.

My First PDF

Let's create our first pdf! If you are not using Visual Studio then it should be possible to follow along using your own knowledge or your development environment and specific passages in this section.
We are going to be doing this in a web application rather than a forms application, but the process is almost exactly the same, and we will highlight the changes at the end of the tutorial.

Create a new project



Let's start with a nice clean project, so from Visual Studio, close any existing solution and Add a New Project.
1. From the dialog choose the .NET Framework 3.5 from the top combo box. And then the web application project, and save it to a useful place. *This will create the project and a default.aspx page and a bunch of other stuff.*
2. Let's keep all our PDFs in a folder of their own, so select project first, then Project (or website) -> Add folder, and call it PDFs..
3. And finally lets add Document.pdfx to our folder with a right click on the folder and from the pop up menu choose Add New Item... Choose the Scryber category on the right and select the Document.pdfx item.

This will add a new document to our PDFs folder and add the required references.

Set up the page

What we need to do now is generate the actual pdf file based on the contents of Document.pdfx
Let's do this when a button is clicked on the web page. So.. Open the Default.aspx page in the Visual Studio html source editor (if it's not already). And add a button like below.

```
<asp:Button runat="server" Text="Click Me!" ID="GeneratePDF" OnClick="Page_GeneratePDF" />
```

We need to respond to the click event in our page code behind. So open Default.aspx.cs (.vb) and add a using for Scryber.Components, plus the handler below, after the page load method.
For C#....

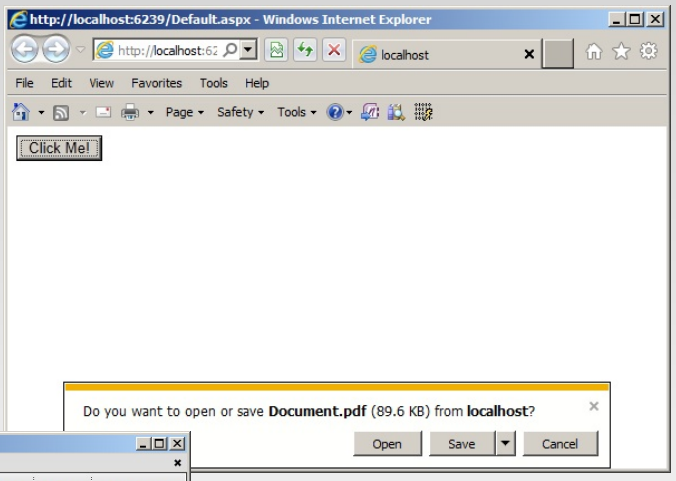
```
using Scryber.Components;
.
.
.
protected void Page_GeneratePDF(object sender, EventArgs e)
{
   PDFDocument doc = PDFDocument.ParseDocument("./App_Data/PDFs/Document.pdfx");
   doc.ProcessDocument(this.Response);
}
```

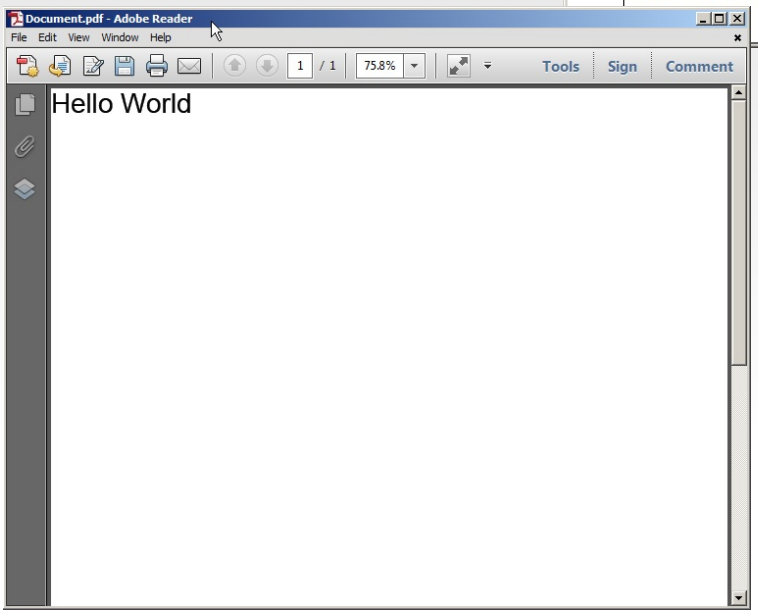
All the source code is available in **Appendix A** for this first version.

Hit the button

So all we do now is run our project.
Our page should open in a browser and we can click the button.
You should then be prompted to open Document.pdf in your favourite PDF Viewer.

Open it and 'Hello World' will appear.
There you have it, your first PDF document flying over the network to greet you in just 2 lines of code.





Adding some document content.

So that's OK, but we haven't actually created any content.
Rather than simply typing lines out or pushing in lorem ipsum, we will do some data binding with Scryber that shows off the dynamic capabilities of the libraries, as well as give us some valuable content.
So let's do some cool stuff!

Where's our data coming from.

The data can be any XML data structue - from a data set, or loaded from a file, but for this exmple the BBC produce a great rss feed for technology and it is also a good starting point for geting some generic information into the pdf document. http://feeds.bbci.co.uk/news/technology/rss.xml
The feed is a standard rss feed, but has all the good things that make this a relevant test.

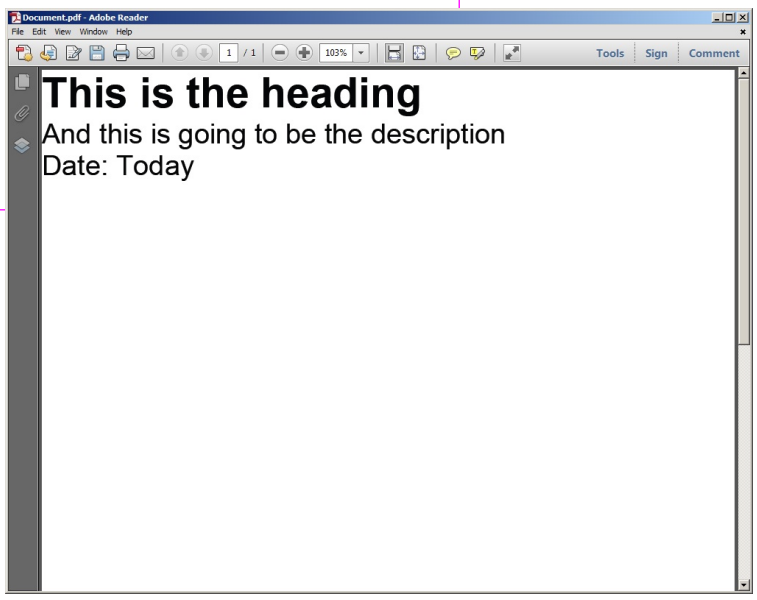
Dynamic uncontrolled input.
A reasonable amount of varied data
A standard structure.

If you want to look at the xml structure, follow the link above.

Layout some basic content

The first thing we need to do is put some features on the document. Scryber supports a simple set of components that can be built up into complex layouts similar to HTML. There are div's, spans, inline text, labels, panels, images, lines and drawings - all of which can be styled and positioned inline, as a block, relativly or even absolutely.
So for the content we are going to have a heading block with some textual elements in it. So remove the'Hello World' text and add the elements as below.

```
<pdf:Document xmlns:pdf="http://www.scryber.co.uk/schemas/v08/Scryber.Components.xsd"
              xmlns:style="http://www.scryber.co.uk/schemas/v08/Scryber.Styles.xsd"
              xmlns:data="http://www.scryber.co.uk/schemas/v08/Scryber.Data.xsd"
          auto-bind="true" >

  <Pages>
    <pdf:Page id="FirstPage" >
      <Content>

        <pdf:Div style:class="heading" >
          <pdf:H1 text="This is the heading" ></pdf:H1>
          <pdf:Label text="And this is going to be the description" ></pdf:Label>
          <BR/>
          Date: <pdf:Label text="Today" />
        </pdf:Div>

      </Content>
    </pdf:Page>


  </Pages>
</pdf:Document>
```



Reference the RSS feed

If you are unfamiliar with Xml or XPath then don't worry, you will still be able to follow along.
First thing is to add a new XML Data source into the Document.pdfx.

```
<Content>

  <data:XMLDataSource id="BBCRss"
        source-path="http://feeds.bbci.co.uk/news/technology/rss.xml" cache-duration="5" />

  <data:ForEach select="rss/channel" datasource-id="BBCRss" >
    <Template>
      <pdf:Div style:class="heading" >
        <pdf:H1 text="{xpath:title}"></pdf:H1>
        <pdf:Label text="{xpath:description}" /><BR/>
        Date: <pdf:Label text="{xpath:lastBuildDate}" /><BR/>
      </pdf:Div>
    </Template>
  </data:ForEach>

</Content>
```

We can then wrap a for each loop atound the heading Div that will enumerate over the one rss/channel element in the feed to generate the content. The text of the headings and labels are then changed to bind based on the {xpath:...} value for the attribute.
Hit your 'Click Me!' button again, as there is no need to rebuild, and you should generate a new pdf with some new *dynamic* content.

We can see the content being brought in from the RSS feed and flowing nicely around the page.
If your machine does not have direct access to the internet, download the feed and save it somewhere in your project. You can change the *source-path* in the Xml data source to a local file reference.

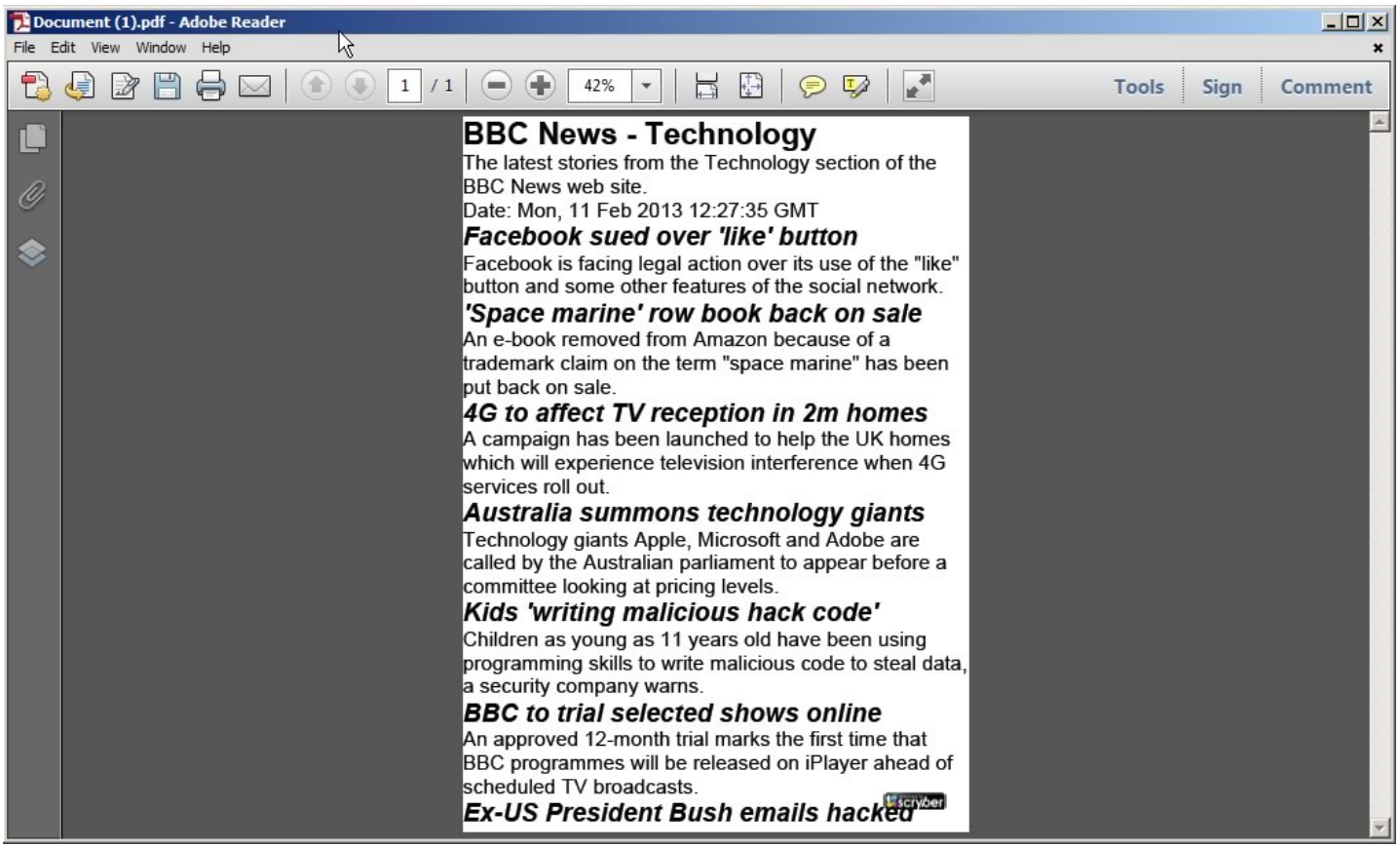
Bringing in the items

Top title is done, but we now need to include all the RSS feed 'items' into the page as well.
We can do this by simply including another for each loop inside the rss/channel as below. We do not need to specify the datasource as we have an existing data context.

```
<data:ForEach select="rss/channel" datasource-id="BBCRss" >
  <Template>
    <pdf:Div style:class="heading" >
      <pdf:H1 text="{xpath:title}"></pdf:H1>
      <pdf:Label text="{xpath:description}" /><BR/>
      Date: <pdf:Label text="{xpath:lastBuildDate}" />
    </pdf:Div>

    <pdf:Div style:class="content" >
      <data:ForEach select="item" >
        <Template>
          <pdf:Div style:class="rss-item" >
            <pdf:Div style:class="rss-item-data" >
              <pdf:H2 text="{xpath:title}" ></pdf:H2>
              <pdf:Label text="{xpath:description}" />
            </pdf:Div>
          </pdf:Div>

        </Template>
      </data:ForEach>
    </pdf:Div>

  </Template>
```

We are wrapping each item in 2 Div tags with style:class attributes on them, and the whole thing is wrapped in an outer div with another style class. *This will be used a little later on - don't forget to add it.*

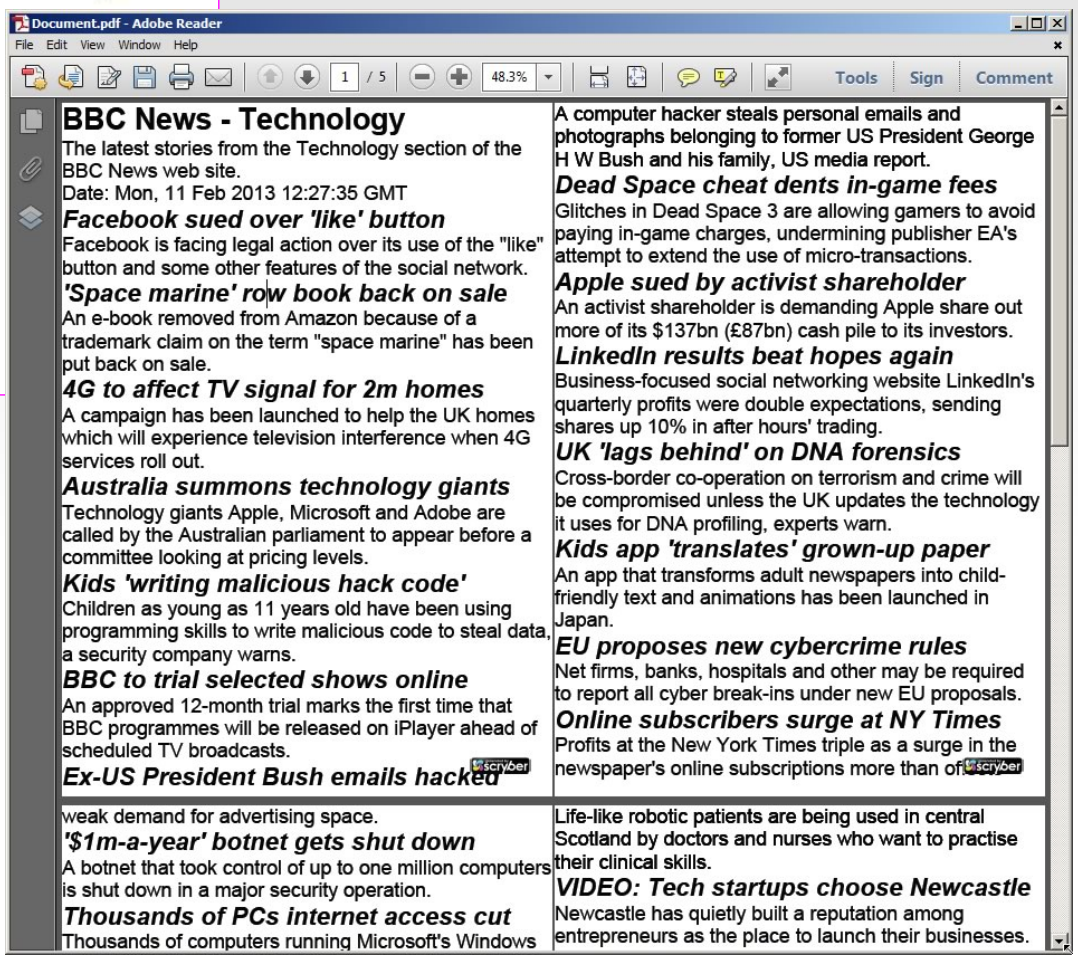
All being well, we should end up with a single page of content that flows down to the bottom of the page.



Overflow onto a new page

So why does it block? The `pdf:Page` component is set by default not to allow content to overflow, and therefore will only be a single rendered page.
There is however, a different component that does overflow `pdf:Section`. If we change the page component to a section component we should see a different result.
Change it now at the top and bottom and hit 'Click Me!'.

```
<Pages>
  <pdf:Section id="FirstPage" >
    <Content>

      .
      .
      .

    </Content>
  </pdf:Section>

</Pages>
```



A quick review

Great - we now have a PDF document that is reaching out for external content and this is binding with XPath into headings and labels and flowing nicely into a series of pages.

But, boy is it **UGLY!** Let's give this document some style.

Bring on the Style

Scryber has very comprehensive style support. All the information about how to render a page and it's components is driven from styles, so let's add some to this document.
Above the Pages element add a new 'Styles' element to the page. Within this element we can either add styles within the XML directly, or add one or more references to external style documents. The content is exactly the same.
Because we may want to use our styles on more than one PDF for a consistent look and feel, we'll create an external file.

Add the reference

Within your project, right click on the PDFs folder and choose 'Add Existing Item'. Navigate to the templates directory and choose the Styles.psfx.
Within the Document.pdfx Styles element add a Style-Ref element and point it to our new Styles file.

```
<?xml version="1.0" encoding="utf-8"?>
<style:Styles xmlns:pdf="http://www.scryber.co.uk/schemas/v08/Scryber.Components.xsd"
              xmlns:style="http://www.scryber.co.uk/schemas/v08/Scryber.Styles.xsd"
              xmlns:data="http://www.scryber.co.uk/schemas/v08/Scryber.Data.xsd">

  <style:Style applied-type="pdf:Page" >
    <style:Page size="A4" orientation="Portrait"/>
  </style:Style>

</style:Styles>
```

```
<pdf:Document xmlns:pdf="Scryber
              xmlns:style="Scryb
              xmlns:data="Scryber
         auto-bind="true" >
  <Styles>
    <style:Styles-Ref source="./Styles.psfx" />
  </Styles>
  <Pages>
    <pdf:Section id="MyFirstPage" >
      <Content>
```

Add the definitions

```
<style:Style applied-type="pdf:Page" >
  <style:Page size="A4" orientation="Portrait"/>
  <style:Margins left="20pt" bottom="40pt" right="20pt" top="15pt"/>
</style:Style>

<style:Style applied-class="heading" >
  <style:Border color="#00FFFF" width="1pt" sides="Top Left Right" />
  <style:Background color="#EEEEEE"/>
  <style:Padding all="10pt"/>
  <style:Margins top="20pt" bottom="20pt"/>
  <style:Position h-align="Center"/>
  <style:Font bold="true" size="12pt" />
</style:Style>
```

We can see that there is already a style declared in this file that will be applied to all components of type Page specifying a size and orientation.
So let's add to this style and give the page some well needed margins.
And while we're at it, we can add a new Style definition for the class 'heading' that our main title block in the document was declared with.
Hit the button again to see what difference this has made.

So What just happened!

When we hit the button, the Document.pdfx file was parsed. Within that file is a reference to this Styles.psfx file, so this was also parsed and included in the document.
When we came to process the document, each component had a full style built based upon these style declarations.

Because our pdf:Section is actually a page (PDFSection inherits from the PDFPage) then it had the pdf:Page style definition applied to it.
Our Div on the other hand has a 'heading' class declared on it, and all the attributes defined on the style were applied to this Div.
Some of these styles will also cascade down to inner components (e.g. the Font size and style).
Inside our heading Div we have text and labels that are now being rendered at 12 point bold.
The H1 component is actually defined with a base style that has a Font size and style set on it, so these 'heading' options were overridden, but the centre alignment was not overridden.

Styles are a complex subject and rely on a number of capabilities, but we hope this serves as a little introduction.

A note about units and color

Within the content of the style there are a number of attributes that accept measurement units and colors.
Units are real numbers optionally postfixed with a scale. The supported scales are..

Points (pt). The standard unit of measurement for PDF documents and the default unit. If no scale is provided then it will be assumed the measurement is in Points. 72pt = 1 inch.

Inches (in). The postfix must be 'in' e.g. 2.5in. A double quote notation is not supported.

Millimeters (mm). The postfix must be 'mm'. 25.4mm = 1 inch.

The pixel is not a supported scale as the layout is not represented on screen dimensions and the percentage notation is also not supported as the page size is a known dimension.
The style:Postion element does, however, support a boolean 'fill-width' attribute to push sizing to the container width.

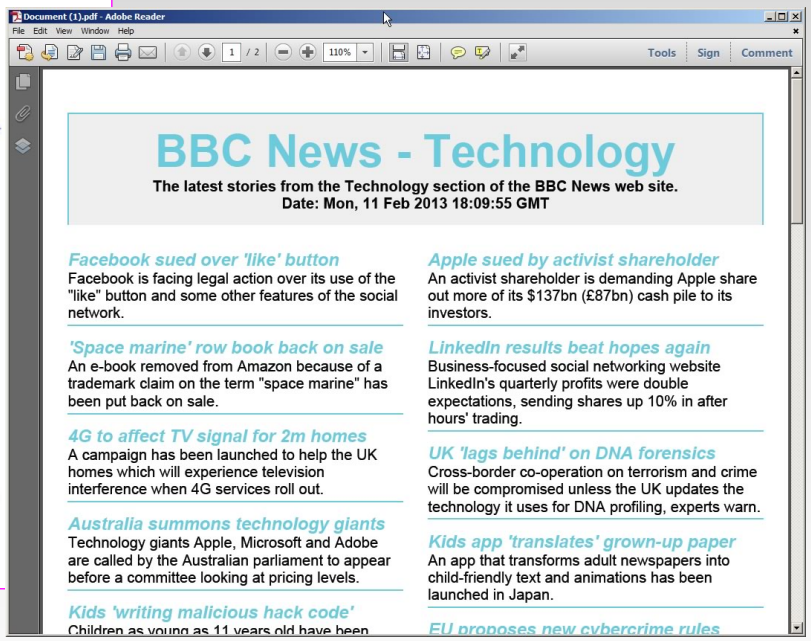
Colours can be identified as one of the known standard 16 colors, or by their hexadecimal notation.
If the notation is 2 characters it is assumed to be grayscale (#FF white, #77 gray, #00 black).
If the notation is 3 characters it is assumed to be RGB short (#FFF white, #F00 red, #000 black).
If the notation is 6 characters it is the full RGB notation (#FFFFFF white #FF0000 red, #000000 black).

Colors can also be defined in G, RGB or HSL decimal notation in the format G(255) or RGB(255,255,255) or HSL(0,0,100) which all denote white. All other formats will throw an exception.

Styling the items in the feed

So let's add a little style to the items that come back in the feed and see what we can do with this.

```
    <style:Padding all="10pt"/>
    <style:Margins top="20pt" bottom="20pt"/>
    <style:Position h-align="Center"/>
    <style:Font bold="true" size="12pt" />
  </style:Style>

  <style:Style applied-type="pdf:Div" applied-class="content" >
    <style:Columns count="2" alley-width="20pt"/>
    <style:Font size="12pt" bold="false" italic="false"  />
  </style:Style>

  <style:Style applied-type="pdf:H2" >
    <style:Font size="14pt" />
    <style:Fill color="#00FFFF"/>
  </style:Style>

  <style:Style applied-type="pdf:H1" >
    <style:Fill color="#00FFFF"/>
  </style:Style>

  <style:Style applied-class="rss-item" >
    <style:Margins bottom="10pt"/>
    <style:Padding bottom="3pt"/>
    <style:Border color="#00FFFF" sides="Bottom" width="1pt"/>
  </style:Style>

</style:Styles>
```



This is starting to look pretty cool. What say we add some images and links?

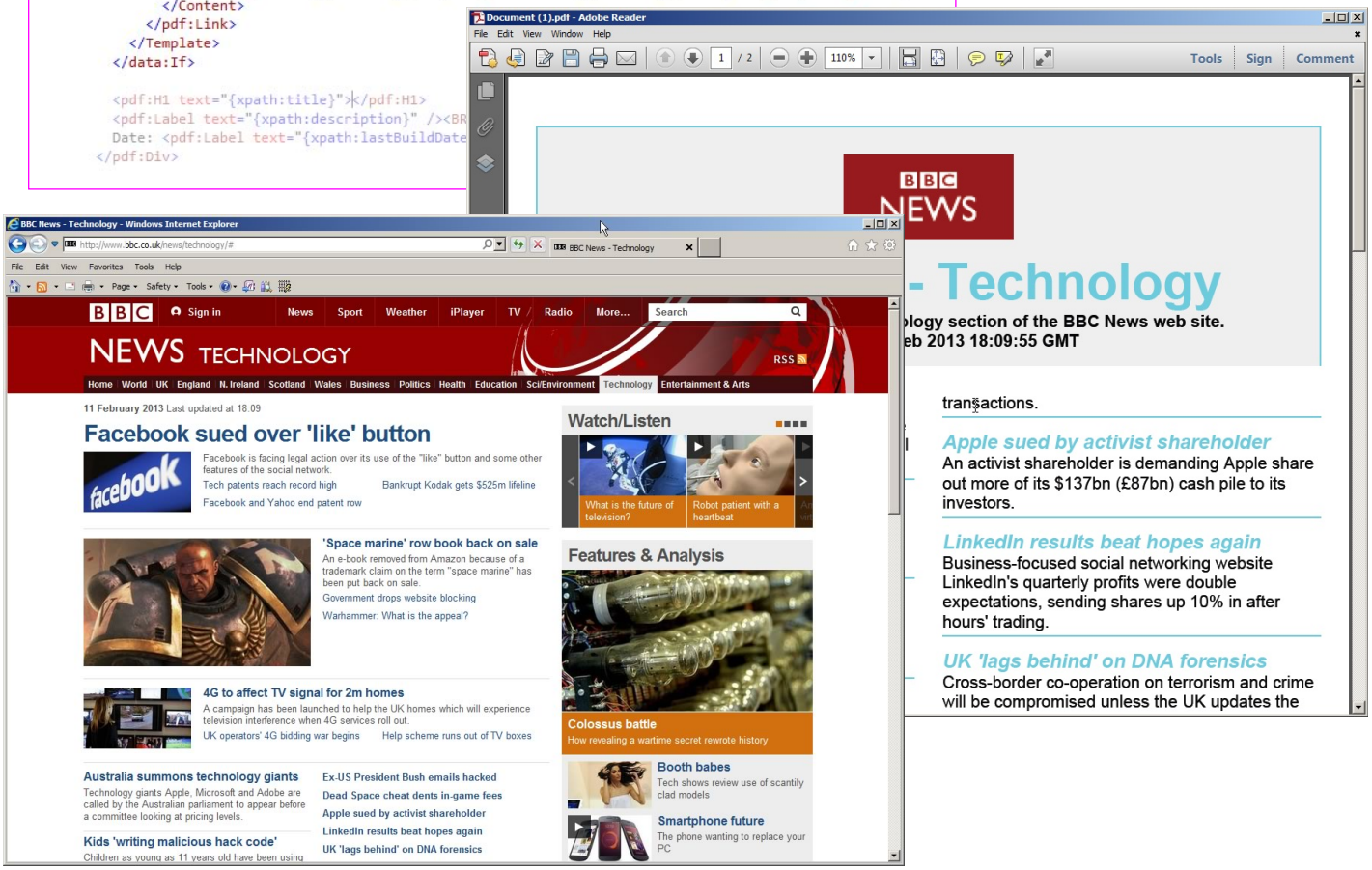
Add some images and links

Adding images is easy, and the BBC feed gives us a logo and an link to include in our feed under the image element of the channel.

```
<rss xmlns:media="http://search.yahoo.com/mrss/" xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
  <channel>
    <title>BBC News - Technology</title>
    <link>http://www.bbc.co.uk/news/technology/#sa-ns_mchannel=rss&amp;ns_source=PublicRSS20-sa</link>
    <description>The latest stories from the Technology section of the BBC News web site.</description>
    <language>en-gb</language>
    <lastBuildDate>Sat, 09 Feb 2013 03:04:48 GMT</lastBuildDate>
    <copyright>Copyright: (C) British Broadcasting Corporation, see http://news.bbc.co.uk/2/hi/help/rss/4498287.stm for terms and
conditions of reuse.</copyright>
    <image>
      <url>http://news.bbcimg.co.uk/nol/shared/img/bbc_news_120x60.gif</url>
      <title>BBC News - Technology</title>
      <link>http://www.bbc.co.uk/news/technology/#sa-ns_mchannel=rss&amp;ns_source=PublicRSS20-sa</link>
      <width>120</width>
      <height>60</height>
    </image>
    <ttl>15</ttl>
```

Back in our Document.pdfx in the heading Div, let's put in an If test to make sure that we have an image on this feed and then we can build a link to a Uri with an image as the content.
Hit the button one more time to generate the document and you should be able to click the image and it will take you to the technology news page.
Pretty Sweet, and we didn't even have to adjust any other components - it all just flowed!

```
<data:ForEach select="rss/channel" datasource-id="BBCRss" >
  <Template>
    <pdf:Div style:class="heading" >
      <data:If test="string-length(image/url) &gt; 0" >
        <Template>
          <pdf:Link action="Uri" file="{xpath:image/link}" new-window="true" alt="{xpath:image/title}" >
            <Content>
              <pdf:Image src="{xpath:image/url}" style:width="{xpath:image/width}" style:padding="10" />
            </Content>
          </pdf:Link>
        </Template>
      </data:If>

      <pdf:H1 text="{xpath:title}"></pdf:H1>
      <pdf:Label text="{xpath:description}" /><BR
      Date: <pdf:Label text="{xpath:lastBuildDate
    </pdf:Div>
```



Secure the files

This is good, and ready to show the client, but there are the more mundane tasks to undertake first.
At the moment the Document.pdfx and the Styles.psfx are actual files on the disk. Therefore they can be served by IIS and will just return to the client the xml content of the file.
We need to block this content from being served (Or provide a mechanism that will serve the generated content).
Back in the scryber installation directory there was a configuration directory, and within there a Scryber.config file.

Open this config file and copy the content within the system.Web/httpHandlers to your config file

```
<add path="*.psfx" verb="*" type="System.Web.HttpForbiddenHandler" />
<add path="*.ppfx" verb="*" type="System.Web.HttpForbiddenHandler" />
<add path="*.pcfx" verb="*" type="System.Web.HttpForbiddenHandler" />
<add path="*.pdfx" verb="*" type="System.Web.HttpForbiddenHandler" />
```

And copy the content within the system.webServer/handlers to your config file.

```
<add name="Scryber.Styles" path="*.psfx" verb="*" type="System.Web.HttpForbiddenHandler" />
<add name="Scryber.Components.Page" path="*.ppfx" verb="*" type="System.Web.HttpForbiddenHandler" />
<add name="Scryber.Components.UserComponent" path="*.pcfx" verb="*" type="System.Web.HttpForbiddenHandler" />
<add name="Scryber.Components.Document" path="*.pdfx" verb="*" type="System.Web.HttpForbiddenHandler"/>
```

If you want to be doubly safe the put <remove... /> elements above these in the config file. You may also need to make IIS pass requests through for these file extensions rather than handling them directly, depending on which version of IIS you are using and whether the application is set up as integrated pipeline.

Just a few small changes

It was all going so well and we showed the document to the client, and they were over the moon.
But there were just a few changes in the feedback...
1. That blue is corporate color, but doesn't fit well with the red of the BBC logo. Can we make the borders (the same) red.
2. The bottom border should also be on the heading.

Pretty minor all in all. Oh, and they also found out that a lot of their clients read the science feed rather than technology feed. So can we offer them the choice of download!

Adjust the style

Well we already know we are going to be building some more PDFs with that blue color, so let's add the red style explicitly to the document.
We can simply extend the existing styles or add more than one style to specific components. This is one solution to the requirements...

```
<Styles>
  <style:Styles-Ref source="../PDFs/Styles.psfx" />

  <style:Style applied-class="heading" >
    <style:Border color="#971B1E" sides="Top Left Bottom Right"/>
  </style:Style>

  <style:Style applied-class="red-text" >
    <style:Fill color="#971B1E"/>
  </style:Style>

  <style:Style applied-class="red-underline" >
    <style:Border color="#971B1E"/>
  </style:Style>

</Styles>
<Pages>
  <pdf:Section id="MyFirstPage" >
    <Content>

      <data:XMLDataSource id="BBCRss"
            source-path="http://feeds.bbci.co.uk/news/technology/
      </data:XMLDataSource>

      <data:ForEach select="rss/channel" datasource-id="BBCRss" >
        <Template>
          <pdf:Div style:class="heading" >
            <data:If test="string-length(image/url) &gt; 0" >
              <Template>
                <pdf:Link action="Uri" file="{xpath:image/link}"
                  <Content>
                    <pdf:Image src="{xpath:image/url}" style:width
                  </Content>
                </pdf:Link>
              </Template>
            </data:If>
            <pdf:H1 style:class="red-text" text="{xpath:title}"
            <pdf:Label text="{xpath:description}" /><BR/>
            Date: <pdf:Label text="{xpath:lastBuildDate}" />
          </pdf:Div>

          <pdf:Div style:class="content" >
            <data:ForEach select="item" >
              <Template>
                <pdf:Div style:class="rss-item red-underline" >
                  <pdf:Div style:class="rss-item-data" >
                    <pdf:H2 style:class="red-text" text="{xpath:title}" ></pdf:H2>
                    <pdf:Label text="{xpath:description}" />
                  </pdf:Div>
                </pdf:Div>

              </Template>
            </data:ForEach>
          </pdf:Div>
```

Again all the source code is available to copy and paste in **Appendix B** for this first version.
Sorted for the styling, but what about that extra feed! We are going to need to go back to our Default.aspx page for that.

Change the document in code

Let us add a second button to our aspx page, and have both buttons raise a Command event, so we know what is required - Science or Technology.

```
<form id="form1" runat="server">
<div>
    <asp:Button runat="server" Text="Click Me for Tech!" ID="GenerateTechPDF" OnCommand="Page_GeneratePDF" CommandName="Technology" /><br />
    <asp:Button runat="server" Text="Click Me for Science!" ID="GenerateSciencePDF" OnCommand="Page_GeneratePDF" CommandName="Science" />
</div>
</form>
```

We now have 2 commands coming to our event, and based on that we can change the feed.

```
protected void Page_GeneratePDF(object sender, CommandEventArgs e)
{
    string path = string.Empty;

    if (string.Equals(e.CommandName, "Science"))
        path = "http://feeds.bbci.co.uk/news/science_and_environment/rss.xml";
    else if (string.Equals(e.CommandName, "Technology"))
        path = "http://feeds.bbci.co.uk/news/technology/rss.xml";

    PDFDocument doc = PDFDocument.ParseDocument("./App_Data/PDFs/Document.pdfx");
    Scryber.Data.PDFXMLDataSource src = doc.FindAComponentById("BBCRss") as Scryber.Data.PDFXMLDataSource;

    if (null != src && !string.IsNullOrEmpty(path))
        src.SourcePath = path;

    doc.ProcessDocument(this.Response);
}
```

Well, let's render and see what happens when we choose Science...

This is quite a big thing! It not only means that the layout is dynamic, we knew that.
But it also means that the parsing of a document creates components that can be manipulated.
And that we can programatically add or remove components to that document as we require.

So how does the parser know which component it is to be created then?
All the xml files have namespaces that are references to .NET Namespaces and Assemblies.
The parser looks at the XML elements and identifies their runtime Type based on the assembly and namespace.
And the classes decorated with the [PDFParsableComponent(..)] attribute can be parsed from the document
This means... you can create your own components and include them within the xml. They will be created and included as part of the document, without changing the scryber source code. Enjoy!

A bit about the badge

You may have noticed that scryber includes a badge on every page - Generated by scryber.
We have thought long and hard about how to release this library. We want to put it out there, we want people to use it. We have also spent a long time developing scryber, investing a lot of time and effort, and we think this is the best way we can continue to work on scryber.
It stays open source under LGPL. Everyone has access, everyone can use it and no one has to pay for it, and it puts a badge on the page so everyone can see where it came from, and may be use it themselves.

If you need to remove the badge, or have a client that wants it removed then we'd really like you to buy a small licence file from us that will remove the badge at www.scryber.co.uk, otherwise you can get your hands dirty in the code and remove the badge rendering from the page.

We obviously prefer the former, and we hope to create extensions / plugins that support the same capability (freely available, but licence is required for badge removal). We also hope that others, including yourselves, will create plugins with the same capability.

If you do remove the code to render the badge, the LGPL says you can do that - if it's for internal use, or you also release under LGPL.
But you will have to invest effort in keeping it up to date with the scryber releases.

We like the badge, and we like scryber, and we like it open source.

What's Next

This has been a pretty deep dive into the capabilities of scryber. Documents, pages, components, styles, binding and code has all been touched on.
And we hope you have enjoyed it.
There are still a lot more things you can do with scryber - component positioning, embedded fonts, background images, referenced components, creating your own components. But we hope this has been enough to spark your interest.
There are also still lot's more things we want to do with scryber - improve flow across regions, fix relative postioning on full width, and other gripes...
Plus - tables, lists, SQL/Object sources, graphs, text/rtf import, SVGL, unicode, security, zipped streams, forms, and many nore extensions.

The best resource for finding out about scryber, and asking questions is http://www.scryber.co.uk/.
Documentation is still very light whilst we are in Beta phase, but it's growing.

Get yourself over there, take a look around, and contribute!

Appendix A: Source code v1 (Blue)

Here's the source for all 4 files created with the this example.

Default.aspx

```
<form id="form1" runat="server">
    <div>
        <asp:Button runat="server" Text="Click Me for Tech!" ID="GenerateTechPDF"
        OnClick="Page_GeneratePDF" />
    </div>
</form>
```

Default.aspx.cs

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void Page_GeneratePDF(object sender, EventArgs e)
    {
        PDFDocument doc = PDFDocument.ParseDocument("./App_Data/PDFs/Document.pdfx");
        doc.ProcessDocument(this.Response);

    }
}
```

Document.pdfx

```
<?xml version="1.0" encoding="utf-8"?>
<pdf:Document xmlns:pdf="Scryber.Components, Scryber.Components, Version=0.8.0.0, Culture=neutral, Publ
xmlns:style="Scryber.Styles, Scryber.Styles, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81
xmlns:data="Scryber.Data, Scryber.Components, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb8
auto-bind="true" >
<Styles>
<style:Styles-Ref source="../PDFs/Styles.psfx" />

</Styles>
<Pages>
<pdf:Section id="MyFirstPage" >
<Content>

<data:XMLDataSource id="BBCRss"
source-path="http://feeds.bbci.co.uk/news/technology/rss.xml" cache-duration="5" >
</data:XMLDataSource>

<data:ForEach select="rss/channel" datasource-id="BBCRss" >
<Template>
<pdf:Div style:class="heading" >
<pdf:H1 text="{xpath:title}"></pdf:H1>
<pdf:Label text="{xpath:description}" /><BR/>
Date: <pdf:Label text="{xpath:lastBuildDate}" />
</pdf:Div>

<pdf:Div style:class="content" >
<data:ForEach select="item" >
<Template>
<pdf:Div >
<pdf:Div style:class="rss-item-data" >
<pdf:H2 text="{xpath:title}" ></pdf:H2>
<pdf:Label text="{xpath:description}" />
</pdf:Div>
</pdf:Div>

</Template>
</data:ForEach>
</pdf:Div>

</Template>
</data:ForEach>

</Content>
</pdf:Section>
</Pages>
</pdf:Document>
```

Styles.psfx

```
<style:Styles xmlns:pdf="Scryber.Components, Scryber.Components, Version=0.8.0.0, Culture=neutral, PublicKeyToken=87
xmlns:style="Scryber.Styles, Scryber.Styles, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe"
xmlns:data="Scryber.Data, Scryber.Components, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe"
>

<style:Style applied-type="pdf:Page" >
<style:Page size="A4" orientation="Portrait"/>
<style:Margins left="20pt" bottom="40pt" right="20pt" top="15pt"/>
</style:Style>

<style:Style applied-class="heading" >
<style:Border color="#00FFFF" width="1pt" sides="Top Left Right" />
<style:Background color="#EEEEEE"/>
<style:Padding all="10pt"/>
<style:Margins top="20pt" bottom="20pt"/>
<style:Position h-align="Center"/>
<style:Font bold="true" size="12pt" />
</style:Style>

<style:Style applied-type="pdf:Div" applied-class="content" >
<style:Columns count="2" alley-width="20pt"/>
<style:Font size="12pt" bold="false" italic="false" />
</style:Style>

<style:Style applied-type="pdf:H2" >
<style:Font size="14pt" />
<style:Fill color="#00FFFF"/>
</style:Style>

<style:Style applied-type="pdf:H1" >
<style:Fill color="#00FFFF"/>
</style:Style>

<style:Style applied-class="rss-item" >
<style:Margins bottom="10pt"/>
<style:Padding bottom="3pt"/>
<style:Border color="#00FFFF" sides="Bottom" width="1pt"/>
</style:Style>

</style:Styles>
```

Appendix B: Source code v2 (Red)

Here's the source for the 3 modified files updated for the red style (with the changes highlighted in red)

Default.aspx

```
<form id="form1" runat="server">
    <div>
        <asp:Button runat="server" Text="Click Me for Tech!" ID="GenerateTechPDF"
        OnCommand="Page_GeneratePDF" CommandName="Technology" /><br />
        <asp:Button runat="server" Text="Click Me for Science!" ID="GenerateSciencePDF"
        OnCommand="Page_GeneratePDF" CommandName="Science" />
    </div>
</form>
```

Default.aspx.cs

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void Page_GeneratePDF(object sender, CommandEventArgs e)
    {
        string path = string.Empty;

        if (string.Equals(e.CommandName, "Science"))
            path = "http://feeds.bbci.co.uk/news/science_and_environment/rss.xml";
        else if (string.Equals(e.CommandName, "Technology"))
            path = "http://feeds.bbci.co.uk/news/technology/rss.xml";
        PDFDocument doc = PDFDocument.ParseDocument("./App_Data/PDFs/Document.pdfx");
        Scryber.Data.PDFXMLDataSource src = doc.FindAComponentById("BBCRss") as
        Scryber.Data.PDFXMLDataSource;

        if (null != src && !string.IsNullOrEmpty(path))
        src.SourcePath = path;
        doc.ProcessDocument(this.Response);

    }
}
```

Document.pdfx

```
<?xml version="1.0" encoding="utf-8"?>
<pdf:Document xmlns:pdf="Scryber.Components, Scryber.Components, Version=0.8.0.0, Culture=neutral, PublicKeyToken=87
xmlns:style="Scryber.Styles, Scryber.Styles, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe"
xmlns:data="Scryber.Data, Scryber.Components, Version=0.8.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe"
auto-bind="true" >
<Styles>
<style:Styles-Ref source="../PDFs/Styles.psfx" />

<style:Style applied-class="heading" >
<style:Border color="#971B1E" sides="Top Left Bottom Right"/>
</style:Style>

<style:Style applied-class="red-text" >
<style:Fill color="#971B1E" />
</style:Style>

<style:Style applied-class="red-underline" >
<style:Border color="#971B1E" />
</style:Style>
</Styles>
<Pages>
<pdf:Section id="MyFirstPage" >
<Content>

<data:XMLDataSource id="BBCRss"
source-path="http://feeds.bbci.co.uk/news/technology/rss.xml" cache-duration="5" >
</data:XMLDataSource>

<data:ForEach select="rss/channel" datasource-id="BBCRss" >
<Template>
<pdf:Div style:class="heading" >
<data:If test="string-length(image/url) &gt; 0" >
<Template>
<pdf:Link action="Uri" file="{xpath:image/link}" new-window="true" alt="{xpath:image/title}">
<Content>
<pdf:Image src="{xpath:image/url}" style:width="{xpath:image/width}" style:padding="10" />
</Content>
</pdf:Link>
</Template>
</data:If>
<pdf:H1 style:class="red-text"text="{xpath:title}"></pdf:H1>
<pdf:Label text="{xpath:description}" /><BR/>
Date: <pdf:Label text="{xpath:lastBuildDate}" />
</pdf:Div>

<pdf:Div style:class="content" >
<data:ForEach select="item" >
<Template>
<pdf:Div style:class="rss-item red-underline">
<pdf:Div style:class="rss-item-data" >
<pdf:H2 style:class="red-text"text="{xpath:title}" ></pdf:H2>
<pdf:Label text="{xpath:description}" />
</pdf:Div>
</pdf:Div>

</Template>
</data:ForEach>
</pdf:Div>

</Template>
</data:ForEach>

</Content>
</pdf:Section>
</Pages>
</pdf:Document>
```