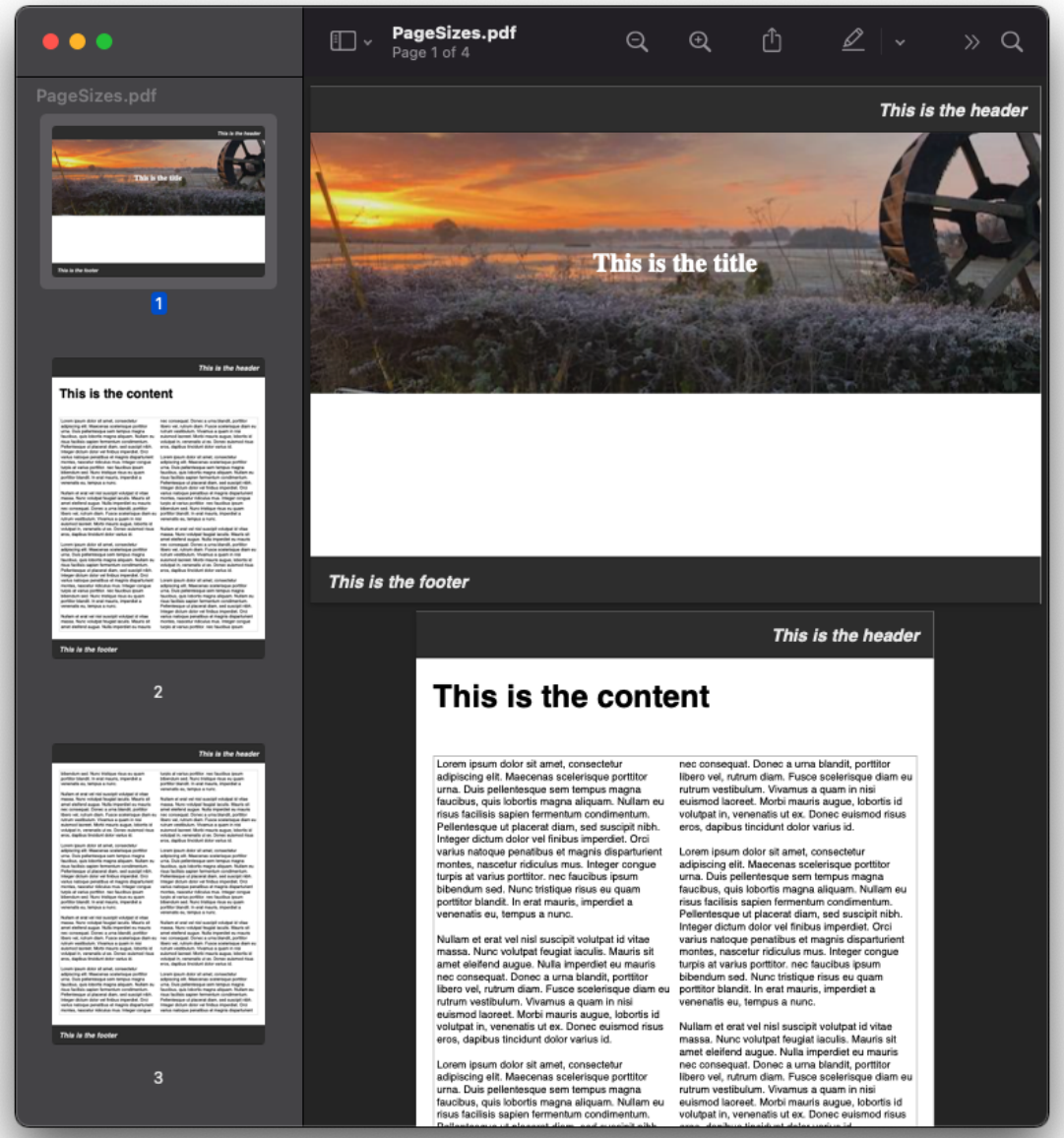
Scryber Core

Oct 01, 2023

Scryber Overview


PageSizes.pdf
Page 1 of 4
This is the header
This is the title
This is the footer
This is the content
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas scelerisque porttitor urna. Duis pellentesque sem tempus magna faucibus, quis lobortis magna aliquam. Nullam eu risus facilisis sapien fermentum condimentum. Pellentesque ut placerat diam, sed suscipit nibh. Integer dictum dolor vel finibus imperdiet. Orci varius natoque penatibus et magnis disparturient montes, nascetur ridiculus mus. Integer congue turpis at varius porttitor. nec faucibus ipsum bibendum sed. Nunc tristique risus eu quam porttitor blandit. In erat mauris, imperdiet a venenatis eu, tempus a nunc.
Nullam et erat vel nisl suscipit volutpat id vitae massa. Nunc volutpat feugiat iaculis. Mauris sit amet eleifend augue. Nulla imperdiet eu mauris nec consequat. Donec a urna blandit, porttitor libero vel, rutrum diam. Fusce scelerisque diam eu rutrum vestibulum. Vivamus a quam in nisi euismod laoreet. Morbi mauris augue, lobortis id volutpat in, venenatis ut ex. Donec euismod risus eros, dapibus tincidunt dolor varius id.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas scelerisque porttitor urna. Duis pellentesque sem tempus magna faucibus, quis lobortis magna aliquam. Nullam eu risus facilisis sapien fermentum condimentum.
nec consequat. Donec a urna blandit, porttitor libero vel, rutrum diam. Fusce scelerisque diam eu rutrum vestibulum. Vivamus a quam in nisi euismod laoreet. Morbi mauris augue, lobortis id volutpat in, venenatis ut ex. Donec euismod risus eros, dapibus tincidunt dolor varius id.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas scelerisque porttitor urna. Duis pellentesque sem tempus magna faucibus, quis lobortis magna aliquam. Nullam eu risus facilisis sapien fermentum condimentum. Pellentesque ut placerat diam, sed suscipit nibh. Integer dictum dolor vel finibus imperdiet. Orci varius natoque penatibus et magnis disparturient montes, nascetur ridiculus mus. Integer congue turpis at varius porttitor. nec faucibus ipsum bibendum sed. Nunc tristique risus eu quam porttitor blandit. In erat mauris, imperdiet a venenatis eu, tempus a nunc.
Nullam et erat vel nisl suscipit volutpat id vitae massa. Nunc volutpat feugiat iaculis. Mauris sit amet eleifend augue. Nulla imperdiet eu mauris nec consequat. Donec a urna blandit, porttitor libero vel, rutrum diam. Fusce scelerisque diam eu rutrum vestibulum. Vivamus a quam in nisi euismod laoreet. Morbi mauris augue, lobortis id volutpat in, venenatis ut ex. Donec euismod risus
PageSizes.pdf
1
2
3

CHAPTER 1

Simple, data driven, good looking, documents from templates

Scryber is **the** engine to create dynamic PDF documents quickly and easily from XHTML templates with consistant styles, your own data, and an easy flowing layout. It's open source; flexible; styles based; data driven and with a low learning curve.

Written entirely in C# for dotnet 5 using HTML, CSS and SVG.

Documentation for the 5.0.x versions is here 5.0.6 Read the docs here Documentation for previous 1.0.x pdfx versions for 1.0.0 Read the docs here

CHAPTER 2

Hello World MVC

Download the nuget package

https://www.nuget.org/packages/Scryber.Core.Mvc

Start with a template. **The xmlns namespace declaration is important.**

```
<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>{{hello}}</title>
    </head>
    <body>
        <div style='padding:10px'>{{hello}}.</div>
    </body>
</html>
```

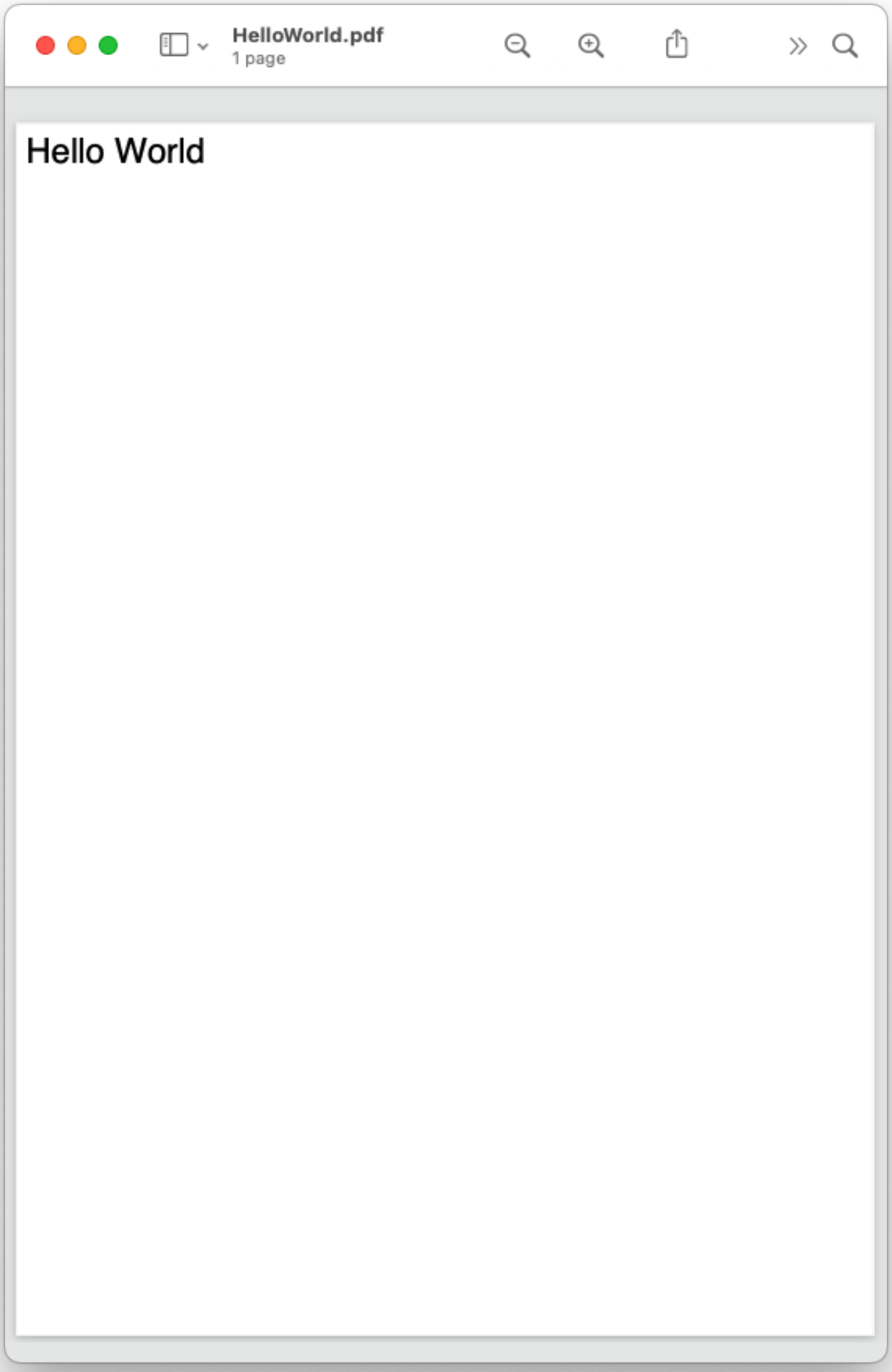
And then generate your template in a view.

```
//add the namespaces
using Scryber.Components;
using Scryber.Components.Mvc;
using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller
{

    private readonly IWebHostEnvironment _env;

    public HomeController(IWebHostEnvironment environment)
    {
        _env = environment;
    }

    [HttpGet]
    public IActionResult HelloWorld()
```

(continues on next page)

(continued from previous page)

```
    {
        // get the path to where you have saved your template
        var path = _env.ContentRootPath;
        path = System.IO.Path.Combine(path, "Views", "PDF", "HelloWorld.html");

        //parsing the document creates a complete object graph from the content
        using(var doc = Document.ParseDocument(path))
        {
            doc.Params["hello"] = "Hello World";
            return this.PDF(doc); //convenience extension method to return the result.
        }
    }

}
```


HelloWorld.pdf
1 page
Hello World

CHAPTER 3

Getting Started

Check out *1.2. MVC Controller - Getting Started* for a full MVC example with styles and binding. Or *1.1. Console or GUI - Getting Started* for a full gui application example (with styles and binding)

CHAPTER 4

Easy, and intuitive structure

Whether you are using xhtml templates or directly in code, scryber is quick and easy to build complex documents from your designs and data using standard xhtml.

2_document/document_overview

CHAPTER 5

Intelligent flowing layout engine

In scryber, content can either be laid out explicitly, or jut flowing with the the page. Change the page size, or insert content and everything will adjust around it.

Components overview - TD

CHAPTER 6

Cascading Styles

With a styles based structure, it's easy to apply designs to templates. Use class names, id's or component types, or nested selectors.

Styles in your template - PD

CHAPTER 7

Drawing and Typographic support

Scryber supports inclusion of Images, Fonts (inc. Google fonts) and SVG components for drawing graphics and icons.

Drawing with SVG - PD

CHAPTER 8

Binding to your data

With a simple handlebars binding notation it's easy to add references to your data structures and pass information and complex data to your document from your model and more.

Now supporting full expressions support including css var and calc support

Dynamic content in your template - PD

CHAPTER 9

Extensible Framework

Scryber was designed from the ground up to be extensible. If it doesn't do what you need, then we think you can make it do it. With iFrame includes, a namespace based parser engine, and configuration options for images, fonts, binding it's down to your imagination

Extending Scryber - TD

CHAPTER 10

Secure and Encrypted

Scryber fully supports the PDF restrictions and both 40 bit and 128 bit encryption of documents using owner and user passwords.

Securing Documents - PD

10.1 1.1. Console or GUI - Getting Started

A Complete example for creating a hello world PDF file in a console application or GUI front end. For us, we have just created a new dotnet core console application in Visual Studio.

10.1.1 1.1.1. Nuget Packages

Make sure you install the Nuget Packages from the Nuget Package Manager

https://www.nuget.org/packages/Scryber.Core

This will add the latest version of the Scryber.Core nuget package.

10.1.2 1.1.2. Add a document template

In our applications we like to add our templates to a PDF folder. You can break it down however works for you, but for now a create a new XHML file called HelloWorld.html in your folder.

And paste the following content into the file

```
<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>Hello World</title>
    </head>
    <body>
```

(continues on next page)

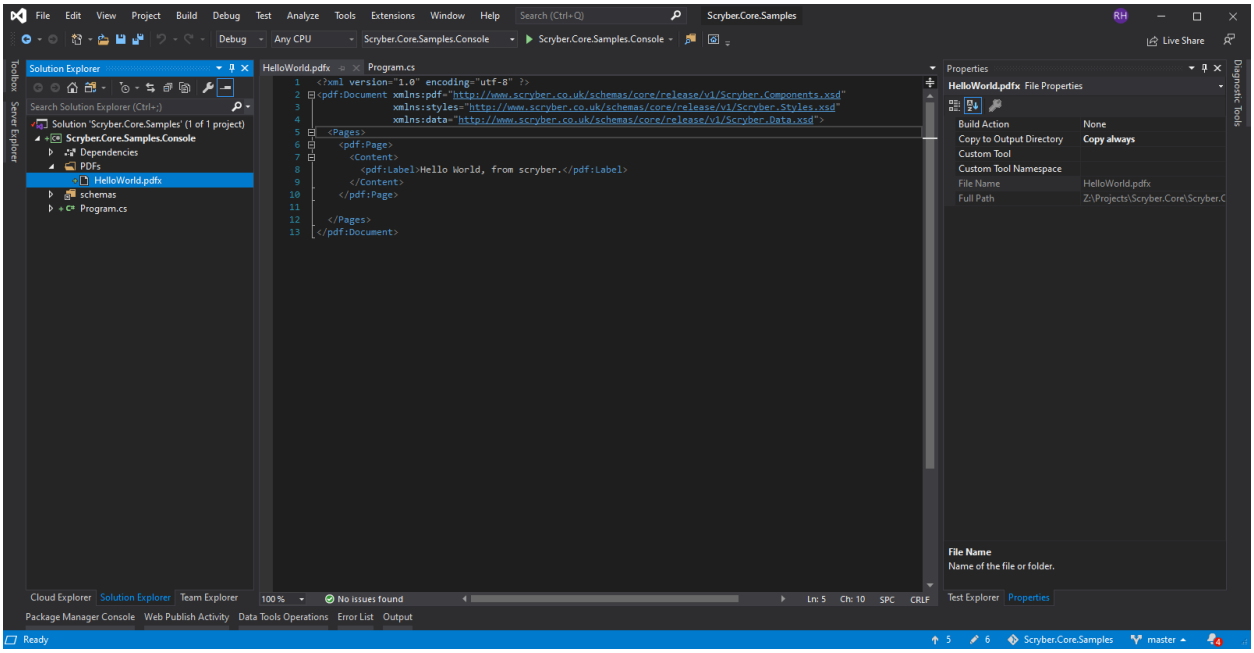
(continued from previous page)

```
        <div style='padding:10px'>Hello World from scryber.</div>
    </body>
</html>
```

10.1.3 1.1.3. File properties

In the file properties for the HelloWorld.html file: Set the Build Action to None (if it is not already) And the Copy to output to Always.

Your solution should look something like this.



10.1.4 1.1.4. Program code

In your program.cs add the namespace to the top of your class.

```
using Scryber.Components;
```

10.1.5 1.1.5. Replace your main method.

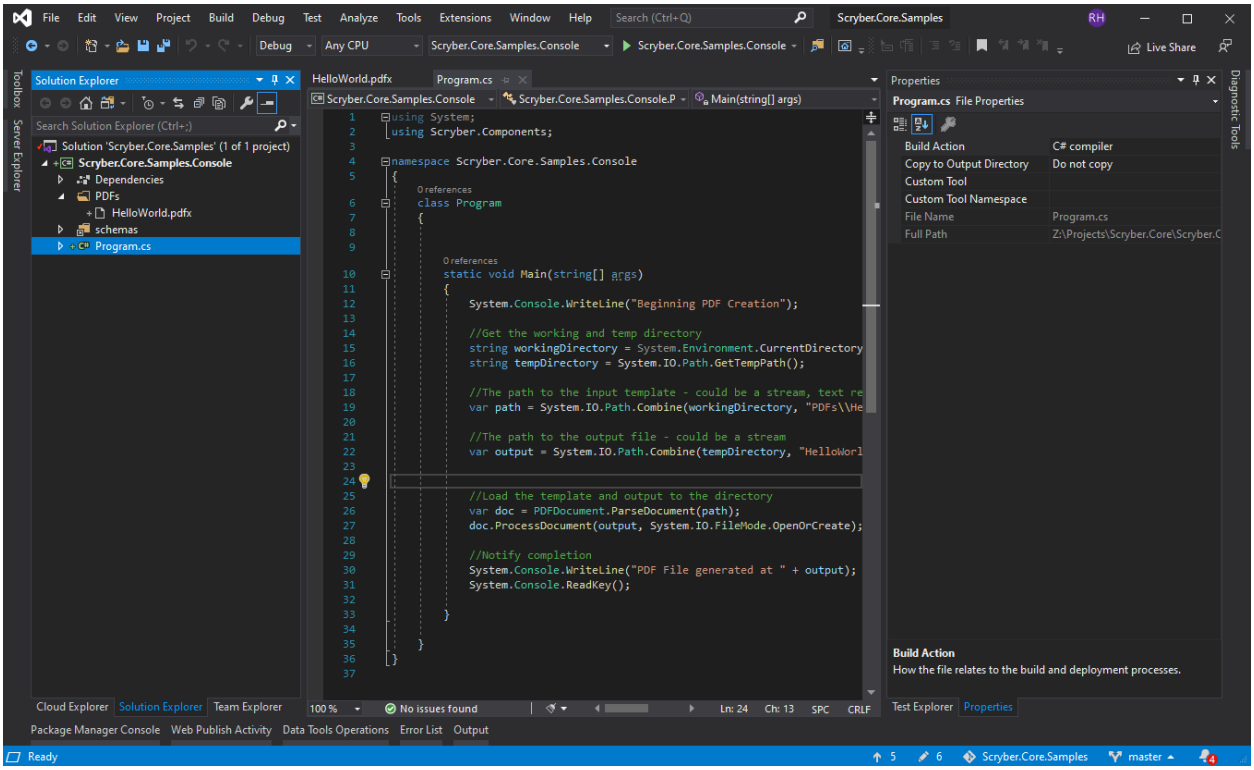
Next change the 'Main' method to your class to load the template and generate the pdf file

```
static void Main(string[] args)
{
    System.Console.WriteLine("Beginning PDF Creation");

    //Get the working and temp directory
    string workingDirectory = System.Environment.CurrentDirectory;
    string tempDirectory = System.IO.Path.GetTempPath();
```

(continues on next page)

(continued from previous page)

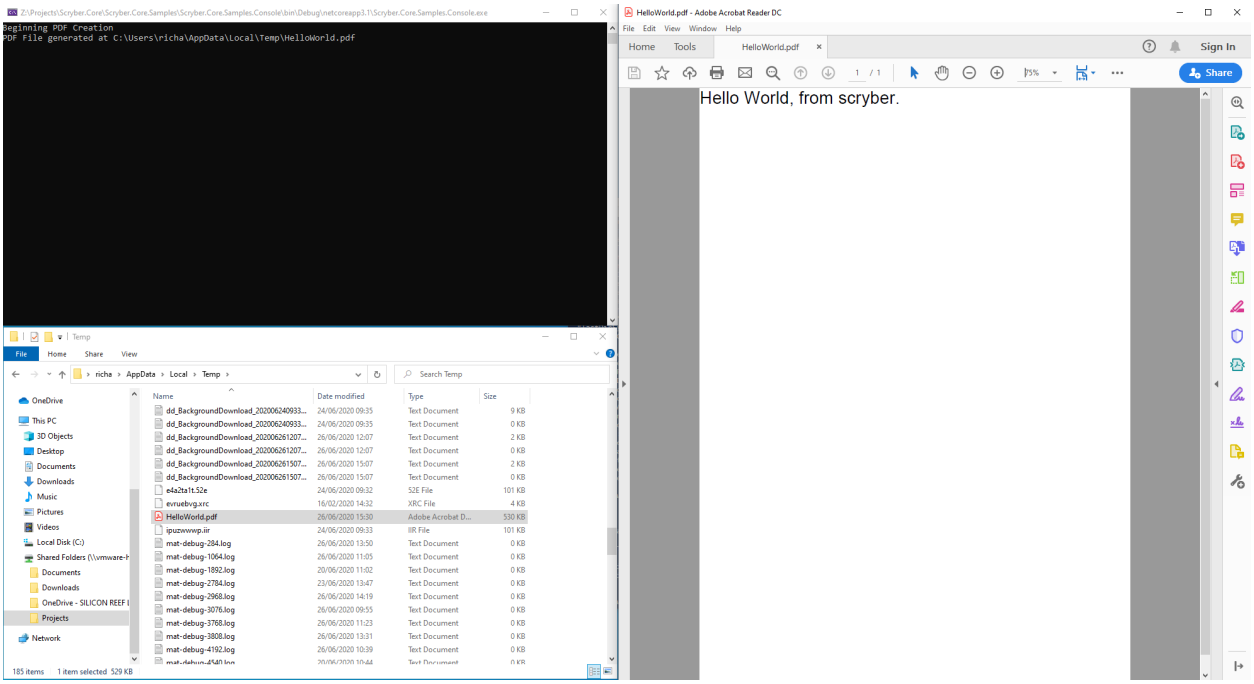
```
    //The path to the input template - could be a stream, text reader, xml reader,␣
↪resource etc
    var path = System.IO.Path.Combine(workingDirectory, "PDFs\\HelloWorld.html");

    //The path to the output file - could be a stream
    var output = System.IO.Path.Combine(tempDirectory, "HelloWorld.pdf");


    //Load the template and output to the directory
    var doc = Document.ParseDocument(path);
    doc.SaveAsPDF(output, System.IO.FileMode.OpenOrCreate);

    //Notify completion
    System.Console.WriteLine("PDF File generated at " + output);
    System.Console.ReadKey();

}
```



The parser will read the document from the XHTML content, and then create a new PDF document in the tempDirectory for the output.

10.1.6 1.1.6. Testing your code

Running your application, you should see the console output the path to the pdf. And opening this will show you the file. you could have saved it to a share, opened in Acrobat reader, or sent via email as a stream attachment.



10.1.7 1.1.7. Adding dynamic content

One of the driving forces behind scryber is the separation of the content, data and style. It is common practice in sites. With scryber all attributes and content is bindable to the data you want to pass to it,

So we can specify our model data with from any source (here we are just using a dynamic object). And we can pass it to the parsed document either explicitly, or using the special 'model' overload on the PDF extension method.

```
private static dynamic GetHelloWorldData()
{
    //get your model data however you wish
    //it's just a sample object for this one.

    var model = new
        {
            titlecolor = "#ff6347", //style data
            title = "scryber", //simple content
            items = new[]                   //or even complex object data
            {
                new { name = "First item" },
                new { name = "Second item" },
                new { name = "Third item" },
            }
        };

    return model;
}

static void Main(string[] args)
{
    System.Console.WriteLine("Beginning PDF Creation");

    string workingDirectory = System.Environment.CurrentDirectory;
```

(continues on next page)

(continued from previous page)

```
    string tempDirectory = System.IO.Path.GetTempPath();

    var path = System.IO.Path.Combine(workingDirectory, "PDFs\\HelloWorld.html");

    var output = System.IO.Path.Combine(tempDirectory, "HelloWorld.pdf");

    var doc = Document.ParseDocument(path);

    //Assign the data model to a parameter
    doc.Params["model"] = GetHelloWorldData();

    doc.SaveAsPDF(output, System.IO.FileMode.OpenOrCreate);

    //Notify completion
    System.Console.WriteLine("PDF File generated at " + output);
    System.Console.ReadKey();

}
```

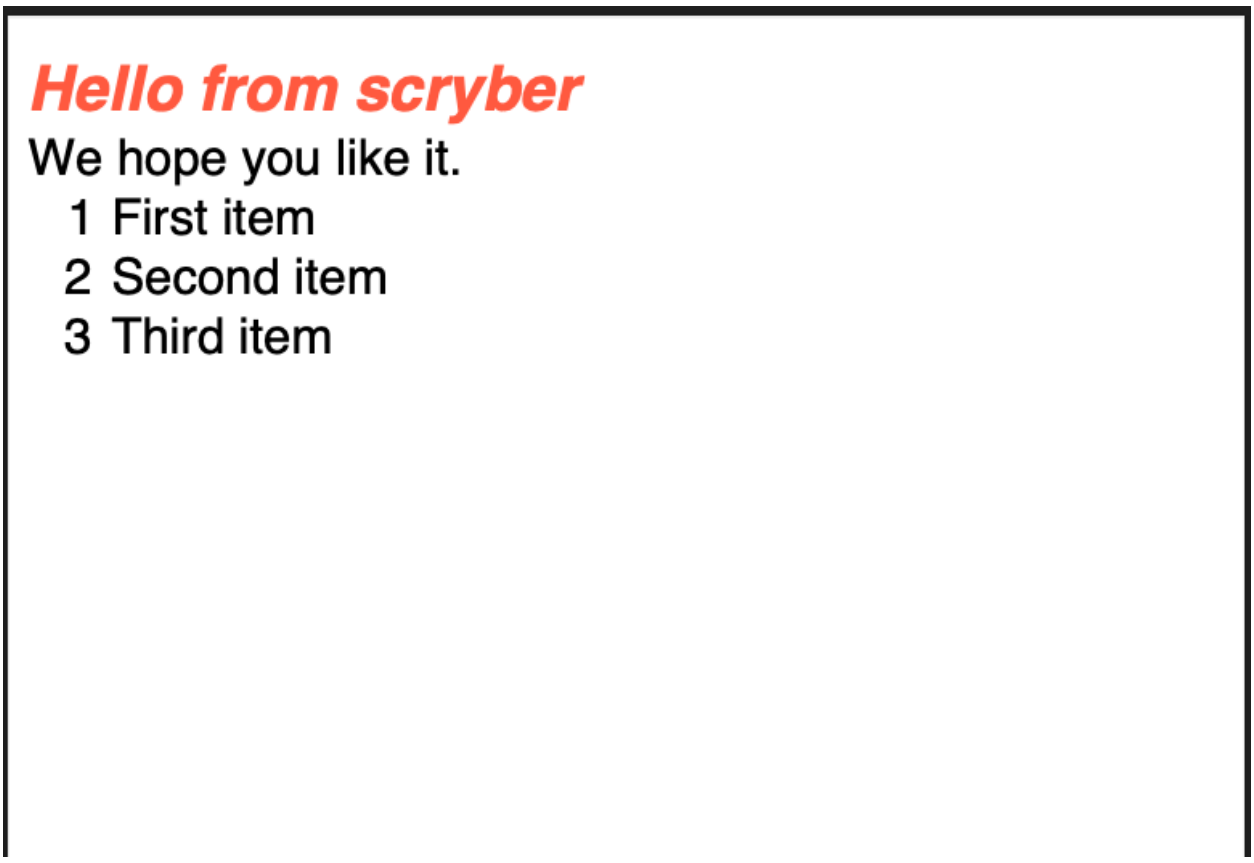
The general syntax for referring paramters in a template is

{{**parameter[.property]**}}

And the html5 tag 'template' is used with the data-bind attribute to loop over one or more items in a collection, and the inner objects and properties can be used with the '.' prefix to reference the current data context.

So we can expand our document body to use the model schema.

```
<body>

    <main style="padding:10pt">

        <!-- binding styles and values on content -->
        <h2 style="color:{{model.titlecolor}}">{{concat("Hello from ",model.title)}}</
↪h2>

        <div>We hope you like it.</div>

        <!-- Loop with nested item collection binding to the objects -->
        <ol>
            <template data-bind='{{model.items}}'>
                <!-- binding within the model.items content, and can be nested -->
                <li>{{.name}}</li>
            </template>
        </ol>
    </main>

</body>
```



10.1.8 1.1.8. Adding Fonts and Styles

It's good but rather uninspiring. With scryber we can use css styles, just as we would in html.

Here we are:

- Adding a stylesheet link to the google 'Fraunces' font with the @font-face at-rule (watch that &display=swap link - it's not xhtml)
- Adding some document styles for the body with fall-back fonts.
- A complex style for a page header, with a colour and single background image, that will be repeated across any page.
- And a page footer table with full width and associated style on the inner cells, that will again be repeated.

The css style could just have easily come from another referenced stylesheet. Do not forget to encode the & character as &

```
<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>Hello World</title>

        <!-- support for complex css selectors (or link ot external style sheets )-->
        <link rel="stylesheet"
            href="https://fonts.googleapis.com/css2?family=Fraunces:ital,wght@0,400;0,
↪700;1,400;1,700&amp;display=swap"
```

(continues on next page)

(continued from previous page)

```
            title="Fraunces" />

        <style>
            body {
                font-family: 'Roboto', sans-serif;
                font-size: 14pt;
            }

            p.header {
                color: #AAA;
                background-color: #333;
                background-image: url('https://avatars.githubusercontent.com/u/
↪65354830?s=64&amp;v=4');
                background-repeat: no-repeat;
                background-position: 10pt 10pt;
                background-size: 20pt 20pt;
                margin-top: 0pt;
                padding: 10pt 10pt 10pt 35pt;
            }


            .foot td {
                border: none;
                text-align: center;
                font-size: 10pt;
                margin-bottom: 10pt;
            }
        </style>
    </head>
    <body>
        <header>
            <!-- document headers -->
            <p class="header">Scryber document creation</p>
        </header>
        <!-- support for many HTML5 tags-->
        <main style="padding:10pt">

            <!-- binding styles and values on content -->
            <h2 style="color:{{model.titlecolor}}">{{concat("Hello from ",model.title)}
↪}</h2>

            <div>We hope you like it.</div>

            <!-- Loop with nested item collection binding to the objects -->
            <ol>
                <template data-bind='{{model.items}}'>
                    <!-- binding within the model.items content, and can be nested -->
                    <li>{{.name}}</li>
                </template>
            </ol>

        </main>
        <footer>
            <!-- footers in a table with style -->
            <table class="foot" style="width:100%">
                <tr>
                    <td>{{model.author}}</td>
```

(continues on next page)

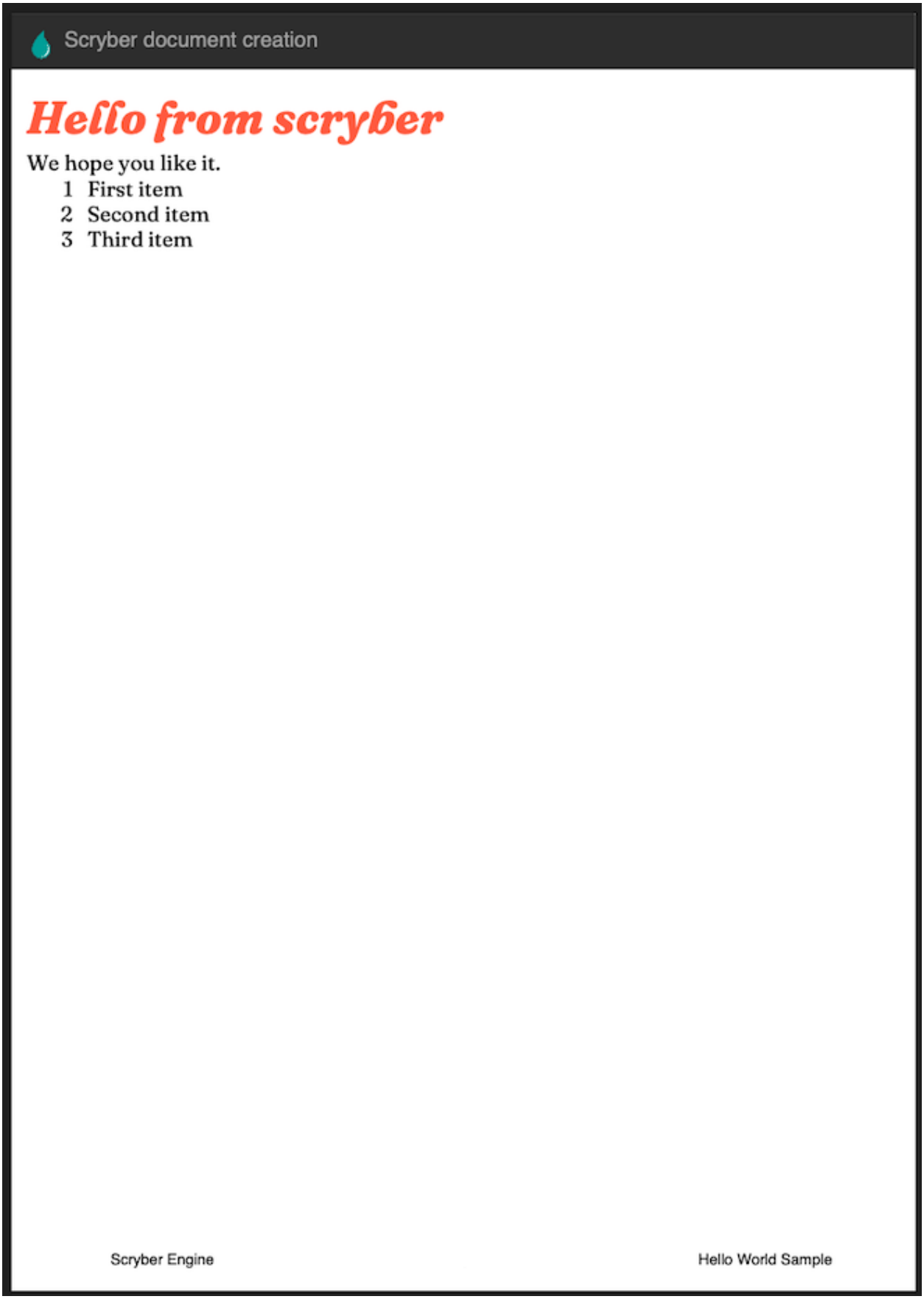
(continued from previous page)

```
                    <td>Hello World Sample</td>
                </tr>
            </table>
        </footer>
    </body>
</html>
```

Make some minor changes to our model.

```
using Scryber.Components;
var model = new
{
     author = "Scryber Engine",
     titlecolor = "#ff6347 font-family:'Fraunces'", //style data
...
```

The output from this is much more pleasing. Especially that Fruances font :-)


Scryber document creation
Hello from scryber
We hope you like it.
1 First item
2 Second item
3 Third item
Scryber Engine
Hello World Sample

10.1.9 1.1.9. Further reading

You can read more about the what you can do with scryber from the contents.

We have no idea what you will be able to create with scryber. It's just there to hopefully help you build amazing documents in an easy and repeatable way.

10.2 1.2. MVC Controller - Getting Started

A Complete example for creating a styled and databound hello world PDF file from an MVC Controller in C# with an HTML template

10.2.1 1.2.1. Nuget Packages

If you have not done so already, make sure you install the Nuget Package in your new or existing MVC project.

https://www.nuget.org/packages/Scryber.Core.Mvc

This will add the latest version of the Scryber.Core, and also the Scryber.Core.Mvc Controller extension methods.

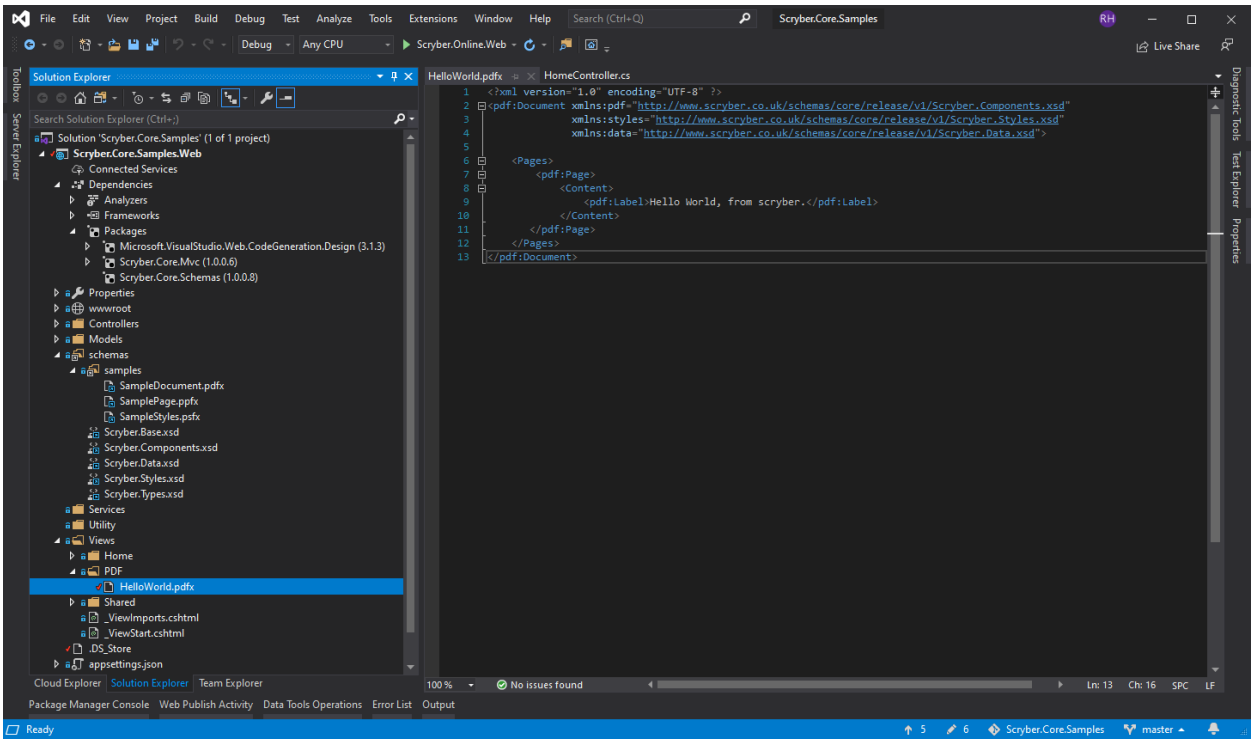
10.2.2 1.2.2. Add a document template

In our applications we like to add our templates to a PDF folder the Views folder. You can break it down however works for you, but for now, a create a new html file called HelloWorld.html in your folder.

And paste the following content into the file

```
<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <div style='padding:10px'>Hello World from scryber.</div>
    </body>
</html>
```

Your solution should look something like this.



The xmlns is really important for knowing what type of document and schema is being used.

For for more information on the namespaces and mappings see this namespaces_and_assemblies documentation

10.2.3 1.2.3. Controller code

Add a new 'Document' controller to your project, and a couple of namespaces are important to add to the top of your controller.

```
using Scryber.Components;
using Scryber.Components.Mvc;
```

10.2.4 1.2.4. Add the Web host service

In order to nicely reference files in the solution, we add a reference to the IWebHostEnvironment to the controller constructor.

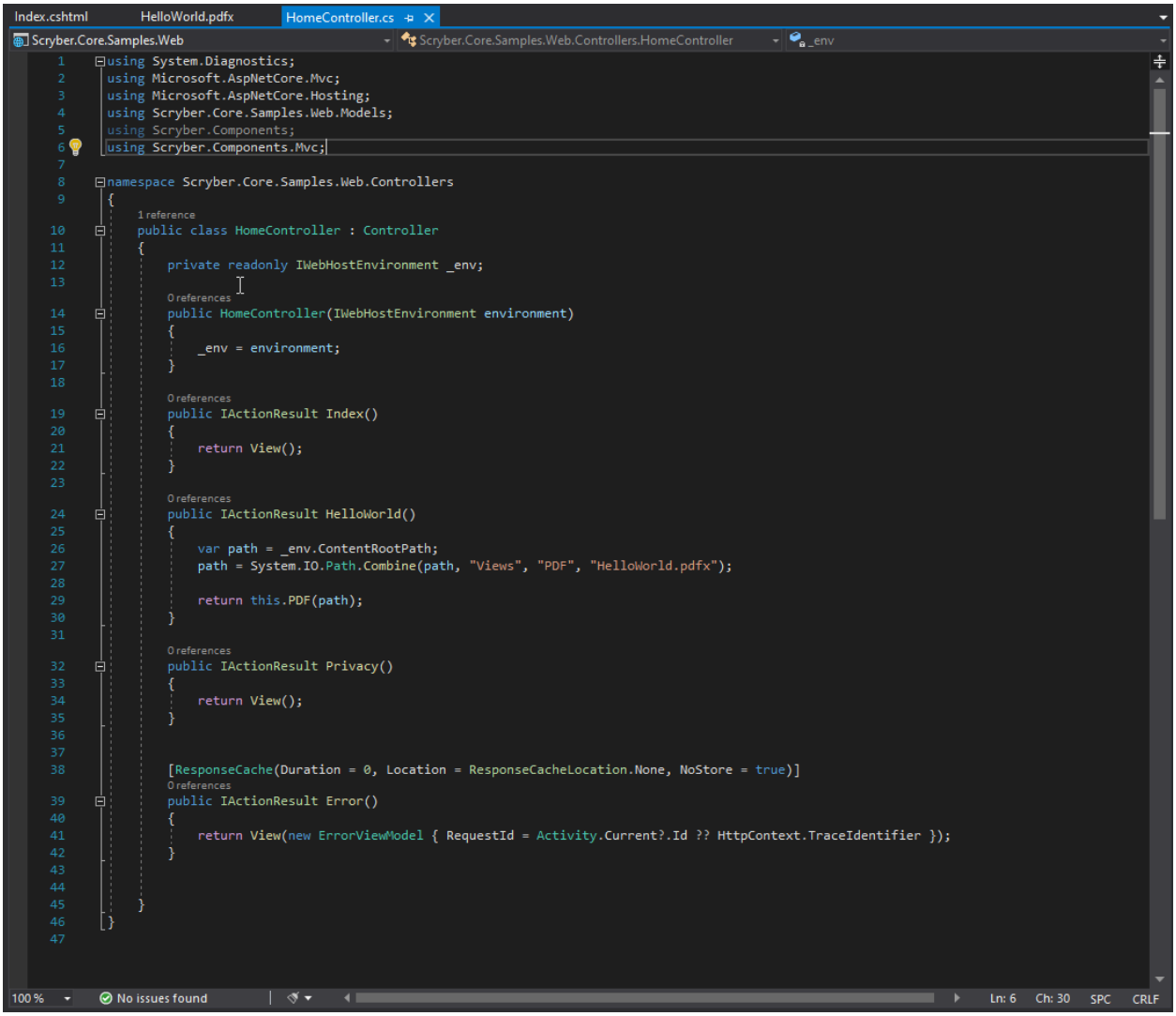
```
private readonly IWebHostEnvironment _env;

public DocumentController(IWebHostEnvironment environment)
{
    _env = environment;
}
```

10.2.5 1.2.5. Add a Controller Method

Next add a new Controller Method to your class for retrieve and generate

```
[HttpGet]
public IActionResult HelloWorld()
{
    var path = _env.ContentRootPath;
    path = System.IO.Path.Combine(path, "Views", "PDF", "HelloWorld.html");

    using(var doc = Document.ParseDocument(path))
        return this.PDF(doc);
}
```

The PDF extension method will read the PDF template from the path and generate the file to the response.



10.2.6 1.2.6. Testing your action

To create your pdf simply add a link to your action method in a view.

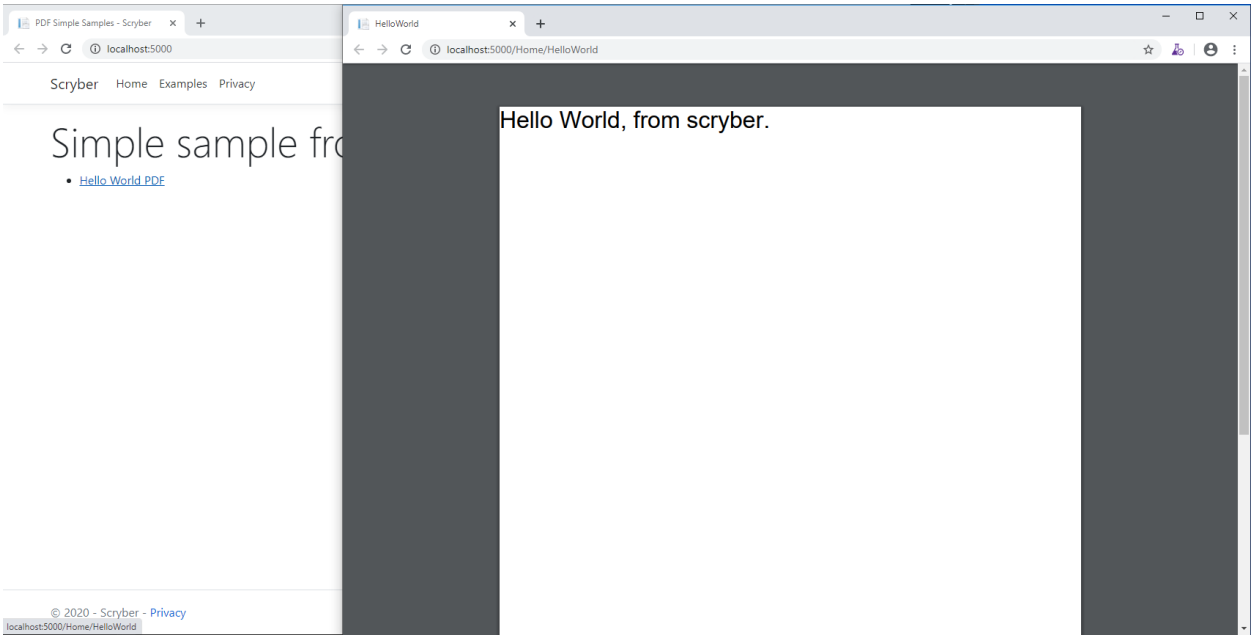
```
<div>
    <h2 class="display-4">Simple sample from the PDF Controller</h2>
```

(continues on next page)

(continued from previous page)

```
    <ul>
        <li><a href='@Url.Action("HelloWorld","Document")' target='_blank'>Hello␣
↪World PDF</a></li>
    </ul>
</div>
```

Running your application, you should see the link and clicking on it will open the pdf in a new tab or window.



10.2.7 1.2.7. Adding dynamic content

One of the driving forces behind scryber is the separation of the content, data and style. It is common practice in sites. With scryber all attributes and content is bindable to the data you want to pass to it,

So we can specify our model data with from any source (here we are just using a dynamic object). And we can pass it to the parsed document either explicitly, or using the special 'model' overload on the PDF extension method.

```
private dynamic GetHelloWorldData(string name)
{
    //get your model data however you wish
    //it's just a sample object for this one.

    var model = new
        {
            titlecolor = "#ff6347", //style data
            name = name, //simple content
            author = "Joe the Mack",
            items = new[]                   //or even complex object data
            {
                new { name = "First item" },
                new { name = "Second item" },
                new { name = "Third item" },
            }
        };
```

(continues on next page)

(continued from previous page)

```
    return model;
}

[HttpGet]
public IActionResult HelloWorld(string name = "scryber")
{
    var path = _env.ContentRootPath;
    path = System.IO.Path.Combine(path, "Views", "PDF", "HelloWorld.html");

    using(var doc = Document.ParseDocument(path))
    {
        //get the data for the model, including an optional parameter from the␣
↪request.

        var model = GetHelloWorldData(name);

        //could use doc.Params["model"] = model; for the same effect.
        //It is just more convenient as below.

        return this.PDF(doc, model);
    }
}
```

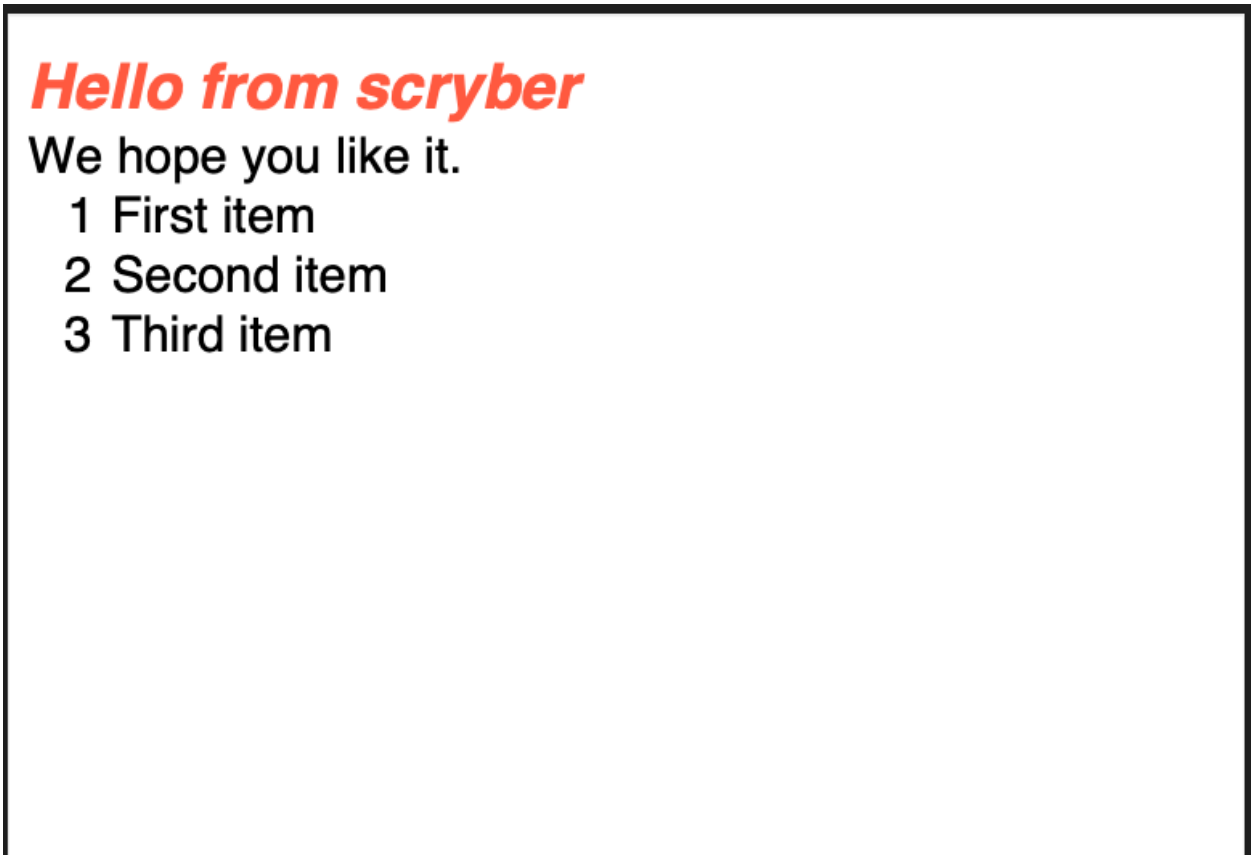
We use a handlebars syntax for the binding to content, and support calculations and expressions within the context.

{{**parameter[.property]**}}

And the html5 tag 'template' is used with the data-bind attribute to loop over one or more items in a collection, and the inner objects and properties can be used with the '.' prefix to reference the current data context.

So we can expand our document body to use the model schema.

```
<body>

    <main style="padding:10pt">

        <!-- binding styles and values on content including expressions and functions␣
↪-->
        <h2 style="color:{{model.titlecolor}}">{{concat('Hello World, from ',model.
↪name)}}</h2>

        <div>We hope you like it.</div>

        <!-- Loop with nested item collection binding to the objects -->
        <ol>
            <template data-bind='{{model.items}}'>
                <!-- binding within the model.items content, and can be nested -->
                <li>{{.name}}</li>
            </template>
        </ol>
    </main>

</body>
```



10.2.8 1.2.8. Adding Fonts and Styles

It's good but rather uninspiring. With scryber we can use styles, just as we would in html.

Here we are:

- Adding a stylesheet link to the google 'Fraunces' font with the @font-face at-rule (watch that &display=swap link - it's not xhtml)
- Adding some document styles for the body with fall-back fonts.
- A complex style for a page header, with a colour and single background image, that will be repeated across any page.
- And a page footer table with full width and associated style on the inner cells, that will again be repeated.

The css style could just have easily come from another referenced stylesheet.

```
<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>Hello World</title>

        <!-- support for complex css selectors (or link to external style sheets )-->
        <link rel="stylesheet"
            href="https://fonts.googleapis.com/css2?family=Fraunces:ital,wght@0,400;0,
↪700;1,400;1,700&amp;display=swap"
            title="Fraunces" />
```

(continues on next page)

(continued from previous page)

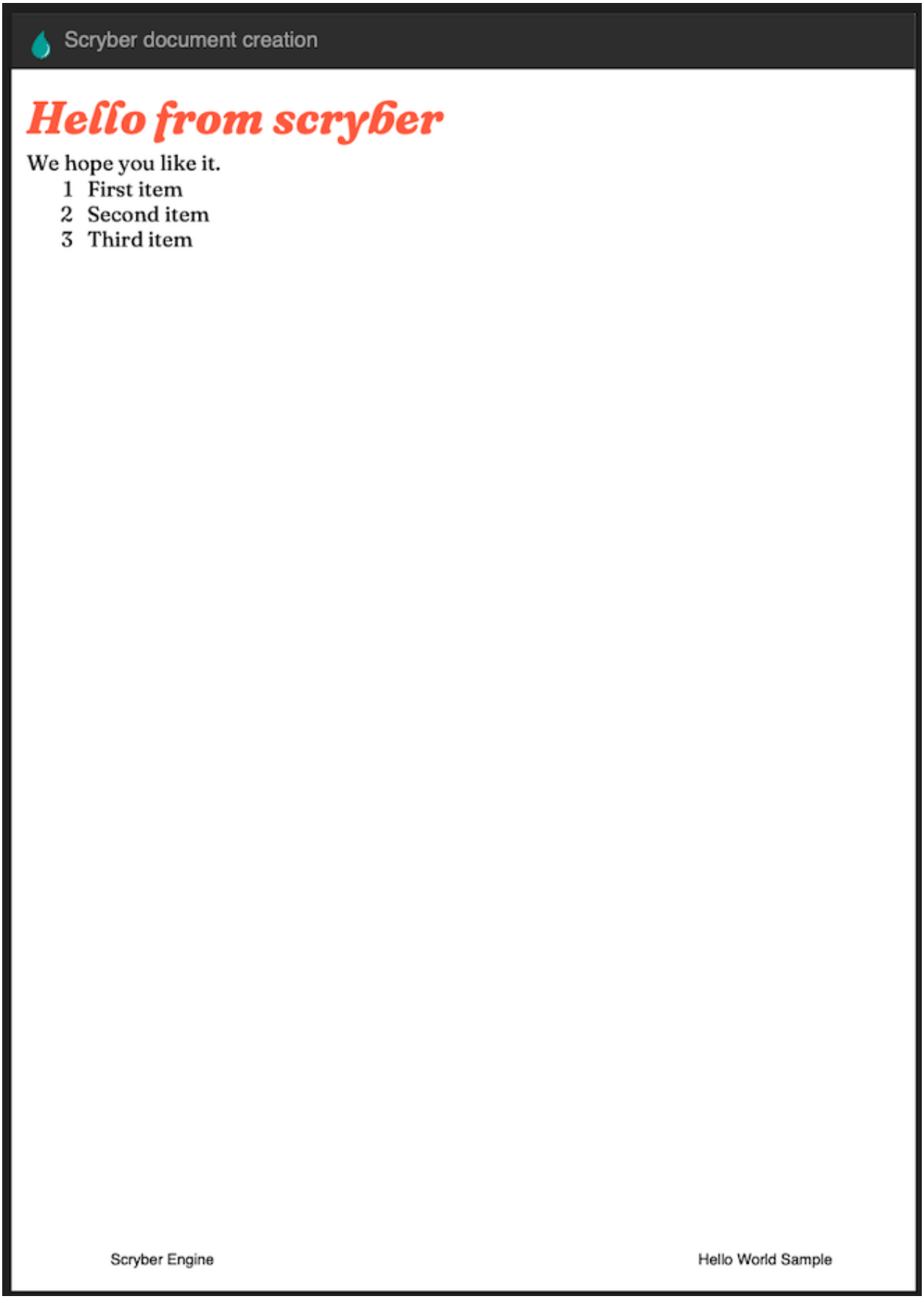
```
        <style>
            body {
                font-family: 'Fraunces', sans-serif;
                font-size: 14pt;
            }

            p.header {
                color: #AAA;
                background-color: #333;
                background-image: url('./html/images/ScyberLogo2_alpha_small.png');
                background-repeat: no-repeat;
                background-position: 10pt 10pt;
                background-size: 20pt 20pt;
                margin-top: 0pt;
                padding: 10pt 10pt 10pt 35pt;
            }


            .foot td {
                border: none;
                text-align: center;
                font-size: 10pt;
                margin-bottom: 10pt;
            }
        </style>
        <!-- Setting the base url for the references so they load the style sheet␣
↪background image from git -->
        <base href='https://raw.githubusercontent.com/richard-scryber/scryber.core/
↪master/Scryber.Core.UnitTest/Content/' />
    </head>
    <body>
        <header>
            <!-- document headers -->
            <p class="header">Scryber document creation</p>
        </header>
        <!-- support for many HTML5 tags-->
        <main style="padding:10pt">

            <!-- binding styles and values on content including expressions and␣
↪functions -->
            <h2 style="color:{{model.titlecolor}}">{{concat('Hello World, from ',
↪model.name)}}</h2>

            <div>We hope you like it.</div>

            <!-- Loop with nested item collection binding to the objects -->
            <ol>
                <template data-bind='{{model.items}}'>
                    <!-- binding within the model.items content, and can be nested -->
                    <li>{{.name}}</li>
                </template>
            </ol>

        </main>
        <footer>
            <!-- footers in a full width table with style -->
            <table class="foot" style="width:100%">
```

(continues on next page)

(continued from previous page)

```
                <tr>
                    <td>{{model.author}}</td>
                    <td>Hello World Sample</td>
                </tr>
            </table>
        </footer>
    </body>
</html>
```

The output from this is much more pleasing. Especially that Fruances font :-)


Scryber document creation
Hello from scryber
We hope you like it.
1 First item
2 Second item
3 Third item
Scryber Engine
Hello World Sample

10.2.9 1.2.9. Further reading

You can read more about the what you can do with scryber from the contents.

We have no idea what you will be able to create with scryber. It's just there to hopefully help you build amazing documents in an easy and repeatable way.

10.3 1.3. Why Scryber, and why not.

We love scyber, and we think it is one of the most flexible and fastest PDF document template generation engines out there.

In the GUI or MVC examples you have hopefully seen some of the power within scryber.

We developed it over a number of years, as there was always an end of project ask to create reports, and most of the alternative libraries were either too low level or too costly.

We really hope it meets the needs of many other developers to exceed the needs of their clients, without a huge learning curve, or overhead.

10.3.1 1.3.1 What scryber is not

The simplest way to put this is, scryber is not an web page printing engine.

It will not simply take an html page and render a PDF from it, there are many other packages; libraries and engines which can do this.

Scryber also does not fully support all css or html tags. Flexbox, :before pseudo elements, content properties, etc. Some are in the pipeline, some are in ideas and others will not make it.

However, we think, with scryber, you will have better control of layout, pagination, styles and content. The maintenance and separation of the data you are presenting with the design you want it to fit.

In the end we think it is faster and better

10.3.2 1.3.2. Using templates

A key feature of the scryber library is the use of templates. Separating the content from the data and the style your projects and solutions is always more manageable. Scryber benefits from this separation and allows designers to work with developers to build great documents.

10.3.3 1.3.3. Built for the cloud

Scryber can reference remote images, fonts, even inner templates and content from any base url. Including these resources is as simple as adding the url. Scryber will inteligently load these and use them in your layouts.

10.3.4 1.3.4. Reduced learning curve

Whilst scryber does not support the full set of css selectors, or relative dimensions, It does use standard capabilities with xhtml and css, that means you are up and running quickly and efficiently, and able to concentrate on the need, rather than coming up to speed.

10.3.5 1.3.5. Low code/No code

Using our expressions library and the handlebars '{{ }}' binding syntax it is easy to create your templates and build out your documents in a repeatable manner that des not rely on the writing of reams of code to build a document. It supports the use of functions to calculate values or make decisions in your templates.

But, we do of course fully support the use of code to create full documents or just partial components.

10.3.6 1.3.6. Full document lifecycle

Scryber was built from the ground up to create documents. It has a strong lifecycle model that allows for interaction and injection at any point to create the document you need. Along with this there is full support for tracing and logging, so the black box is more transparent.

10.3.7 1.3.7. Extensibility

Finally, scryber is completely extensible. The components we have built meet our needs, but if there is something it doesn't do, there is usually a way to make it work.

10.4 1.4. Scryber packages and the libraries - TD

It is not nescessary to know the structure of the scryber code, or how it processes a document into a PDF. But it helps in understanding what is going on under the hood, and also understanding the logs.

10.4.1 1.4.1. NuGet Packages

There are 3 NuGet packages for scryber.

The Scryber.Core.OpenType package contains a single library for parsing ttf (open type) font files.

The Scryber.Core package contains the main libraries for PDF generation.

The Scryber.Core.Mvc package contains the MVC extensions that allow for easy generation of you PDF from a web request.

10.4.2 1.4.2. Source code

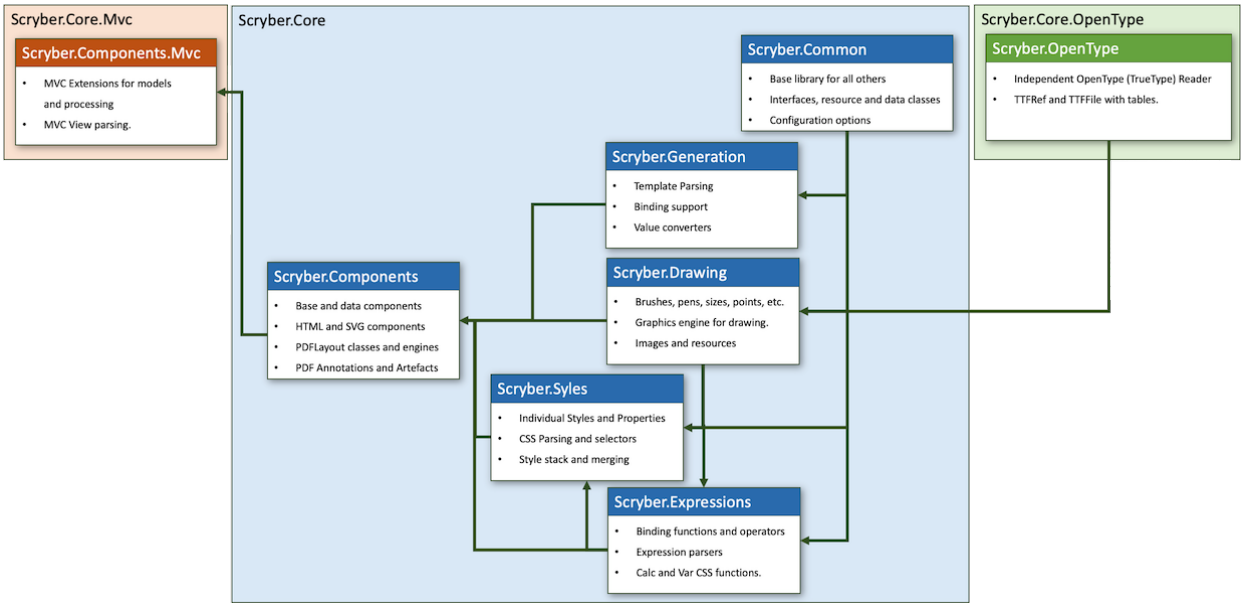
The Scryber.Core Git repository contains the open source code, you are at liberty to use in your own projects for if wanted.

It also contains the samples for this documentation in the *Scryber.UnitSamples* project of the source.

10.4.3 1.4.3. Scryber.Core libraries

Within the core package are 6 main libraries with the top level library Scryber.Components referencing others, and the Scryber.Common containing most of the interfaces and base structures.

Full size version



10.4.4 1.4.4 Useful Namespaces

When building documents, knowing the namespaces for Components and content is useful. As an overview these are the main namespaces in each of the libraries, anong with the purpose of the classes in that namespace.

The Scryber.Common library

The base library for all other libraries

1. Scryber - Contains the core interfaces, attributes, events and handler delegate definitions.
2. Scryber.Logging - Contains specific implementations of the Scryber logging infrastructure
3. Scryber.Options - Classes for the configuration of scryber.
4. Scryber.Native - Classes for the reading and parsing of an existing PDF file.
5. Scryber.Caching - Base classes and interfaces for data caching.
6. Scryber.Utility - Helper classes for paths, data load; numbering and types; frameworks and versions.

The Scryber.Drawing library

Contains all the drawing structures and classes for units, points, rects, thicknesses and colors, along with text, imaging and resources.

1. Scryber - The interfaces specific to drawing, and the PDFxxxOptions that are built from styles.
2. Scryber.Drawing - THe main graphics classes with units, points, rects, thicknesses, colors, gradients, dashes, fonts, brushes and pens.
3. Scryber.Drawing.Imaging - The classes that understand different pixel image formats.
4. Scryber.Options - Classes for configuration of the drawing library.
5. Scryber.Resources - Classes that are shared resources in a document.
6. Scryber.Text - Classes associated with reading textual content and encodings.

The Scryber.Expressive library

Contains all the expression parsing and evaluation classes for the {{handlebars}} syntax. This is based on the source from Shaun Lawrence in the **'https://github.com/bijington/expressive'_** repository

1. Scryber.Expressive - Root namespace for the parsers, top expression class.
2. Scryber.Expressive.Tokenisation - Conversion of expression strings to tokens.
3. Scryber.Expressive.Funtions - All available function classes.
4. Scryber.Expressive.Operators - All available operator classes.
5. Scryber.Expressive.Expressions - All expression classes that operators build.
6. Scryber.Expressive.Helpers - Type conversion and number calculation methods.

The Scryber.Generation library

Contains all functionality to read from streams and files and parse into instances and objects.

1. Scryber - Interfaces, enumerations and delegates that are commonly used for generation.
2. Scryber.Generation - The main classes for parsing content and building instances based on reflection of attributes.
3. Scryber.Binding - The factories and classes for creating the expressions for binding in a document.

The Scryber.Styles library

Contains all functionality for parsing, building, merging and referencing styles.

1. Scryber - Interfaces, enumerations and delegates that are commonly used for styles.
2. Scryber.Styles - The main classes for style properties, the keys they are stored with and values assigned to them.
3. Scryber.Styles.Parsing - The classes for converting css to styles and their definitions.
4. Scryber.Styles.Selectors - The classes for css Selectors and matching to components.

The Scryber.Components library

10.5 1.5. Using the documentation Samples

All the samples in the documentation are available in the source code to follow along with.

Scryber.Core Git repository

They are located as unit tests, so can be run individually, or as a group / whole in the project Scryber.UnitSamples

They will load any needed templates from the project folder /Templates/[category]/[filename.html]

And they will save files to the output /My Documents/Scryber Test Output/[category]/[filename.pdf]

10.5.1 1.5.1. Sample test base class

To reduce the boilerplate code the methods `GetTemplatePath` and `GetOutputStream` have been set up on a base class called `TestBase` that all sample tests inherit from.

```
#define OUTPUT_FILES

using System;
using System.IO;

```

(continues on next page)

(continued from previous page)

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Scryber.UnitSamples
{
    public class SampleBase
    {
        public const Environment.SpecialFolder baseOutput = Environment.SpecialFolder.
↪MyDocuments;

        public const string SampleOutputFolder = "Scryber Test Output";
        public const string TemplatesFolder = "../../../Templates/";

        /// <summary>
        /// Gets a new output stream to a file with the path
        /// </summary>
        /// <param name="category"></param>
        /// <param name="fileNameWithExtension"></param>
        /// <returns></returns>
        protected static Stream GetOutputStream(string category, string␣
↪fileNameWithExtension)
        {

#if OUTPUT_FILES

            //We are actually outputting to a file on the test machine.
            var path = System.Environment.GetFolderPath(baseOutput);

            if (!Directory.Exists(path))
                throw new DirectoryNotFoundException("The special folder directory "␣
↪+ baseOutput.ToString() + " does not exist");

            path = Path.Combine(path, SampleOutputFolder);

            if (!Directory.Exists(path))
                Directory.CreateDirectory(path);

            path = Path.Combine(path, category);

            if (!Directory.Exists(path))
                Directory.CreateDirectory(path);

            var output = Path.Combine(path, fileNameWithExtension);

            return new FileStream(output, FileMode.Create);

#else

            //No output wanted so just use a memory stream.
            var ms = new MemoryStream();
            return ms;

#endif
        }


        /// <summary>
        /// Gets the full path to the template specified, and (by default) checks to␣
↪make sure the file exists.
```

(continues on next page)

(continued from previous page)

```
        /// </summary>
        /// <param name="category">The category grouping (inner folder for the␣
↪template). If null or empty, then the template file will be assumed to be in the␣
↪root TemplatesFolder</param>
        /// <param name="fileNameWithExtension">The file name + extension for the␣
↪file e.g MySample.html</param>
        /// <param name="assertExists">Optional but if true, a check will be made to␣
↪ensure the file actually exists before returning</param>
        /// <returns>The full path to the template specified</returns>
        protected static string GetTemplatePath(string category, string␣
↪fileNameWithExtension, bool assertExists = true)
        {
            var path = System.Environment.CurrentDirectory;

            if (string.IsNullOrEmpty(category))
                path = Path.Combine(path, TemplatesFolder, fileNameWithExtension);
            else
                path = Path.Combine(path, TemplatesFolder, category,␣
↪fileNameWithExtension);

            //Clean the path
            path = Path.GetFullPath(path);

            if (assertExists)
                Assert.IsTrue(File.Exists(path), "The sample file at path '" + path +
↪"' does not exist");

            return path;
        }
    }
}
```

10.5.2 1.5.2 Changing the defaults

The pre-defined values for the output folder, the location of the templates folder, and the 'SpecialFolder' where the output will be saved can be modified to alter location either if you are experiencing dificulties in locating the samples or want to change where they will be created.

If you **do not** want to execute the tests to save to an actual file, the compiler directive OUTPUT_FILES can be removed (or commented)

10.5.3 1.5.3. Empty Sample Test class

A basic set up for a sample in a test class is

```
//Standard using namespaces

using Microsoft.VisualStudio.TestTools.UnitTesting;
using Scryber.Components;
using Scryber.Styles;
using Scryber.Drawing;

namespace Scryber.UnitSamples
{
```

(continues on next page)

(continued from previous page)

```
    //Inherits from Scryber.UnitSamples.SampleBase

    [TestClass]
    public class MySamples : SampleBase
    {
        //Declare a test method

        [TestMethod]
        public void SimpleSample()
        {
            //Get the path to the template
            var path = GetTemplatePath("Samples", "Simple.html");

            //Parse the document at the path
            using (var doc = Document.ParseDocument(path))
            {
                //do any further processing needed

                //Create an output stream
                using(var stream = GetOutputStream("Samples", "Simple.pdf"))
                {
                    //And save the document to that file
                    doc.SaveAsPDF(stream);
                }

            }
        }
    }
}
```

10.5.4 1.5.4. Contributing examples

We would love to add more samples and starter documents / recipies.

If you have an example you are proud of, or think would be useful to others. Please **do** fork the repository and propose the additions.

And if you find a *bug* please let us know, or even fork and fix.

10.6 1.6. Parsing documents from content

When you parse the contents of an file, or a stream or a reader, Scryber builds a full object model of the content plus any referenced content. As the parser is based around XML it is important that all content is valid - it does not like unclosed tags or elements.

10.6.1 1.6.1. Content namespaces

As the most basic example

```
<!-- /Templates/Overview/SimpleParsing.html -->

<!DOCTYPE HTML >
```

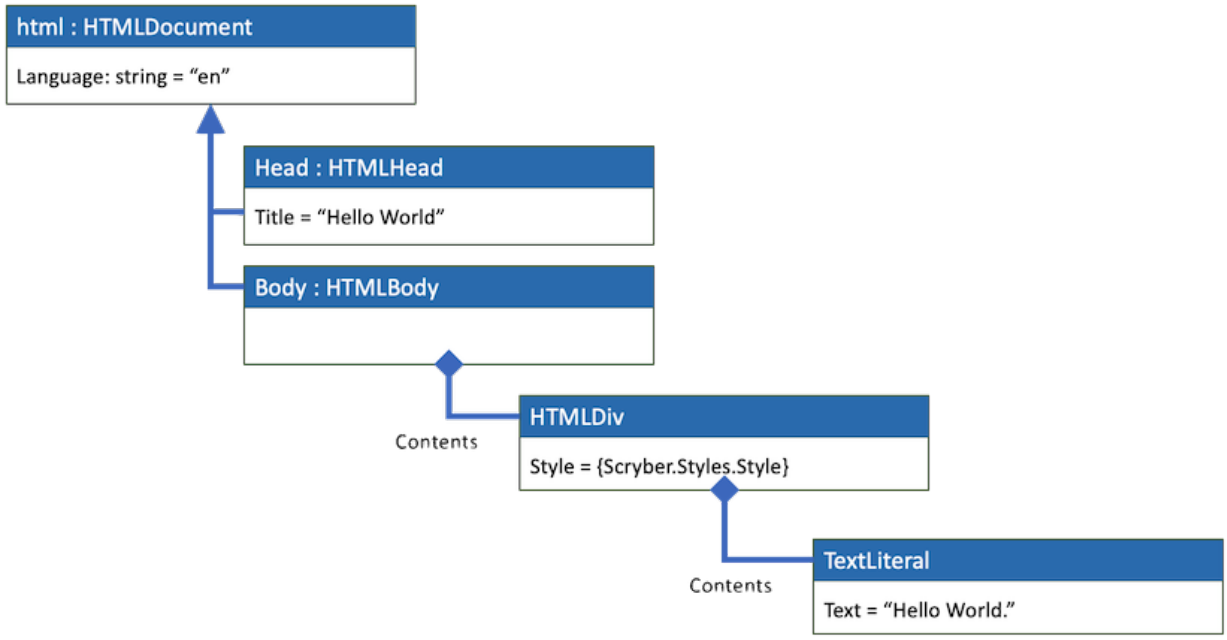
(continues on next page)

(continued from previous page)

```
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <div style='padding:10px'>Hello World.</div>
    </body>
</html>
```

```
//Scryber.UnitSamples/OverviewSamples.cs

public void SimpleParsing()
{
    var path = GetTemplatePath("Overview", "SimpleParsing.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Overview", "SimpleParsing.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Would be parsed into the following Document Object Model

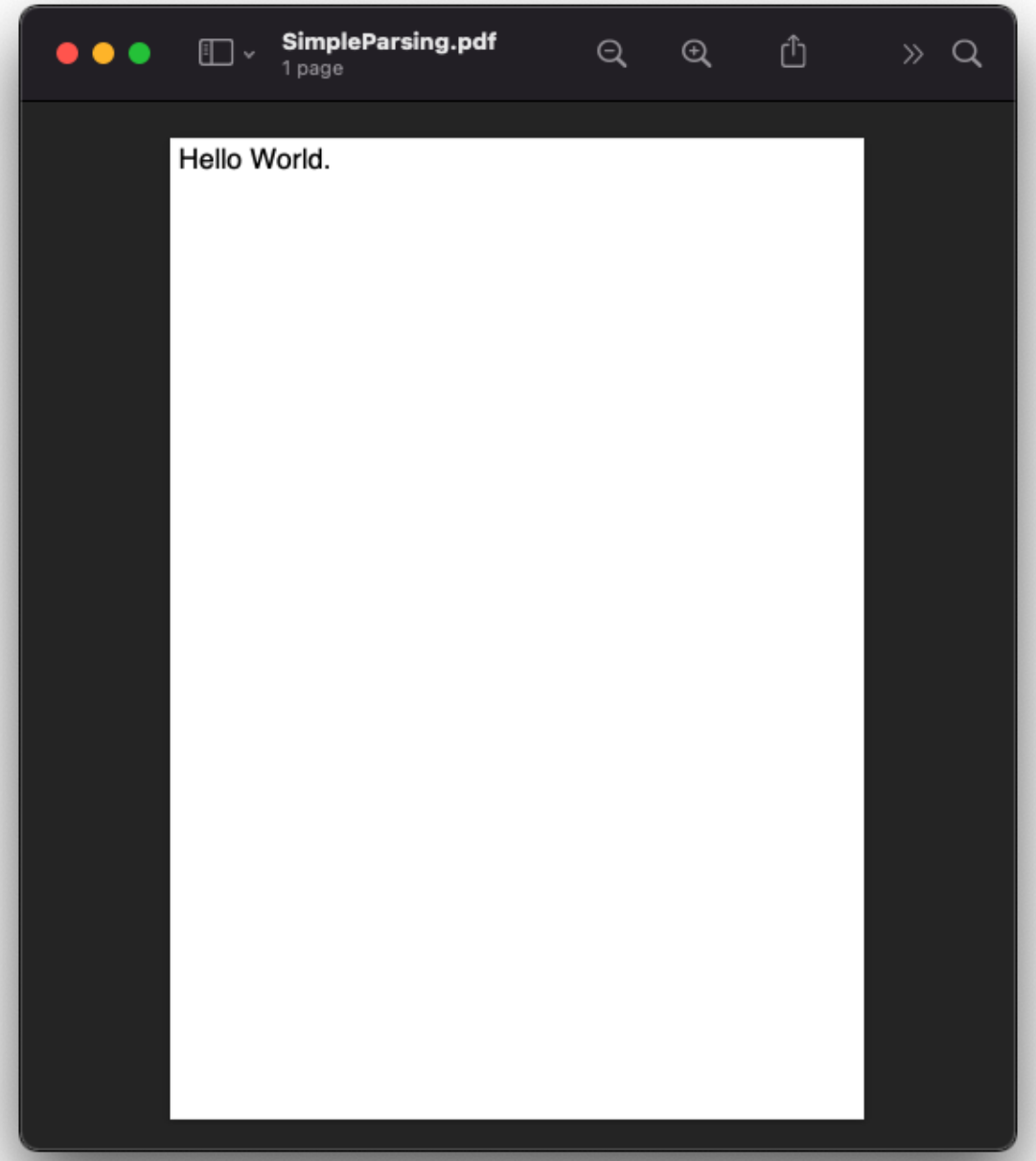


Full size version

And the output would be

Full size version

The namespace for an element must be known to the parser. For most XHTML templates this will be the standard XML


SimpleParsing.pdf
1 page
Hello World.

Namespace (xmlns) http://www.w3.org/1999/xhtml This namespace is mapped directly onto the library assembly and namespace `Scryber.Html.Components, Scryber.Components`

In the library there is a class called `HTMLDocument` that is decorated with the `PDFParsableComponent` attribute with a name of 'html'. This is how the parser knows that when it sees an XML element called *html* it should create an instance of the `Scryber.Html.Components.HTMLDocument` class.

This class has a couple of properties on it for *Head* and *Body* that are decorated with the `PDFElement` attribute with names *head* and *body* respectively. So the parser knows when it reads elements with this name, the values should be set as instances of the classes `HTMLHead` and `HTMLBody`.

It also has an attribute for the *lang* value that will be set.

```
namespace Scryber.Html.Components
{
    [PDFParsableComponent("html")]
    public class HTMLDocument : Document
    {
        [PDFElement("head")]
        public HTMLHead Head
        {
            get;
            set;
        }

        [PDFElement("body")]
        public HTMLBody Body
        {
            get ;
            set ;
        }

        [PDFAttribute("lang")]
        public string Language
        {
            get;
            set;
        }

        .
        .
        .
    }
}
```

And so it goes on into the rest of the xml, reading elements and attributes, and trying to set the values to components or property values.

10.6.2 1.6.2. Parsing documents from files

The easiest way to parse any xml content is to use the various static methods on the `Scryber.Components.Document` class.

There are 2 variants called `ParseDocument` and `Parse`.

`ParseDocument` has 6 overloads and the content parsed must have a root object that is (or inherits from) `Scryber.Components.Document` The simplest is to load directly from a file

```
//using Scryber.components

string filepath = GetPathToFile();
var doc = Document.ParseDocument(filepath);
```

This reads the file from the stream and will resolve any references to relative content (images, stylesheets, etc) based on the *filepath*.

10.6.3 1.6.3. Parsing documents from streams

If you want to load content dynamically from a stream then you can use the overloads that take a stream. An enumeration value for ParseSourceType must be provided, and an optional path value, so the parser can know where other references may reside.

```
//from a stream with no references
using(var stream = GetMyDocumentContent())
{
    doc = Document.ParseDocument(stream, PaseSourceType.DynamicContent);
}
```

If the stream will contain relative path references to other content such as stylesheets or embedded content then a path should be provided. If no path is provided then content will be looked for relative to any basePath specified in the source stream. If no base path is provided then content will be looked for relative to the current executing assembly.

```
//from a stream where references are known to be stored
var path = "C:/MyFiles/BasePath";
using(var stream = GetMyDocumentContent())
{
    doc = Document.ParseDocument(stream, path, PaseSourceType.DynamicContent);
}
```

The options for the content can be any of the following.

- A `System.IO.Stream` or one of its sublcasses.
- A `System.IO.TextReader` or one of its subclasses.
- A `System.XML.XmlReader` or one of its subclasses.

Ultimately the content should be valid XML that can be read.

For example, using an XmlReader

```
//using System.Xml.Linq

//Scryber.UnitSamples/OverviewSamples.cs

public void XLinqParsing()
{

    XNamespace ns = "http://www.w3.org/1999/xhtml";

    var html = new XElement(ns + "html",
        new XElement(ns + "head",
            new XElement(ns + "title",
                new XText("Hello World"))
            ),
```

(continues on next page)

(continued from previous page)

```
        new XElement(ns + "body",
            new XElement(ns + "div",
                new XAttribute("style", "padding:10px"),
                new XText("Hello World."))
            )
        );

    using (var reader = html.CreateReader())
    {
        //passing an empty string to the path as we don't have images or other␣
↪references to load
        using (var doc = Document.ParseDocument(reader, string.Empty, ParseSourceType.
↪DynamicContent))
        {
            using (var stream = GetOutputStream("Overview", "XLinqParsing.pdf"))
            {
                doc.SaveAsPDF(stream);
            }
        }
    }
}
```

Or from a string itself

```
//using System.IO

//Scryber.UnitSamples/OverviewSamples.cs

public void StringParsing()
{
    var title = "Hello World";
    var src = @"<html xmlns='http://www.w3.org/1999/xhtml' >
            <head>
                <title>" + title + @"</title>
            </head>
            <body>
                <div style='padding: 10px' >" + title + @".</div>
            </body>
        </html>";

    using (var reader = new StringReader(src))
    {
        using (var doc = Document.ParseDocument(reader, string.Empty, ParseSourceType.
↪DynamicContent))
        {
            using (var stream = GetOutputStream("Overview", "StringParsing.pdf"))
            {
                doc.SaveAsPDF(stream);
            }
        }
    }
}
```

All 3 methods create exactly the same document. It also allows for building dynamic documents at runtime - but there are other ways :see::7_parameters_and_expressions

10.6.4 1.6.4. Building in code

The template parsing engine is both flexible and extensible, but it does not have to be used. Scryber components are **real** object classes, they have properties and methods along with inner collections.

We can just as easily create the document using a method.

```
//using Scryber.Components
//using Scryber.Drawing

//Scryber.UnitSamples/OverviewSamples.cs

protected Document GetHelloWorld()
{
    var doc = new Document();
    doc.Info.Title = "Hello World";

    var page = new Page();
    doc.Pages.Add(page);

    var div = new Div() { Padding = new PDFThickness(10) };
    page.Contents.Add(div);

    div.Contents.Add(new TextLiteral("Hello World"));

    return doc;
}


public void DocumentInCode()
{

    using (var doc = GetHelloWorld())
    {
        using (var stream = GetOutputStream("Overview", "CodedDocument.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

This works well, and may have benefits for your implementations, but ultimately could become very complex and difficult to maintain.

10.6.5 1.6.5. Embedding other content

Including content from other sources (files) is easy within the template by using the `<embed>` element with the `src` attribute set to the name of the source file. This can either be a relative or an absolute path to the content to be included.

```
<embed src='./fragments/tsandcs.html' />
```

The content will be loaded by the parser syncronously rather than later at load time, which is the case for css stylesheets and images. This is to ensure there is a full file content to be parsed.

The embedded content should be a fragment of valid xhtml / xml rather than a full html file. The namespaces are required in the embessed file, as well.

```
<!-- /Templates/Overview/Fragments/TsAndCs.html -->

<!-- Standard terms and conditions, with namespace -->
<div id='MyTsAndCs' xmlns='http://www.w3.org/1999/xhtml'>
    <p>1. We will look after you</p>
    <p>2. If you look after us</p>
</div>
```

```
<!DOCTYPE HTML>
<html lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Hello World</title>
</head>
<body>
    <div style='padding:10px'>Hello World.</div>
    <!-- embedded content within a div -->
    <div style="border:solid 1px black; margin:10pt; padding:5pt">
        <embed src="./fragments/tsandcs.html" />
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/OverviewSamples.cs

public void EmbedContent()
{
    var path = GetTemplatePath("Overview", "EmbeddedContent.html");

    using (var doc = Document.ParseDocument(path))
    {
        //Embedded content is loaded at parse time
        var embedded = doc.FindAComponentById("MyTsAndCs") as Div;
        Assert.IsNotNull(embedded);

        using (var stream = GetOutputStream("Overview", "EmbeddedContent.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

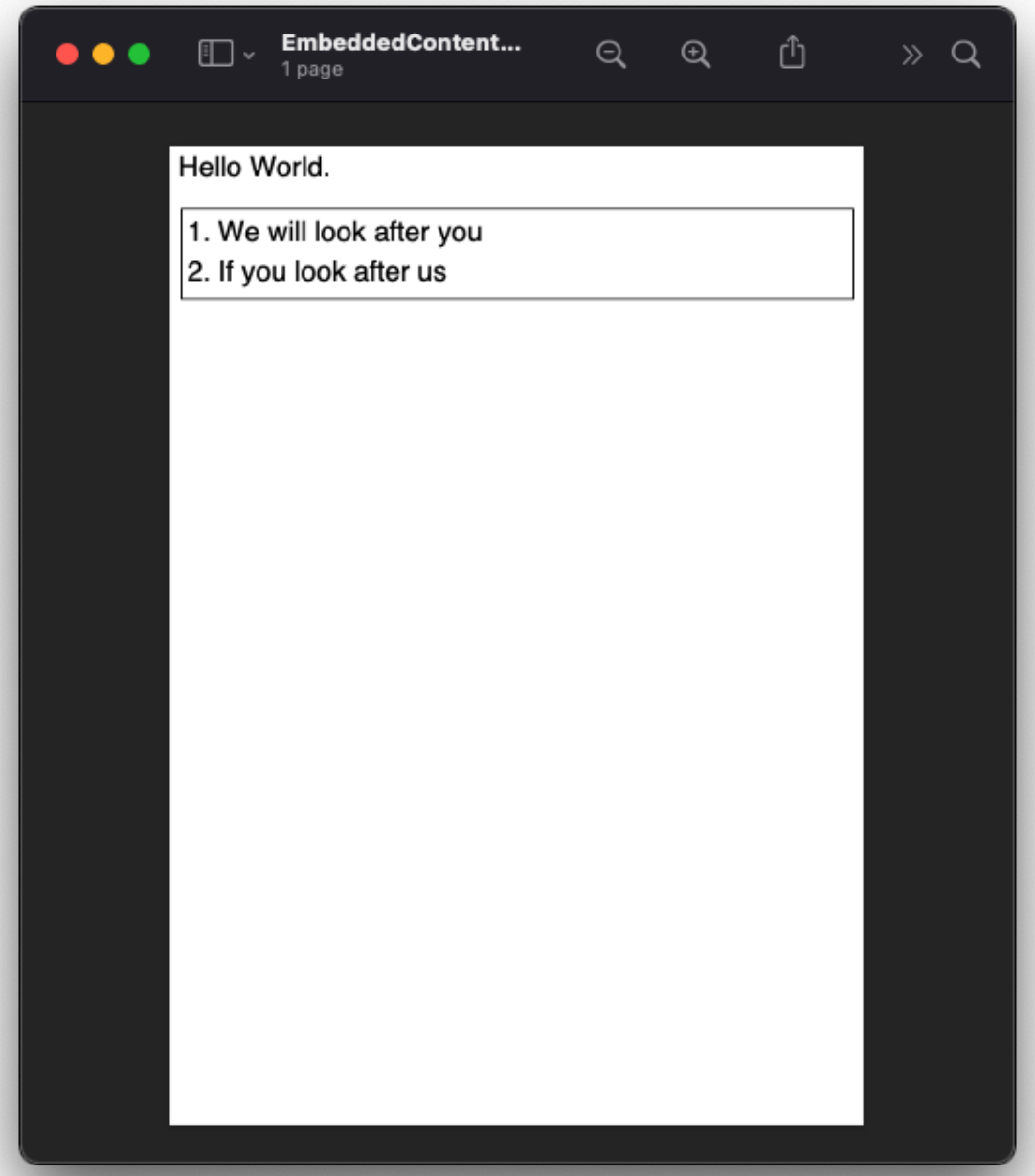
Full size version

When loading with relative references, the original path to the source file will be used to resolve the location of the embedded source. As with the examples above - if the content is being parsed dynamically, either the base path to the location sould be specified in the `ParseDocument` method, or a `PDFReferenceResolver` should be provided, as below.

10.6.6 1.6.6. Resolving references

The `Document.Parse` method, and its 12 overloads allows for parsing of any xml content as long as the root component returned implements the `IPDFComponent` interface.

If there are references to other content, that needs to be resolved at runtime it is also possible to pass a `PDFReferenceResolver` delegate to the parser so that your code can load it's own content and return it.


EmbeddedContent...
1 page
Hello World.
1. We will look after you
2. If you look after us

```
public delegate IPDFComponent PDFReferenceResolver(string filename, string xpath,␣
↪PDFGeneratorSettings settings);
```

This delegate will be called each time a remote reference is found, with the name of the file, and an optional xpath selector. It is upto the implementor to perform the parsing.

For example if we wanted to embed some standard content we could provide our own implementation.

```
private IPDFComponent CustomResolve(string filepath, string xpath,␣
↪PDFGeneratorSettings settings)
{
    if(filepath == "MyTsAndCs")
    {
        using(var tsAndCs = LoadTermsStream())
        {
            //We have our stream so just do the parsing again with the same settings
            return Document.Parse(filepath, tsAndCs, ParseSourceType.DynamicContent,␣
↪CustomResolve, settings);
        }
    }
    else
    {
        filepath = System.IO.Path.Combine(MyBasePath, filepath);
        return Document.Parse(filepath, CustomResolve, settings);
    }
}
```

And our document can reference the custom resolver with the

```
var doc = Document.Parse(string.Empty, reader, ParseSourceType.DynamicContent,␣
↪CustomResolve) as Document;
```

This will allow content from databases, or authenticated feeds to be added, or even transformed and added.

Note: Remember, the content to be parsed MUST be valid XML, including all XML namespaces, OR wrapped in an xml element.

It is also possible to return just coded objects in the return of the reference resolver, and the `PDFReferenceResolver` delegate can be any instance.

```
//using Scryber.Components
//using Scryber.Drawing

private IPDFComponent CustomResolve(string filepath, string xpath,␣
↪PDFGeneratorSettings settings)
{
    if(filepath == "MyTsAndCs")
    {
        var  p = new Paragraph(){ BackgroundColor = PDFColors.Aqua };
        p.Contents.Add(new PDFTextLiteral("These are my terms"));
        return p;
    }
    else
    {
        filepath = System.IO.Path.Combine(MyBasePath, filepath);
        return Document.Parse(filepath, CustomResolve, settings);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

10.6.7 1.6.7. Default namespaces

The html and svg namespaces are also automatically added.

- **http://www.w3.org/1999/xhtml**
  - The html components used in scryber. e.g. div, span, section etc.
  - It refers to the Scryber.Html.Components namespace in the Scryber.Components assembly (Version=1.0.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe)
- **http://www.w3.org/2000/svg**
  - The svg drawing components used in scryber. e.g. ellipse, circle, rect etc.
  - It refers to the Scryber.Svg.Components namespace in the Scryber.Components assembly (Version=1.0.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe)

See *Extending Scryber - TD* for more information on how to extend the namespaces used by the parser, and create your own components.

10.6.8 1.6.8. Further reading

- Learn about *1.7. Document parameters and binding* in the next section.
- For more about code vs templates see *XHTML Template*
- All the available components see *Components overview - TD*
- All the available styles see *Styles in your template - PD*
- Split your files? See 2_document/14_document_references for more on stylesheet links and embedding content.

10.7 1.7. Document parameters and binding

Within the content of a document the parser will look for expressions that will be evaluated at binding time into actual values. Every attribute in scryber, and all text can be bound with an expression.

The usual method for specifying these values uses the handlebars syntax - `{{expression}}`.

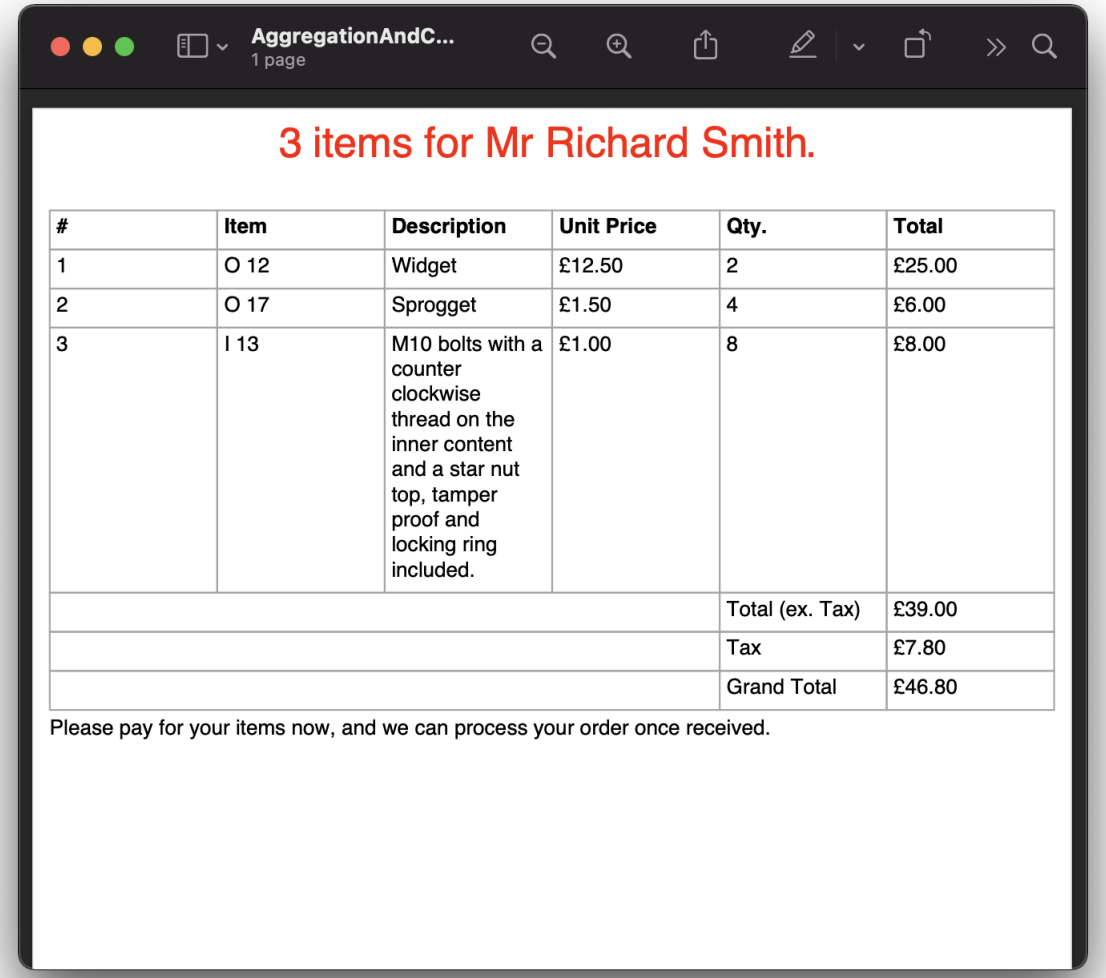
```
<div>{{concat('Hello ',model.userName)}}</div>
```

For styles and css the `calc()` method is extended to used expressions.

```
.banner {
    background-image: calc(model.logoSource);
}
```

This allows the inclusion of dynamic content at runtime either for specific values, for binding onto repeating content, or for evaluating expressions.

The values are passed to the document through the `Params` property of a document instance.



3 items for Mr Richard Smith.

#	Item	Description	Unit Price	Qty.	Total
1	O 12	Widget	£12.50	2	£25.00
2	O 17	Sprogget	£1.50	4	£6.00
3	I 13	M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included.	£1.00	8	£8.00
				Total (ex. Tax)	£39.00
				Tax	£7.80
				Grand Total	£46.80

Please pay for your items now, and we can process your order once received.

```
doc.Params["model"] = new {
    userName = "Richard",
    logoSource = "url(../images/mylogo.png)"
};
```

10.7.1 1.7.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in *1.5. Using the documentation Samples*

10.7.2 1.7.2 Simple Binding Example

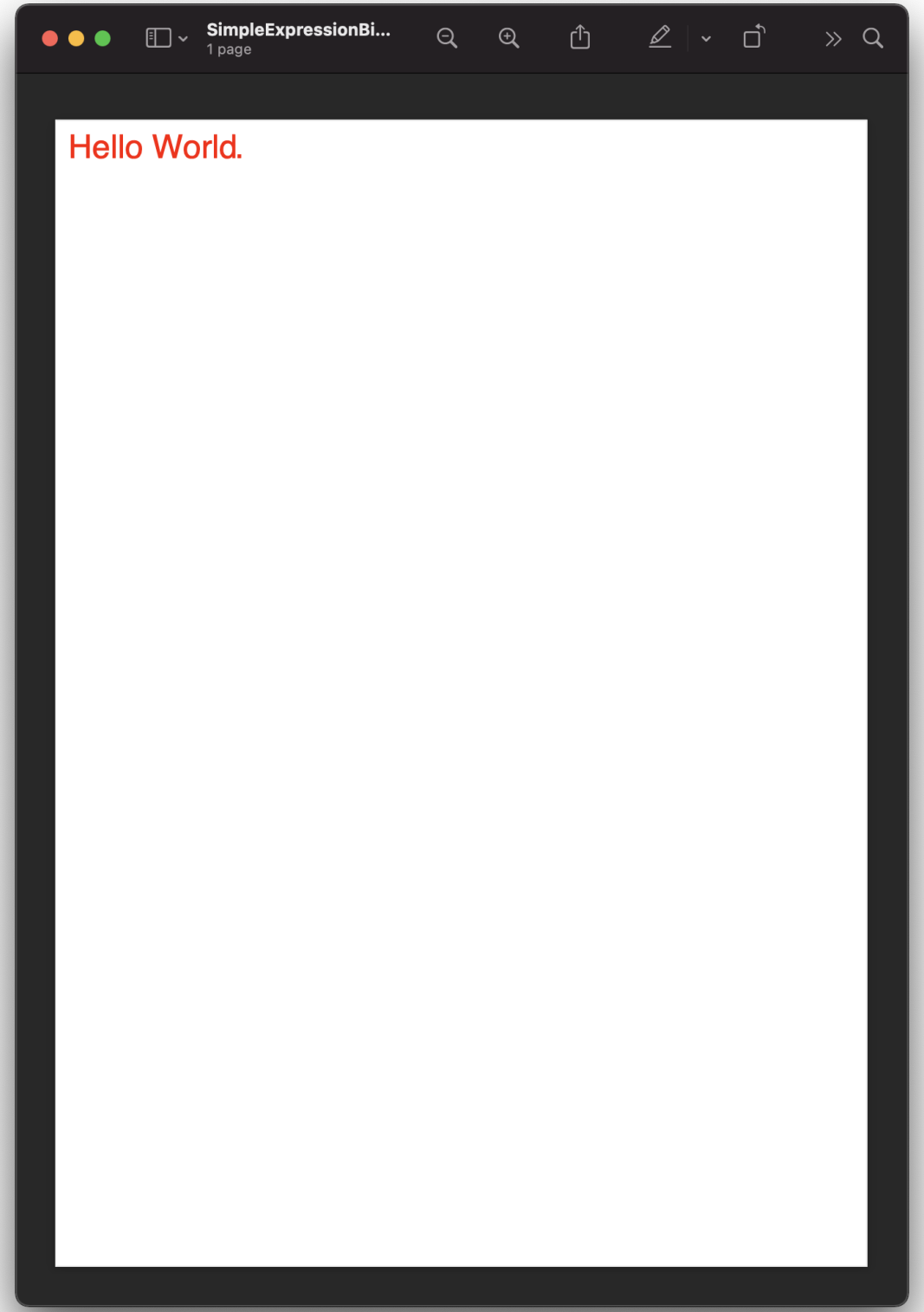
```
<!-- Templates/Overview/SimpleBinding.html -->

<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>{{title}}</title>
    </head>
    <body>
        <div style='color: #FF0000; padding: 10pt'>{{title}}.</div>
    </body>
</html>
```

When processing the document, the value for `title` can be provided.

```
//Scryber.UnitSamples/OverviewSamples.cs

public void SimpleBinding()
{
    var path = GetTemplatePath("Overview", "SimpleBinding.html");

    using (var doc = Document.ParseDocument(path))
    {
        doc.Params["title"] = "Hello World";

        //Before databinding - value is not set
        Assert.IsNull(doc.Info.Title);

        using (var stream = GetOutputStream("Overview", "SimpleBinding.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

        //After databinding
        Assert.AreEqual("Hello World", doc.Info.Title);
    }
}
```

At generation time these values will be interpeted and set on the appropriate properties and rendered to the file. As the layout has not executed before the databind, the content will be flowed with the rest of the document.

Full size version


SimpleExpressionBi...
1 page
Hello World.

Note: Scryber is strongly typed. It will try and convert or parse the values on databinding, and most of the style values and properties can be parsed. But the content should be of the correct type.

10.7.3 1.7.3. Complex expressions

As you can imagine the parameters could start to get unmanageable and complex. Thankfully the support for expressions allows both interrogation and calculation.

It is possible to use both strongly typed or dynamic objects (or a combination of both) for parameters. And expressions support any depth of property, and also an indexor in brackets. For example the following are all supported.

```
model.property
model.property[index]
model.property[function()].name
```

The classes can be dynamic or strongly typed but the properties are **Case Sensitive** to ensure language compatibility. If properties are not found, then the whole expression will return null.

10.7.4 1.7.4. Binding to complex objects

We can add both, a strongly typed user in the model, and also a dynamic theme object.

```
//Scryber.UnitSamples/OverviewSamples.cs

public class User {

    public string Salutation {get;set;}

    public string FirstName {get;set;}

    public string LastName {get;set;}
}


public void ComplexBinding()
{
    var path = GetTemplatePath("Overview", "ComplexBinding.html");

    using (var doc = Document.ParseDocument(path))
    {
        var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName =
↪"Smith" };

        doc.Params["model"] = new
        {
            user = user
        };
        doc.Params["theme"] = new
        {
            color = "#FF0000",
            space = "10pt",
            align = "center"
        };
```

(continues on next page)

(continued from previous page)

```
        using (var stream = GetOutputStream("Overview", "ComplexBinding.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

Our template can then access the properties on each of these objects. It can either be used in a function e.g. `{{concat()}}` or as a direct value `{{model.user.FirstName}}` For styles, the handlebars syntax is supported, but also the `calc()` css function.

```
<!-- Templates/Overview/ComplexBinding.html -->

<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>{{concat('Hello ', model.user.FirstName)}}</title>
    </head>
    <body>
        <div style='color: calc(theme.color); padding: calc(theme.space); text-align:␣
↪calc(theme.align)'>
            Hello {{model.user.FirstName}}.
        </div>
    </body>
</html>
```

And the output as below.

Full size version

10.7.5 1.7.5. Looping over collections

Along with the interrogation of the object properties scryber supports the enumeration over collections using the `<template />` tag. To set the value of the item or items to loop over use the `data-bind` attribute.

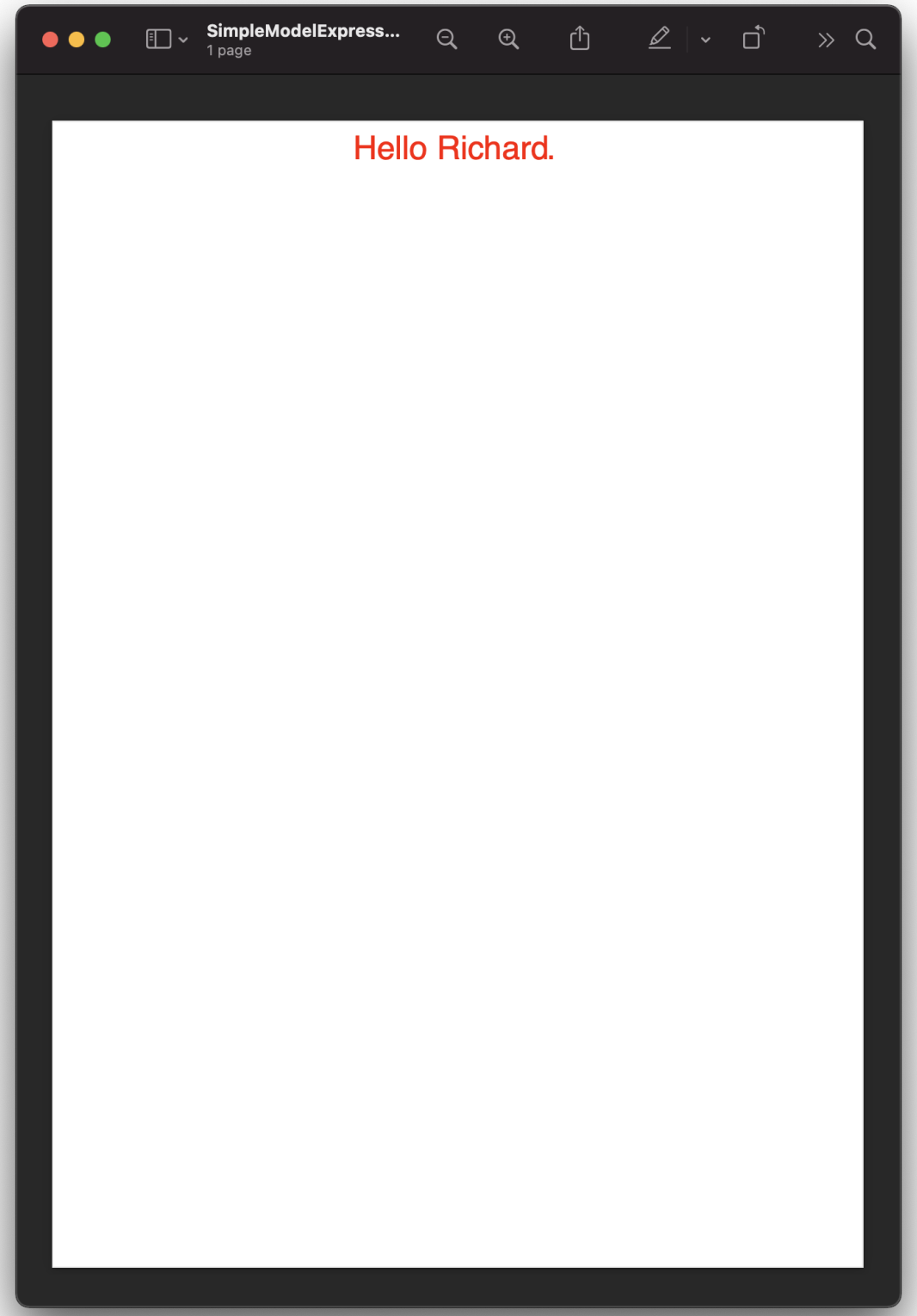
```
<template data-bind='{{ expression }}'>
    <!-- any inner content --->
</template>
```

Inside the template the current item can be referred to using the dot prefix `.property`. And the zero based index of the current loop is accessible with the `index()` function.

If we add 2 more model classes and a mock service to our code.

```
//Scryber.UnitSamples/OverviewSamples.cs

public class Order {

    public int ID {get;set;}

    public string CurrencyFormat {get;set;}

    public double TaxRate {get;set;}

    public double Total {get;set;}
```

(continues on next page)


SimpleModelExpress...
1 page
Hello Richard.

(continued from previous page)

```
    public List<OrderItem> Items {get;set;}
}


public class OrderItem{

    public string ItemNo {get;set;}

    public string ItemName {get;set;}

    public double Quantity {get;set;}

    public double ItemPrice {get;set;}

}


public class OrderMockService {

    public Order GetOrder(int id)
    {
        var order = new Order() { ID = id, CurrencyFormat = "£##0.00", TaxRate = 0.2 }
↪;
        order.Items = new List<OrderItem>(){
            new OrderItem() { ItemNo = "O 12", ItemName = "Widget", Quantity = 2,␣
↪ItemPrice = 12.5 },
            new OrderItem() { ItemNo = "O 17", ItemName = "Sprogget", Quantity = 4,␣
↪ItemPrice = 1.5 },
            new OrderItem() { ItemNo = "I 13", ItemName = "M10 bolts with a counter␣
↪clockwise thread on the inner content and a star nut top, tamper proof and locking␣
↪ring included.", Quantity = 8, ItemPrice = 1.0 }
        };
        order.Total = (2.0 * 12.5) + (4.0 * 1.5) + (8 * 1.0);

        return order;
    }

}
```

We can then set the `order` property on our model.

```
//Scryber.UnitSamples/OverviewSamples.cs

public void LoopBinding()
{
    var path = GetTemplatePath("Overview", "LoopBinding.html");

    using (var doc = Document.ParseDocument(path))
    {
        var service = new OrderMockService();
        var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName =
↪"Smith" };
        var order = service.GetOrder(1);

        doc.Params["model"] = new
        {
```

(continues on next page)

(continued from previous page)

```
            user = user,
            order = order
        };

        doc.Params["theme"] = new
        {
            color = "#FF0000",
            space = "10pt",
            align = "center"
        };

        using (var stream = GetOutputStream("Overview", "LoopBinding.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

In our template we can then **bind** the values in a table, looping over each one in a table body using the `template` element and a `data-bind` value.

```
<!-- Templates/Overview/LoopBinding.html -->

<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>{{concat('Hello ', model.user.FirstName)}}</title>
    </head>
    <body>
        <div style='color: calc(theme.color); padding: calc(theme.space); text-align:␣
↪calc(theme.align)'>
            Hello {{model.user.FirstName}}.
        </div>
        <div style='padding: 10pt; font-size: 12pt'>
            <table style='width:100%'>
                <thead>
                    <tr>
                        <td>#</td>
                        <td>Item</td>
                        <td>Description</td>
                        <td>Unit Price</td>
                        <td>Qty.</td>
                        <td>Total</td>
                    </tr>
                </thead>
                <tbody>
                    <!-- Binding on each of the items in the model.order -->
                    <template data-bind='{{model.order.Items}}'>
                        <tr>
                            <!-- The indexing of the loop + 1 -->
                            <td>{{index() + 1}}</td>
                            <td>{{.ItemNo}}</td>
                            <td>{{.ItemName}}</td>
                            <td>
                                <!-- we use a number tag to specify the data-format␣
↪referring to the top model -->
```

(continues on next page)

(continued from previous page)

```
                                <num value='{{.ItemPrice}}' data-format='{{model.
↪order.CurrencyFormat}}' />
                            </td>
                            <td>{{.Quantity}}</td>
                            <td>
                                <num value='{{.ItemPrice * .Quantity}}' data-format='{
↪{model.order.CurrencyFormat}}' />
                            </td>
                        </tr>
                    </template>
                </tbody>
            </table>
        </div>
    </body>
</html>
```

Full size version

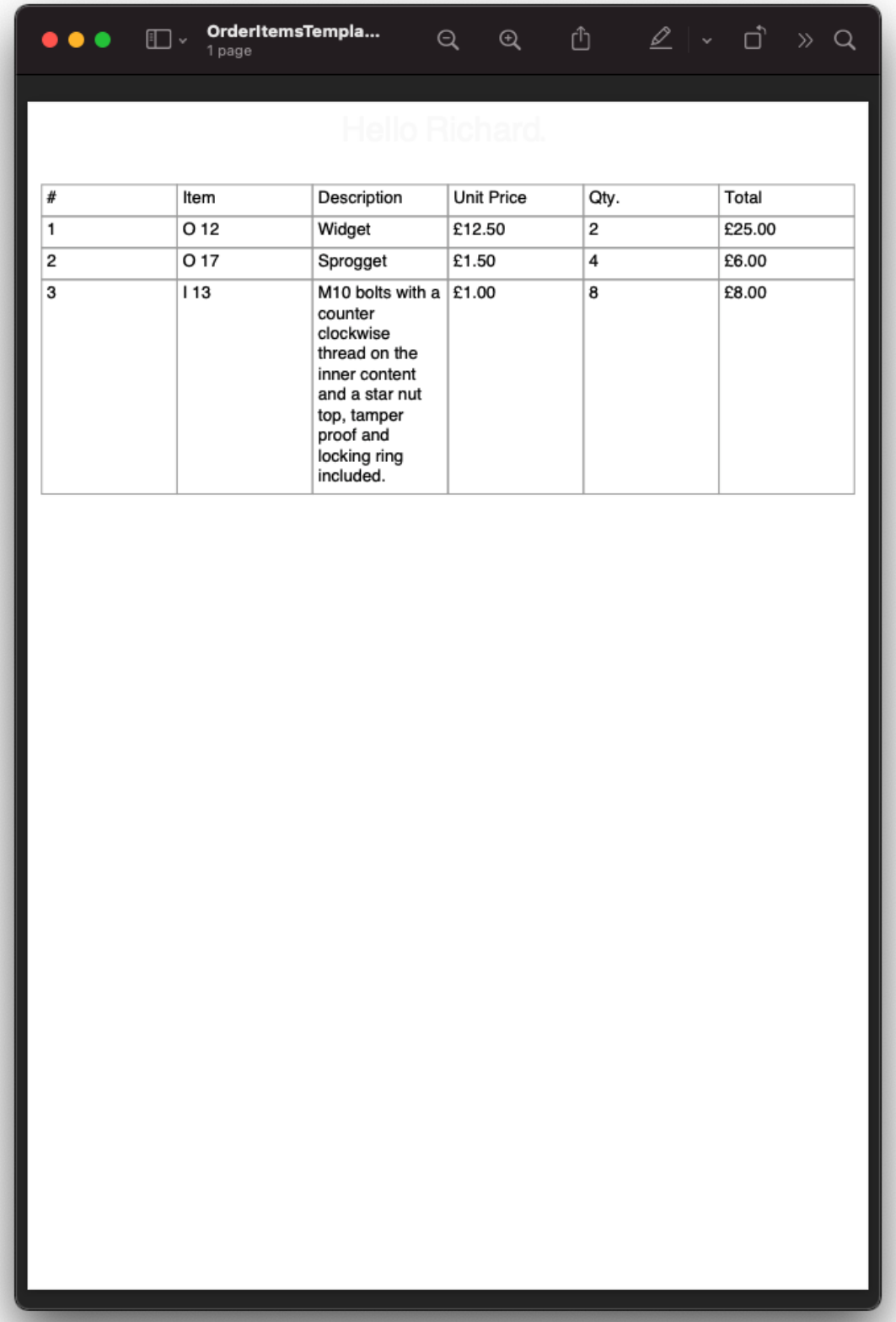
10.7.6 1.7.6. Showing and hiding content

Scryber supports visual changes to the content based on decisions in the data. The use of the css style `display:none` is supported, and evaluated at layout time. Scyber also supports the standard html `hidden='hidden'` flag on tags, or a boolean `visible` attribute. The advantage of the hidden/visible attributes are that they are explict rather than in the style, and easier to see in calculations.

If we extend our `Order` class we can use a comparison expression to show or hide some content within the template. And set the value in the document generation...

```
//Scryber.UnitSamples/OverviewSamples.cs

public class OrderWithTerms : Order
{
    public int PaymentTerms { get; set; }

}

public class OrderMockService2
{
    public Order GetOrder(int id)
    {
        //Use the order with terms
        var order = new OrderWithTerms() { ID = id, CurrencyFormat = "£##0.00",␣
↪TaxRate = 0.2 };
        order.Items = new List<OrderItem>(){
            new OrderItem() { ItemNo = "O 12", ItemName = "Widget", Quantity = 2,␣
↪ItemPrice = 12.5 },
            new OrderItem() { ItemNo = "O 17", ItemName = "Sprogget", Quantity = 4,␣
↪ItemPrice = 1.5 },
            new OrderItem() { ItemNo = "I 13", ItemName = "M10 bolts with a counter␣
↪clockwise thread on the inner content and a star nut top, tamper proof and locking␣
↪ring included.", Quantity = 8, ItemPrice = 1.0 }
        };
        order.Total = (2.0 * 12.5) + (4.0 * 1.5) + (8 * 1.0);
        //and set the payment terms
        order.PaymentTerms = 30;
```

(continues on next page)


OrderItemsTempla...
1 page
Hello Richard,
Item Description Unit Price Qty. Total
1 O 12 Widget £12.50 2 £25.00
2 O 17 Sprogget £1.50 4 £6.00
3 I 13 M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included. £1.00 8 £8.00

(continued from previous page)

```
        return order;
    }

}

public void ChoicesBinding()
{
    var path = GetTemplatePath("Overview", "ChoicesBinding.html");

    using (var doc = Document.ParseDocument(path))
    {
        //Use mock service 2
        var service = new OrderMockService2();

        var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName =
↪"Smith" };
        var order = service.GetOrder(1);

        doc.Params["model"] = new
        {
            user = user,
            order = order
        };

        doc.Params["theme"] = new
        {
            color = "#FF0000",
            space = "10pt",
            align = "center"
        };

        using (var stream = GetOutputStream("Overview", "ChoicesBinding.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

We can then change the output based upon the PaymentTerms value directly in the template using the `if` function.

```
hidden='{{if(model.order.PaymentTerms &lt; 0, "", "hidden")}}'
```

We can check the payment terms value and show or hide some content based on this.

```
<!-- Templates/Overview/ChoicesBinding.html -->

<!DOCTYPE HTML >
<html lang='en' xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>{{concat('Hello ', model.user.FirstName)}}</title>
    </head>
    <body>
        <div style='color: calc(theme.color); padding: calc(theme.space); text-align:␣
↪calc(theme.align)'>
            Hello {{model.user.FirstName}}.
        </div>
        <div style='padding: 10pt; font-size: 12pt'>
```

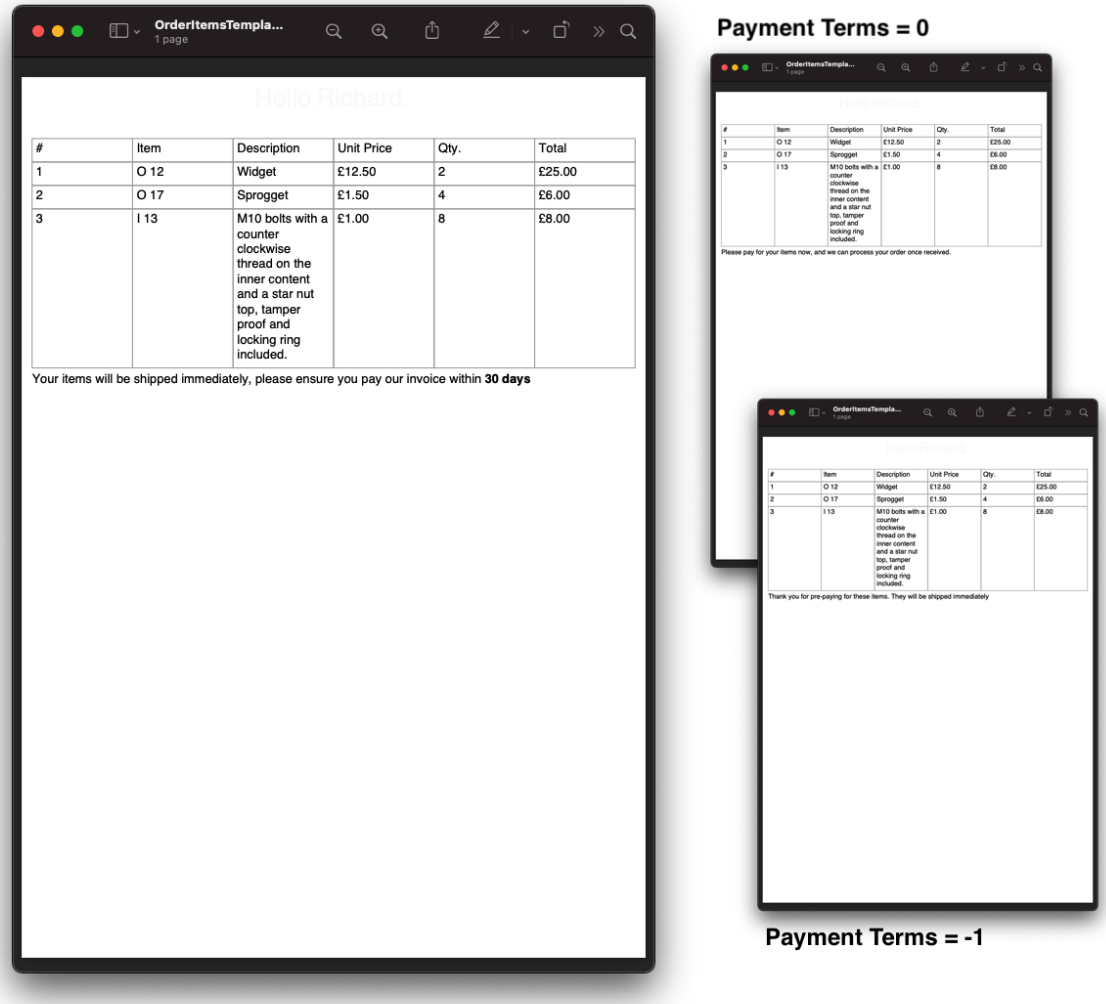
(continues on next page)

(continued from previous page)

```
            <table style='width:100%'>
                <thead>
                    <tr>
                        <td>#</td>
                        <td>Item</td>
                        <td>Description</td>
                        <td>Unit Price</td>
                        <td>Qty.</td>
                        <td>Total</td>
                    </tr>
                </thead>
                <tbody>
                    <!-- Binding on each of the items in the model.order -->
                    <template data-bind='{{model.order.Items}}'>
                        <tr>
                            <!-- The indexing of the loop + 1 -->
                            <td>{{index() + 1}}</td>
                            <td>{{.ItemNo}}</td>
                            <td>{{.ItemName}}</td>
                            <td>
                                <!-- we use a number tag to specify the data-format␣
↪referring to the top model -->
                                <num value='{{.ItemPrice}}' data-format='{{model.
↪order.CurrencyFormat}}' />
                            </td>
                            <td>{{.Quantity}}</td>
                            <td>
                                <num value='{{.ItemPrice * .Quantity}}' data-format='{
↪{model.order.CurrencyFormat}}' />
                            </td>
                        </tr>
                    </template>
                </tbody>
            </table>
            <div id='terms'>
                <div id='paidAlready' hidden='{{if(model.order.PaymentTerms &lt; 0, "
↪", "hidden")}}' >
                    <p>Thank you for pre-paying for these items. They will be shipped␣
↪immediately</p>
                </div>
                <div id='payNow' hidden='{{if(model.order.PaymentTerms == 0, "",
↪"hidden")}}'>
                    <p>Please pay for your items now, and  we can process your order␣
↪once received.</p>
                </div>
                <div id='payLater' hidden='{{if(model.order.PaymentTerms &gt; 0, "",
↪"hidden")}}'>
                    <p>Your items will be shipped immediately, please ensure you pay␣
↪our invoice within <b> {{model.order.PaymentTerms}} days</b></p>
                </div>
            </div>
        </div>
    </body>
</html>
```

Note: Because we are valid xhtml/xml we must escape the < and > calculations as < and > respectively. The

parser will convert them back within the calculation.



Full size version

With the above example, our service instance has changed, our template has been adapted, but there is no need to update any other code. This flexibility allows data models to change, templates to be updated. And the rest of the code keep working.

10.7.7 1.7.7. Changing in code

We could also do this directly in our output method by looking for the items and setting their `Visible` property.

```
var doc = Document.ParseDocument("MyFile.html");

var service = new OrderMockService2();
var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName = "Smith" }
↪;
```

(continues on next page)

(continued from previous page)

```
// A cast is needed to know the terms
var order = service.GetOrder(1) as OrderWithTerms;

doc.Params["model"] = new {
            user =  user,
            order = order
};

doc.Params["theme"] = new
{
    color = "#FF0000",
    space = "10pt",
    align = "center"
};

//Update the visibility of lookup items - dependent on them being there.
doc.FindAComponentById("paidAlready").Visible = (order.PaymentTerms < 0);
doc.FindAComponentById("payNow").Visible = (order.PaymentTerms == 0);
doc.FindAComponentById("payLater").Visible = (order.PaymentTerms > 0);

doc.SaveAsPDF("OutputPath.pdf");
```

This does, however, start to create a dependacy on the layout and the code along with potential errors this may cause later on plus dependencies on types and casting.

10.7.8 1.7.8. Expressions and calculations

We have already seen some binding syntax in scryber templates with functions and calculations between the handlebars.

```
{{.ItemPrice * .Quantity}}
{{index() + 1}}
{{concat('Hello ', model.user.FirstName)}}
```

There are many other functions for mathematical, comparison, aggregation and string operation. A complete list with examples of each are defined in the *Available expression functions - TD* section.

We can use standard calculations and aggregation operations on out model directly in the template.

```
<!DOCTYPE HTML>
<html lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>{{concat('Hello ', model.user.FirstName)}}</title>
</head>
<body>
    <!-- count of items and join of user name -->
    <div style='color: calc(theme.color); padding: calc(theme.space); text-align:␣
↪calc(theme.align)'>
        {{count(model.order.Items)}} items for {{join(' ',model.user.Salutation,␣
↪model.user.FirstName, model.user.LastName)}}.
    </div>
    <div style='padding: 10pt; font-size: 12pt'>
        <table style='width:100%'>
            <thead>
```

(continues on next page)

(continued from previous page)

```
                <tr>
                    <td>#</td>
                    <td>Item</td>
                    <td>Description</td>
                    <td>Unit Price</td>
                    <td>Qty.</td>
                    <td>Total</td>
                </tr>
            </thead>
            <tbody>
                <template data-bind='{{model.order.Items}}'>
                    <tr>
                        <td>{{index() + 1}}</td>
                        <td>{{.ItemNo}}</td>
                        <td>{{.ItemName}}</td>
                        <td>
                            <num value='{{.ItemPrice}}' data-format='{{model.order.
↪CurrencyFormat}}' />
                        </td>
                        <td>{{.Quantity}}</td>
                        <td>
                            <num value='{{.ItemPrice * .Quantity}}' data-format='{
↪{model.order.CurrencyFormat}}' />
                        </td>
                    </tr>
                </template>
            </tbody>
            <tfoot>
                <tr>
                    <td colspan="4"></td>
                    <td>Total (ex. Tax)</td>
                    <td><num value='{{model.order.Total}}' data-format='{{model.order.
↪CurrencyFormat}}' /></td>
                </tr>
                <tr>
                    <td colspan="4"></td>
                    <td>Tax</td>
                    <!-- Caclulate the tax -->
                    <td><num value='{{model.order.Total * model.order.TaxRate}}' data-
↪format='{{model.order.CurrencyFormat}}' /></td>
                </tr>
                <tr>
                    <td colspan="4"></td>
                    <td>Grand Total</td>
                    <!-- Calculate the grand total with tax -->
                    <td><num value='{{model.order.Total * (1 + model.order.TaxRate)}}
↪' data-format='{{model.order.CurrencyFormat}}' /></td>
                </tr>
            </tfoot>
        </table>
        <div id='terms'>
            <div id='paidAlready' hidden='{{if(model.order.PaymentTerms &lt; 0, "",
↪"hidden")}}'>
                <p>Thank you for pre-paying for these items. They will be shipped␣
↪immediately</p>
            </div>
            <div id='payNow' hidden='{{if(model.order.PaymentTerms == 0, "", "hidden
↪")}}'>
```

(continues on next page)

(continued from previous page)

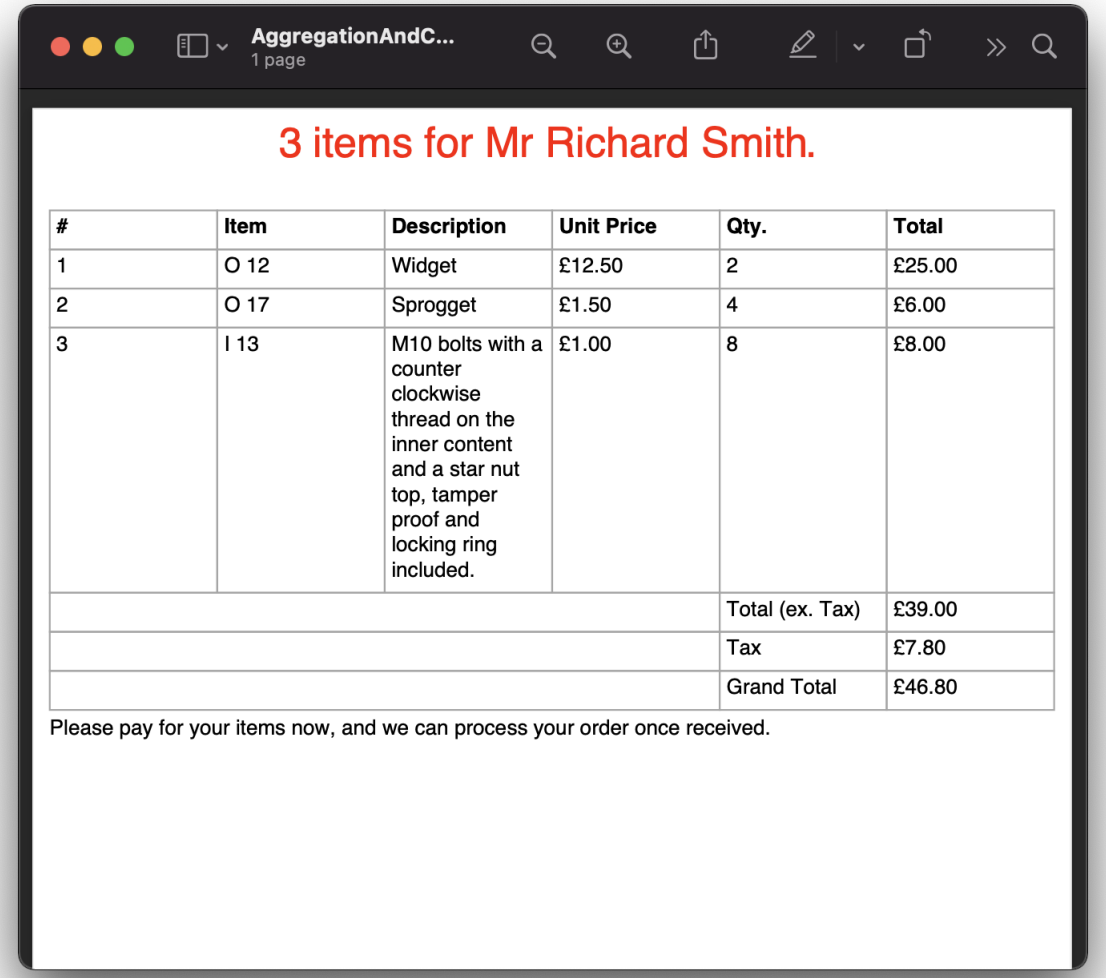
```
                <p>Please pay for your items now, and  we can process your order once␣
↪received.</p>
            </div>
            <div id='payLater' hidden='{{if(model.order.PaymentTerms &gt; 0, "",
↪"hidden")}}'>
                <p>Your items will be shipped immediately, please ensure you pay our␣
↪invoice within <b> {{model.order.PaymentTerms}} days</b></p>
            </div>
        </div>
    </div>
</body>
</html>
```

```
public void AggregationAndCalcBinding()
{
    var path = GetTemplatePath("Overview", "AggregationAndCalcBinding.html");

    using (var doc = Document.ParseDocument(path))
    {
        //Use mock service 2
        var service = new OrderMockService2();

        var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName =
↪"Smith" };
        var order = service.GetOrder(1);

        doc.Params["model"] = new
        {
            user = user,
            order = order
        };

        doc.Params["theme"] = new
        {
            color = "#FF0000",
            space = "10pt",
            align = "center"
        };

        using (var stream = GetOutputStream("Overview", "AggregationAndCalcBinding.pdf
↪"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

Full size version

10.7.9 1.7.9. Further Reading

- Next we can add some style to the template with *1.8. Styles, Classes and selectors*.
- See *Dynamic content in your template - PD* for more on the databinding capabilities and available functions.
- See *Controllers for your templates - PD* for a deep dive into interacting with your templates in code.



3 items for Mr Richard Smith.

#	Item	Description	Unit Price	Qty.	Total
1	O 12	Widget	£12.50	2	£25.00
2	O 17	Sprogget	£1.50	4	£6.00
3	I 13	M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included.	£1.00	8	£8.00
				Total (ex. Tax)	£39.00
				Tax	£7.80
				Grand Total	£46.80

Please pay for your items now, and we can process your order once received.

- See *Extending Scryber - TD* for more about the options for binding and configuration.

10.8 1.8. Styles, Classes and selectors

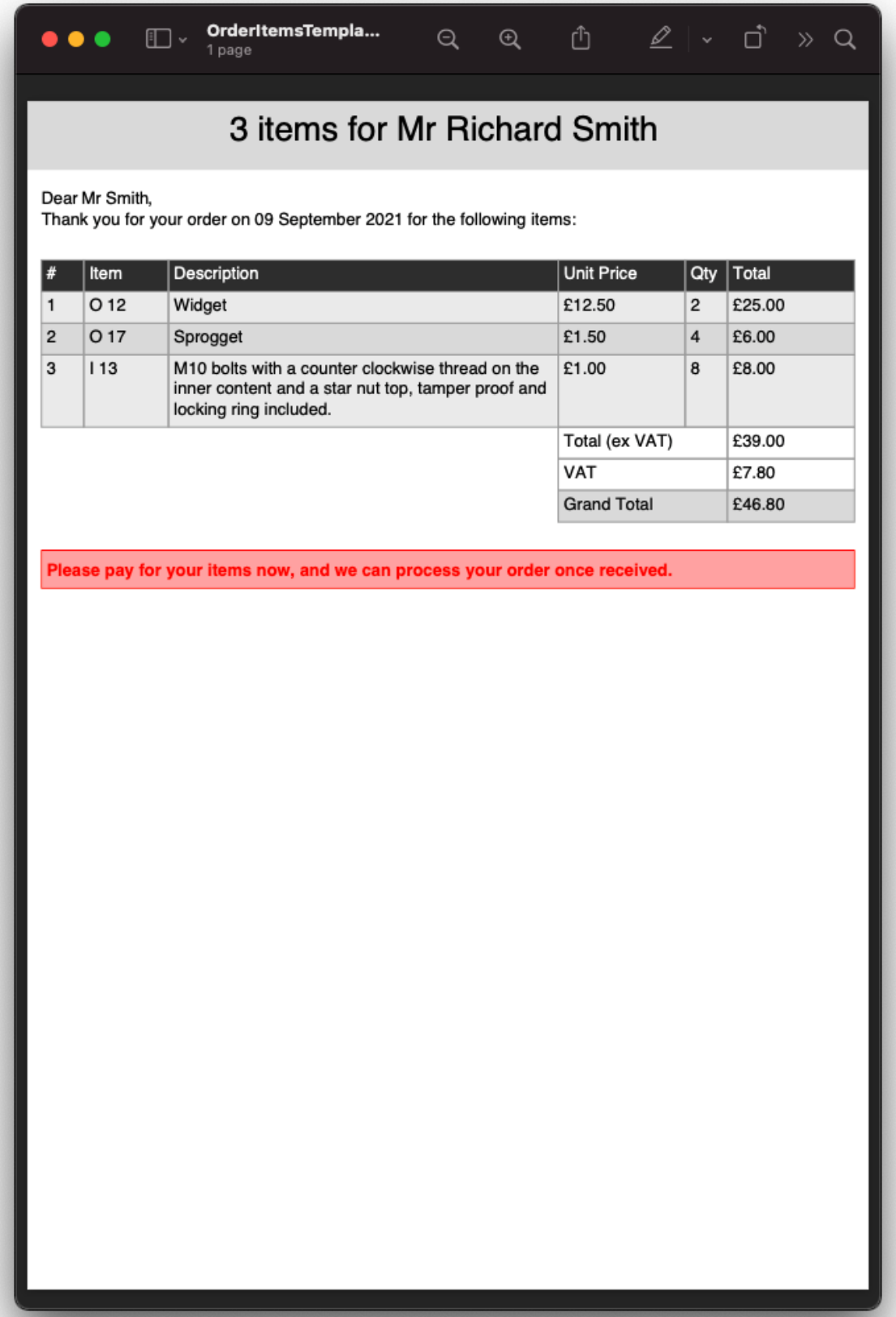
Scryber supports full cascading styles on all visual components. It also supports declaration of classes and styles within either the html head or as referenced stylesheets. And the use of `var()` and `calc()` for expressions.

10.8.1 1.8.1 Orders with style

From our previous example (*1.7. Document parameters and binding*) we can declare some styles inside the document head.

```
<!DOCTYPE HTML>
<html lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>{{concat('Hello ', model.user.FirstName)}}</title>
    <style type="text/css">
        :root {
            --theme_color: #000000;
            --theme_bg: #AAA;
            --theme_logo: url(../../Images/ScryberLogo.png);
            --theme_space: 10pt;
            --theme_align: center;
        }

        table.orderlist {
            width: 100%;
            font-size: 12pt;
        }

        table.orderlist thead {
            background-color: #333;
            color: white;
        }

        #terms {
            margin-top: 20pt;
            font-size: 12pt;
        }

        #payNow {
            border: 1px solid red;
            padding: 5px;
            background-color: #FFAAAA;
            color: #FF0000;
            font-weight: bold;
        }

        .heading{
            color: var(--theme_color, #000);
            background-color: var(--theme_bg, #FFF);
            background-image: var(--theme_logo);
            background-position: 5pt 5pt;
            background-repeat: no-repeat;
```

(continues on next page)



3 items for Mr Richard Smith

Dear Mr Smith,
Thank you for your order on 09 September 2021 for the following items:

#	Item	Description	Unit Price	Qty	Total
1	O 12	Widget	£12.50	2	£25.00
2	O 17	Sprogget	£1.50	4	£6.00
3	I 13	M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included.	£1.00	8	£8.00
			Total (ex VAT)		£39.00
			VAT		£7.80
			Grand Total		£46.80

Please pay for your items now, and we can process your order once received.

(continued from previous page)

```
            background-size: 35pt;
            padding: var(--theme_space, 10pt);
            text-align: var(--theme_align, left);
            border-bottom: solid 2px var(--theme_color);
        }
    </style>
</head>
<body>
    <!-- our heading has explicit styles -->
    <div class="heading">
        {{count(model.order.Items)}} items for {{join(' ',model.user.Salutation,␣
↪model.user.FirstName, model.user.LastName)}}.
    </div>
    <div style='padding: 10pt; font-size: 12pt'>
        <table style='width:100%'>
            <thead>
                <tr>
                    <td>#</td>
                    <td>Item</td>
                    <td>Description</td>
                    <td>Unit Price</td>
                    <td>Qty.</td>
                    <td>Total</td>
                </tr>
            </thead>
            <tbody>
                <template data-bind='{{model.order.Items}}'>
                    <tr>
                        <td>{{index() + 1}}</td>
                        <td>{{.ItemNo}}</td>
                        <td>{{.ItemName}}</td>
                        <td>
                            <num value='{{.ItemPrice}}' data-format='{{model.order.
↪CurrencyFormat}}' />
                        </td>
                        <td>{{.Quantity}}</td>
                        <td>
                            <num value='{{.ItemPrice * .Quantity}}' data-format='{
↪{model.order.CurrencyFormat}}' />
                        </td>
                    </tr>
                </template>
            </tbody>
            <tfoot>
                <tr>
                    <td colspan="4"></td>
                    <td>Total (ex. Tax)</td>
                    <td><num value='{{model.order.Total}}' data-format='{{model.order.
↪CurrencyFormat}}' /></td>
                </tr>
                <tr>
                    <td colspan="4"></td>
                    <td>Tax</td>
                    <!-- Caclulate the tax -->
                    <td><num value='{{model.order.Total * model.order.TaxRate}}' data-
↪format='{{model.order.CurrencyFormat}}' /></td>
                </tr>
```

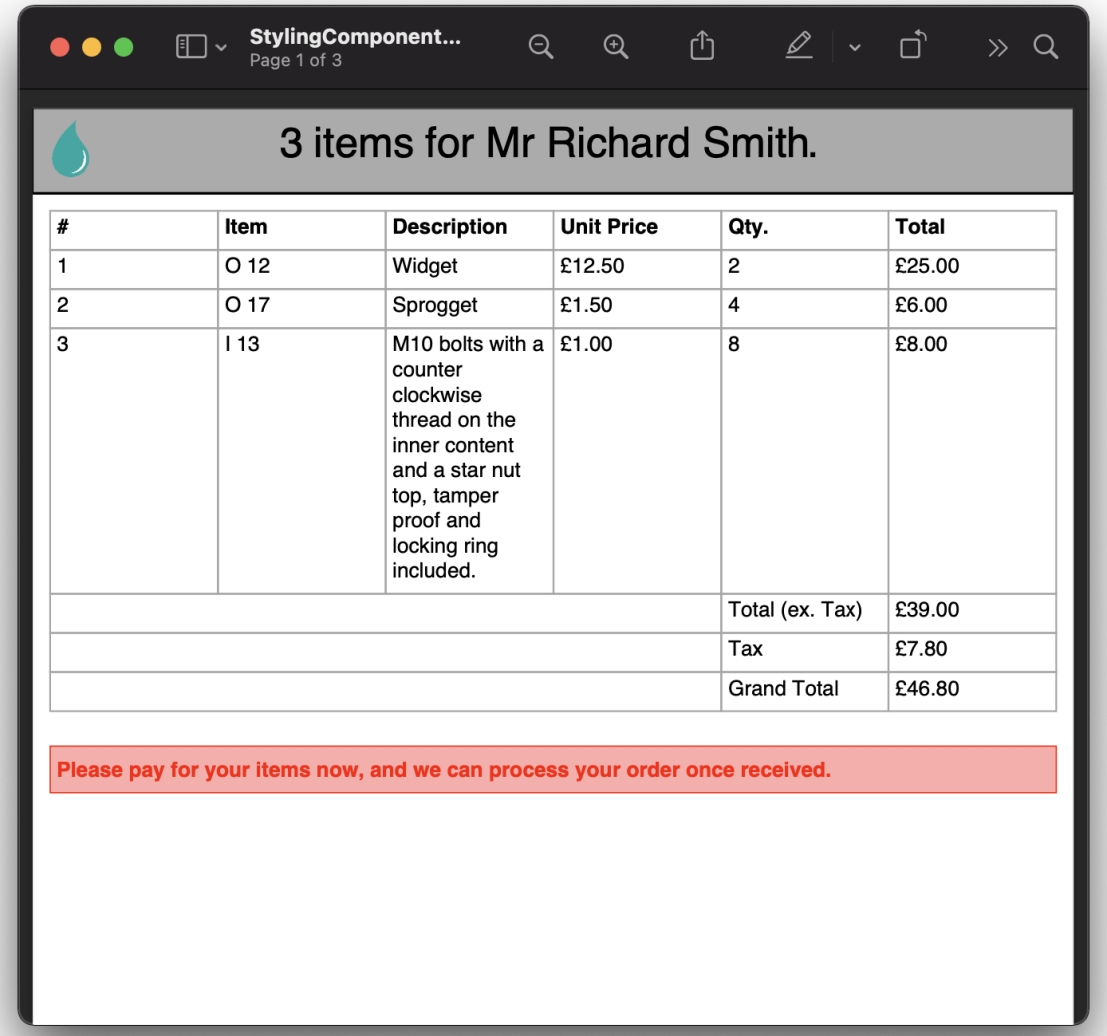
(continues on next page)

(continued from previous page)

```
                <tr>
                    <td colspan="4"></td>
                    <td>Grand Total</td>
                    <!-- Calculate the grand total with tax -->
                    <td><num value='{{model.order.Total * (1 + model.order.TaxRate)}}
↪' data-format='{{model.order.CurrencyFormat}}' /></td>
                </tr>
            </tfoot>
        </table>
        <div id='terms'>
            <div id='paidAlready' hidden='{{if(model.order.PaymentTerms &lt; 0, "",
↪"hidden")}}'>
                <p>Thank you for pre-paying for these items. They will be shipped␣
↪immediately</p>
            </div>
            <div id='payNow' hidden='{{if(model.order.PaymentTerms == 0, "", "hidden
↪")}}'>
                <p>Please pay for your items now, and  we can process your order once␣
↪received.</p>
            </div>
            <div id='payLater' hidden='{{if(model.order.PaymentTerms &gt; 0, "",
↪"hidden")}}'>
                <p>Your items will be shipped immediately, please ensure you pay our␣
↪invoice within <b> {{model.order.PaymentTerms}} days</b></p>
            </div>
        </div>
    </div>
</body>
</html>
```

```
public void ComponentStyles()
{
    var path = GetTemplatePath("Overview", "StylingComponents.html");

    using (var doc = Document.ParseDocument(path))
    {
        //Use mock service 2
        var service = new OrderMockService2();

        var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName =
↪"Smith" };
        var order = service.GetOrder(1);


        doc.Params["model"] = new
        {
            user = user,
            order = order
        };

        using (var stream = GetOutputStream("Overview", "StylingComponents.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

Now we can set up our theme and apply styles to the .header, .orderlist table and #payNow box. We were also able to remove the dependency on the *theme* parameter.



#	Item	Description	Unit Price	Qty.	Total
1	O 12	Widget	£12.50	2	£25.00
2	O 17	Sprogget	£1.50	4	£6.00
3	I 13	M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included.	£1.00	8	£8.00
				Total (ex. Tax)	£39.00
				Tax	£7.80
				Grand Total	£46.80

Full size version

In the same way as css, the font size of the #terms div is cascaded to the #payNow div, and the #payNow styles are applied over the top, overriding where appropriate.

The .header takes a number of css variables declared at the *:root* and applies them to the top div. These can be changed within selectors, can fallback, and as we will see later can be changed in code.

```
color: var(--theme_color, #000);
```

10.8.2 1.8.2 Allowed style selectors.

Scyber does **not** support the full set of selectors or the *!important* modifier (at the moment). We only support the use of

- Chained selectors on tags, classes and id.
- The > direct descendant selector.
- The :root selector
- The @font-face, @media and @page rules.

Other unsupported selectors and rules will be ignored.

```
/* these are supported */

.classname { }
#id { }
tag { }

tag.classname { }

tag.classname .inner { }
tag.classname tag.inner {  }

tag.classname > .direct.descendant { }

@media print {

    tag.mediaoverrides {

    }
}

/* these and other pseudo classes will not be supported

td:first {}
td::last {}

*/

/* Or these other rules

@import {}

@supports () {}

*/
```

10.8.3 1.8.3. Supported css properties

For a complete list of all the supported style properties see *Styles in your template - PD*, but as an overview scyber currently supports.

- Fills - Colors, images, positions, repeats and gradients.
- Strokes - Widths, dashes, colors and joins.
- Backgrounds - Colors, images, positions, repeats and gradients.

- Borders - Width, dashes, colors and individual sides.
- Text - Fonts, alignment, spacing, wrapping
- Size - Explicit width, height, minimum and maximum widths and heights.
- Positions - Block, Inline, Relative to parents, Absolute to the current page, 100% width.
- Spacing - Margins, padding including individual sides
- Lists - style, groups, number formats and labels.
- Page - sizes, orientations, numbers and formats.
- Columns - count, widths, gutter/alleys.

Note: All dimensions in scryber are based on actual sizes, rather than relative sizes. We are hoping to implement relative sizes, but for the moment units should be in Points (pt), Millimeters (mm) and Inches (in).

10.8.4 1.8.4. Caclulation in styles

As already seen using the `cacl()` function on css styles and classes is fully supported. This will respont to model content and also css values.

We can add a linked stylesheet with some table layout options, some calculated from a standard variable.

```
/* /Templates/Overview/Fragments/orderStyles.css */

body {
    --std-width: 30pt;
}

td.w1 {
    width: var(--std-width);
}

td.w2 {
    width: calc(var(--std-width) * 2.0);
}

td.w3 {
    width: calc(var(--std-width) * 3.0);
}

td.empty {
    border: none;
}

td.num {
    text-align: center;
}

td.curr {
    text-align:right;
}
```

We can reference this stylesheet in our html and apply the styles to the content including support for multiple classes.

```
<!--  /Templates/Overview/StylingWithCSSLink -->

<!DOCTYPE HTML>
<html lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>{{concat('Hello ', model.user.FirstName)}}</title>
    <!-- reference the stylesheet -->
    <link rel="stylesheet" href="./Fragments/orderStyles.css" />
    <style type="text/css">
        :root {
            --theme_color: #000000;
            --theme_bg: #AAA;
            --theme_logo: url(../../Images/ScryberLogo.png);
            --theme_space: 10pt;
            --theme_align: center;
        }

        table.orderlist {
            width: 100%;
            font-size: 12pt;
        }

        table.orderlist thead {
            background-color: #333;
            color: white;
        }

        #terms {
            margin-top: 20pt;
            font-size: 12pt;
        }

        #payNow {
            border: 1px solid red;
            padding: 5px;
            background-color: #FFAAAA;
            color: #FF0000;
            font-weight: bold;
        }

        .heading{
            color: var(--theme_color, #000);
            background-color: var(--theme_bg, #FFF);
            background-image: var(--theme_logo);
            background-position: 5pt 5pt;
            background-repeat: no-repeat;
            background-size: 35pt;
            padding: var(--theme_space, 10pt);
            text-align: var(--theme_align, left);
            border-bottom: solid 2px var(--theme_color);
        }
    </style>
</head>
<body>
    <div class="heading">
        {{count(model.order.Items)}} items for {{join(' ',model.user.Salutation,␣
↪model.user.FirstName, model.user.LastName)}}.
```

(continues on next page)

(continued from previous page)

```
    </div>
    <div style='padding: 10pt; font-size: 12pt'>
        <table style='width:100%'>
            <thead>
                <tr>
                    <td class="num">#</td>
                    <td>Item</td>
                    <td>Description</td>
                    <td class="curr">Unit Price</td>
                    <td class="num">Qty.</td>
                    <td class="curr">Total</td>
                </tr>
            </thead>
            <tbody>
                <!-- apply the widths and alignment styles to the rows -->
                <template data-bind='{{model.order.Items}}'>
                    <tr>
                        <td class="w1 num">{{index() + 1}}</td>
                        <td class="w2">{{.ItemNo}}</td>
                        <td>{{.ItemName}}</td>
                        <td class="w3 curr">
                            <num value='{{.ItemPrice}}' data-format='{{model.order.
↪CurrencyFormat}}' />
                        </td>
                        <td class="w1 num">{{.Quantity}}</td>
                        <td class="w3 curr">
                            <num value='{{.ItemPrice * .Quantity}}' data-format='{
↪{model.order.CurrencyFormat}}' />
                        </td>
                    </tr>
                </template>
            </tbody>
            <tfoot>
                <tr>
                    <td class="empty" colspan="3"></td>
                    <td colspan="2">Total (ex. Tax)</td>
                    <td><num value='{{model.order.Total}}' data-format='{{model.order.
↪CurrencyFormat}}' /></td>
                </tr>
                <tr>
                    <td class="empty" colspan="3"></td>
                    <td colspan="2">Tax</td>
                    <!-- Caclulate the tax -->
                    <td><num value='{{model.order.Total * model.order.TaxRate}}' data-
↪format='{{model.order.CurrencyFormat}}' /></td>
                </tr>
                <tr>
                    <td class="empty" colspan="3"></td>
                    <td colspan="2">Grand Total</td>
                    <!-- Calculate the grand total with tax -->
                    <td><num value='{{model.order.Total * (1 + model.order.TaxRate)}}
↪' data-format='{{model.order.CurrencyFormat}}' /></td>
                </tr>
            </tfoot>
        </table>
        <div id='terms'>
            <div id='paidAlready' hidden='{{if(model.order.PaymentTerms &lt; 0, "",
↪"hidden")}}'>
```

(continues on next page)

(continued from previous page)

```
                <p>Thank you for pre-paying for these items. They will be shipped␣
↪immediately</p>
            </div>
            <div id='payNow' hidden='{{if(model.order.PaymentTerms == 0, "", "hidden
↪")}}'>
                <p>Please pay for your items now, and  we can process your order once␣
↪received.</p>
            </div>
            <div id='payLater' hidden='{{if(model.order.PaymentTerms &gt; 0, "",
↪"hidden")}}'>
                <p>Your items will be shipped immediately, please ensure you pay our␣
↪invoice within <b> {{model.order.PaymentTerms}} days</b></p>
            </div>
        </div>
    </div>
</body>
</html>
```

And using the same method generate our document.

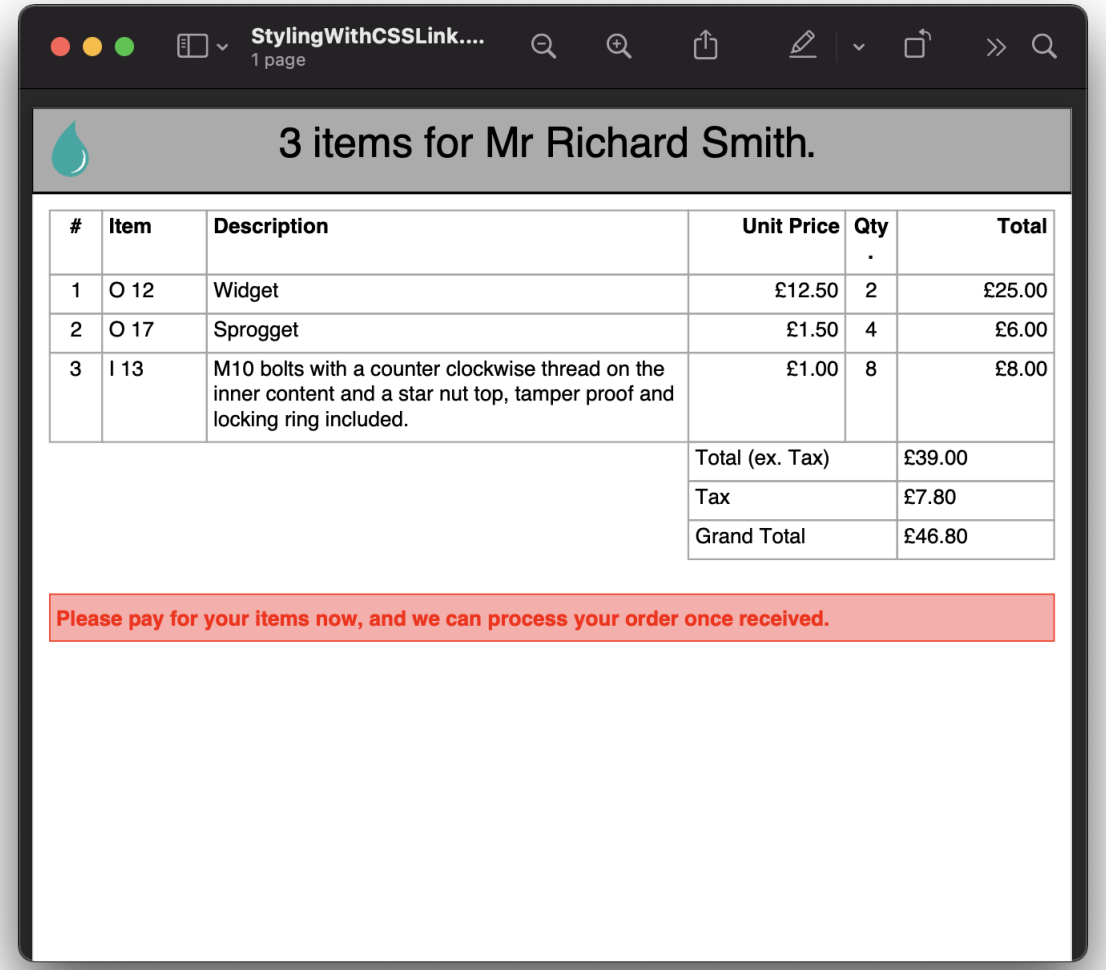
```
// Scryber.UnitSamples/OverviewSamples.cs

public void StylesWithCSSLink()
{
    var path = GetTemplatePath("Overview", "StylingWithCSSLink.html");

    using (var doc = Document.ParseDocument(path))
    {
        //Use mock service 2
        var service = new OrderMockService2();

        var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName =
↪"Smith" };
        var order = service.GetOrder(1);


        doc.Params["model"] = new
        {
            user = user,
            order = order
        };

        using (var stream = GetOutputStream("Overview", "StylingWithCSSLink.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

Full size version

10.8.5 1.8.5. Style values in code

Remember that all content parsed is converted to an object graph? This applies to styles as well.

All visual components (generally anything on a page) has a range of properties for setting styles As well as a `Style`


StylingWithCSSLink....
1 page
3 items for Mr Richard Smith.
Item Description Unit Price Qty. Total
1 O 12 Widget £12.50 2 £25.00
2 O 17 Sprogget £1.50 4 £6.00
3 I 13 M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included. £1.00 8 £8.00
Total (ex. Tax) £39.00
Tax £7.80
Grand Total £46.80
Please pay for your items now, and we can process your order once received.

property for seting the values. And they can even be set directly using the defined `StyleKeys`.

```
// These are all equivalent

//using Scryber.Drawing;
//using Scryber.Styles;

div.BackgroundColor = new PDFColor(255, 0, 0);
div.Style.Background.Color = new PDFColor(255, 0, 0);
div.Style.SetValue(StyleKeys.BgColorKey, new PDFColor(255, 0, 0));
```

We can also override css variables in our document, as values in the params dictionary

```
doc.Params["--bg-color"] = new PDFColor(255, 0, 0); // could also be a string value.
```

We can even define our own styles in the document to override

```
//using Scryber.Drawing;
//using Scryber.Styles;

//A style definition is a style with a selector.

var defn = new StyleDefn("div#payNow");
defn.Background.Color = new PDFColor(255, 0, 0);
doc.Styles.Add(defn);
```

```
//using Scryber.Components
//using Scryber.Drawing
//using Scryber.Styles

var doc = Document.ParseDocument("MyFile.html");

var service = new OrderMockService();
var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName = "Smith" }
↪;
var order = service.GetOrder(1);
order.PaymentTerms = 30;

doc.Params["model"] = new {
            user =  user,
            order = order
};

var grid = doc.FindAComponentById("orders") as TableGrid;
var pay = doc.FindAComponentById("payNow") as Div;


//Properties directly on the visual component.
grid.BackgroundColor = "#EEE";

//Using the style property
grid.Style.Margins.Right = 20;
grid.Style.Margins.Left = 20;

//Using style keys
pay.Style.SetValue(StyleKeys.BorderStyleKey, LineType.Dash);
pay.Style.SetValue(StyleKeys.BorderDashKey, PDFDashes.LongDash);
```

(continues on next page)

(continued from previous page)

```
//A new style to the document
StyleDefn style = new StyleDefn("#terms div#payNow");
style.Border.Width = 2;
doc.Styles.Add(style);

doc.SaveAsPDF("OutputPath.pdf");
```

Full size version

Note: We had to set the Border Style to dash, as well as providing a dash value, as our css styles had defined the border as solid.

All the style properties are strongly typed, even the `Style.SetValue` as the style keys are strongly typed. However most of the values used have an explicit or implicit conversion from numbers or strings, or a simple constructor. The main classes (and structs) used in styles are

- PDFUnit - a basic dimension with units. Implicit conversion from a number, along with parsing and constructors. See drawing_units
- PDFColor - a standard color in either RGB, CMYK or Gray scale. Implicit conversion from a string, along with parsing and constructors. See drawing_colors
- PDFThickness - 4 PDFUnits in a top, right, bottom and left order. Parsing and constructors. See drawing_units
- PDFFontSelector - A chained list of names of fonts, e.g "Arial" sans-serif. Explicit conversion along with parsing and constructor. See drawing_fonts
- Various Enumerations - Used for setting style types such as line caps, background styles, etc.

10.8.6 Base components styles

Each component has a standard base style applied. For example the Div has a position mode of block. The paragraph also has a position mode of block, but includes a top margin of 4 points. The table cell has a standard gray 1 point border. By defining these there is a consistant appearance, but these can be easily overriden using css styles in your document or referenced css stylesheet.

```
td { border: none; }
```

10.8.7 Using calc() and binding dynamic values.

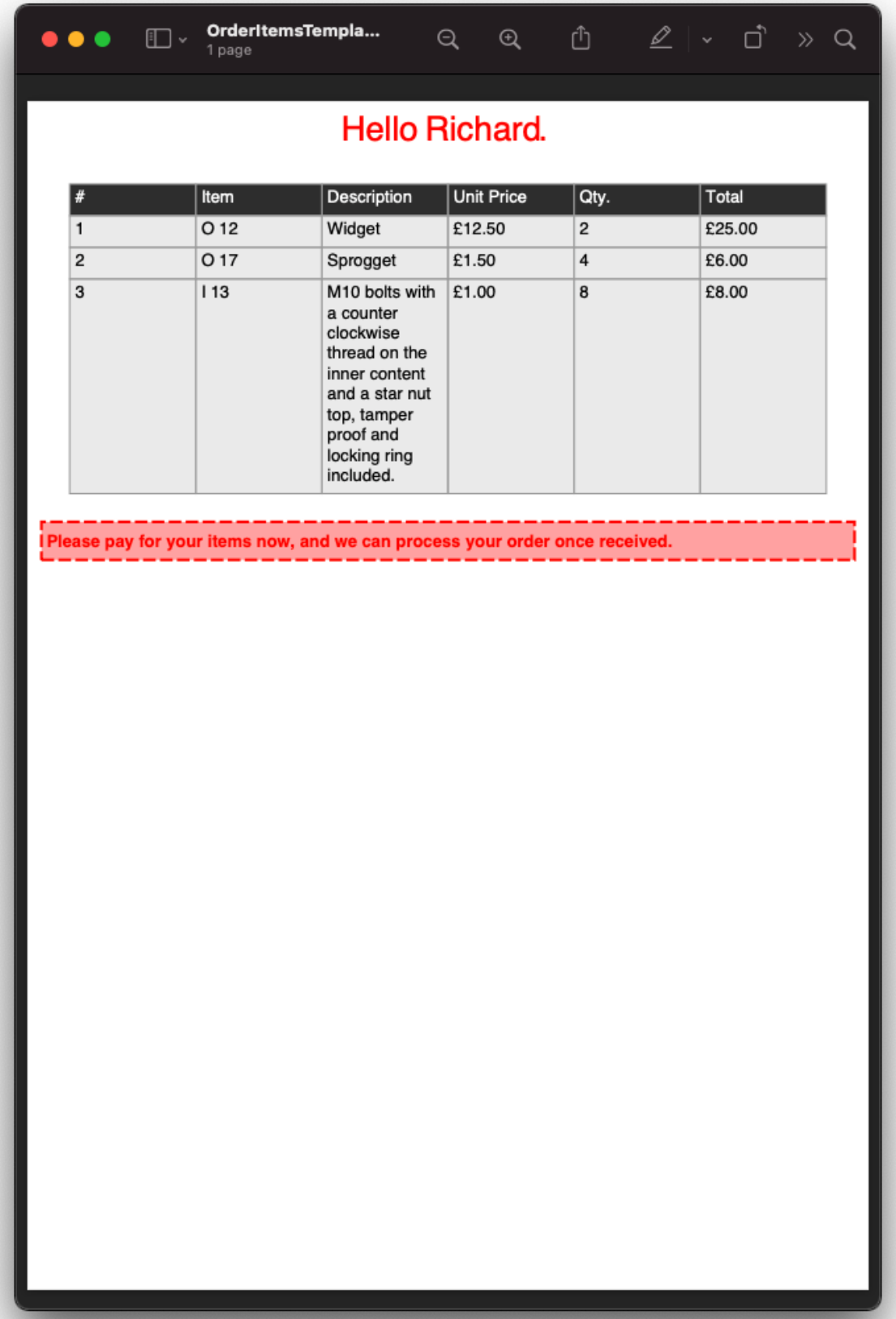
Along with support for `var()` for looking up css variables, scryber supports `calc()`. This enables styles to be completely dynamic as well as the data.

The functions can either be on the css classes or wthin the style attribute itself.

In our linked orderStyles.css file we can set up some standard widths.

```
:root {
    --std-width: 30pt;
}

.td_w1 {
    width: var(--std-width);
}
```

(continues on next page)


OrderItemsTempla...
1 page
Hello Richard.
Item Description Unit Price Qty. Total
1 O 12 Widget £12.50 2 £25.00
2 O 17 Sprogget £1.50 4 £6.00
3 I 13 M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included. £1.00 8 £8.00
Please pay for your items now, and we can process your order once received.

(continued from previous page)

```
.td_w2 {
    width: calc(var(--std-width) * 2.0);
}

.td_w3 {
    width: calc(var(--std-width) * 3.0);
}
```

And in our code we can create a style parameter.

```
//using Scryber.Components
//using Scryber.Drawing

var doc = Document.ParseDocument("MyFile.html");

var service = new OrderMockService2();
var user = new User() { Salutation = "Mr", FirstName = "Richard", LastName = "Smith" }
↪;
var order = service.GetOrder(1);
order.PaymentTerms = 30;

doc.Params["model"] = new {
            user =  user,
            order = order
};

//new style document parameter
doc.Params["style"] = new
{
    rowColor = (PDFColor)"#EEE",
    altColor = (PDFColor)"#DDD",
    dateFormat ="dd MMMM yyyy",
    currencyFormat = "£##0.00"
};

doc.SaveAsPDF("OutputPath.pdf");
```

And finally we can update our template to use the new styles and add a bit more juice to the template.

```
<!DOCTYPE HTML>
<html lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>{{concat('Orders for ', model.user.FirstName)}}</title>
    <link rel="stylesheet" href="./css/orderStyles.css" />
    <style type="text/css">
        :root {
            --theme_color: #FF0000;
            --theme_space: 10pt;
            --theme_align: center;
            --theme_fsize: 12pt;
        }

        table.orderlist {
            width: 100%;
            font-size: var(--theme_fsize);
        }
```

(continues on next page)

(continued from previous page)

```
        table.orderlist thead {
            background-color: #333;
            color: white;
        }


        #terms {
            margin-top: 20pt;
            font-size: var(--theme_fsize);
        }

        #payNow {
            border: 1px solid red;
            padding: 5px;
            background-color: #FFAAAA;
            color: #FF0000;
            font-weight: bold;
        }

    </style>
</head>
<body>
    <!-- setting the background color to the style -->
    <div style='background-color: calc(style.altColor); padding: var(--theme_space);␣
↪text-align: var(--theme_align)'>
        {{count(model.order.Items)}} items for {{join(' ', model.user.Salutation,␣
↪model.user.FirstName, model.user.LastName)}}
    </div>
    <!-- a preamble paragraph with concatenated values and a date -->
    <div style="padding: var(--theme_space);">
        <p style="font-size: var(--theme_fsize);" >
            Dear {{concat(model.user.Salutation, ' ', model.user.LastName)}},<br/>
            Thank you for your order on <time data-format="{{var(style.dateFormat,
↪'dd MMM yyyy')}}" /> for the following items:
        </p>
    </div>
    <div style='padding: var(--theme_space);'>
        <!-- classes on the header cells define the width of the table cells
            relative to the variable in the css stylesheet. -->
        <table id="orders" class="orderlist">
            <thead>
                <tr>
                    <td class="td_w1">#</td>
                    <td class="td_w2">Item</td>
                    <td >Description</td>
                    <td class="td_w3">Unit Price</td>
                    <td class="td_w1">Qty</td>
                    <td class="td_w3">Total</td>
                </tr>
            </thead>
            <tbody>
                <!-- Changing the row color for alternates -->
                <template data-bind='{{model.order.Items}}'>
                    <tr style="background-color: calc(if(index() % 2 == 1, style.
↪altColor, style.rowColor));">
                        <!-- The indexing of the loop + 1 -->
```

(continues on next page)

(continued from previous page)

```
                        <td>{{index() + 1}}</td>
                        <td>{{.ItemNo}}</td>
                        <td>{{.ItemName}}</td>
                        <td>
                            <!-- Data format is now coming from the style parameter --
↪>
                            <num value='{{.ItemPrice}}' data-format='{{style.
↪currencyFormat}}' />
                        </td>
                        <td>{{.Quantity}}</td>
                        <td>
                            <num value='{{.ItemPrice * .Quantity}}' data-format='{
↪{style.currencyFormat}}' />
                        </td>
                    </tr>
                </template>
            </tbody>
            <!-- Added some footer rows for calculations with fallback values -->
            <tfoot>
                <tr>
                    <td class="noborder" colspan="3"></td>
                    <td colspan="2">Total (ex VAT)</td>
                    <td colspan="1">
                        <num value='{{model.order.Total}}' data-format='{{style.
↪currencyFormat}}' />
                    </td>
                </tr>
                <tr>
                    <td class="noborder" colspan="3"></td>
                    <td colspan="2">VAT</td>
                    <td colspan="1">
                        <num value='{{model.order.Total * var(model.order.TaxRate,0.
↪2)}}' data-format='{{style.currencyFormat}}' />
                    </td>
                </tr>
                <tr>
                    <td class="noborder" colspan="3"></td>
                    <td colspan="2" style="background-color: calc(style.altColor);">
↪Grand Total </td>
                    <td colspan="1" style="background-color: calc(style.altColor);">
                        <num value='{{model.order.Total + (model.order.Total *␣
↪var(model.order.TaxRate, 0.2))}}' data-format='{{style.currencyFormat}}' />
                    </td>
                </tr>
            </tfoot>
        </table>
        <div id='terms'>
            <div id='paidAlready' hidden='{{if(model.order.PaymentTerms &lt; 0, "",
↪"hidden")}}'>
                <p>Thank you for pre-paying for these items. They will be shipped␣
↪immediately</p>
            </div>
            <div id='payNow' hidden='{{if(model.order.PaymentTerms == 0, "", "hidden
↪")}}'>
                <p>Please pay for your items now, and we can process your order once␣
↪received.</p>
            </div>
```

(continues on next page)

(continued from previous page)

```
            <div id='paySoon' hidden='{{if(model.order.PaymentTerms &gt; 0, "",
↪"hidden")}}'>
                <p>Your items will be shipped immediately, please ensure you pay our␣
↪invoice within <b>{{model.order.PaymentTerms}} days</b></p>
            </div>
        </div>
    </div>
</body>
</html>
```

And our output should now look something similar to this.

Full size version

There is a lot going on here, but...

- The heading is counting the number of order items and joining some strings together with the alt style
- The table head is setting the widths of the columns that the content flows into, leaving description to fill the rest of the space.
- The table body has a `template` and is looping over the `model.order.items` collection, and creating a row for each of the items.
- The `index()` function is returning the *zero-based* index in the collection.
- The `if(calc, true, false)` function is setting the style for alternate rows.
- Inside the template row we are referring to the current item with the dot prefix.
- Inside the template row we can still reference the global document parameters without the dot prefix.
- The I 13 item has a long desciption that is flowing across multiple line in the cell.
- The `footer` rows are performing some calculations based on the summary information, and outputting the total values.
- The `model.order.TaxRate` is being looked for, but is not available, so the fallback var() value is being used.
- The `num @data-format` and `time @data-format` are changing the output text to a formatted value within the style

10.8.8 Next Steps

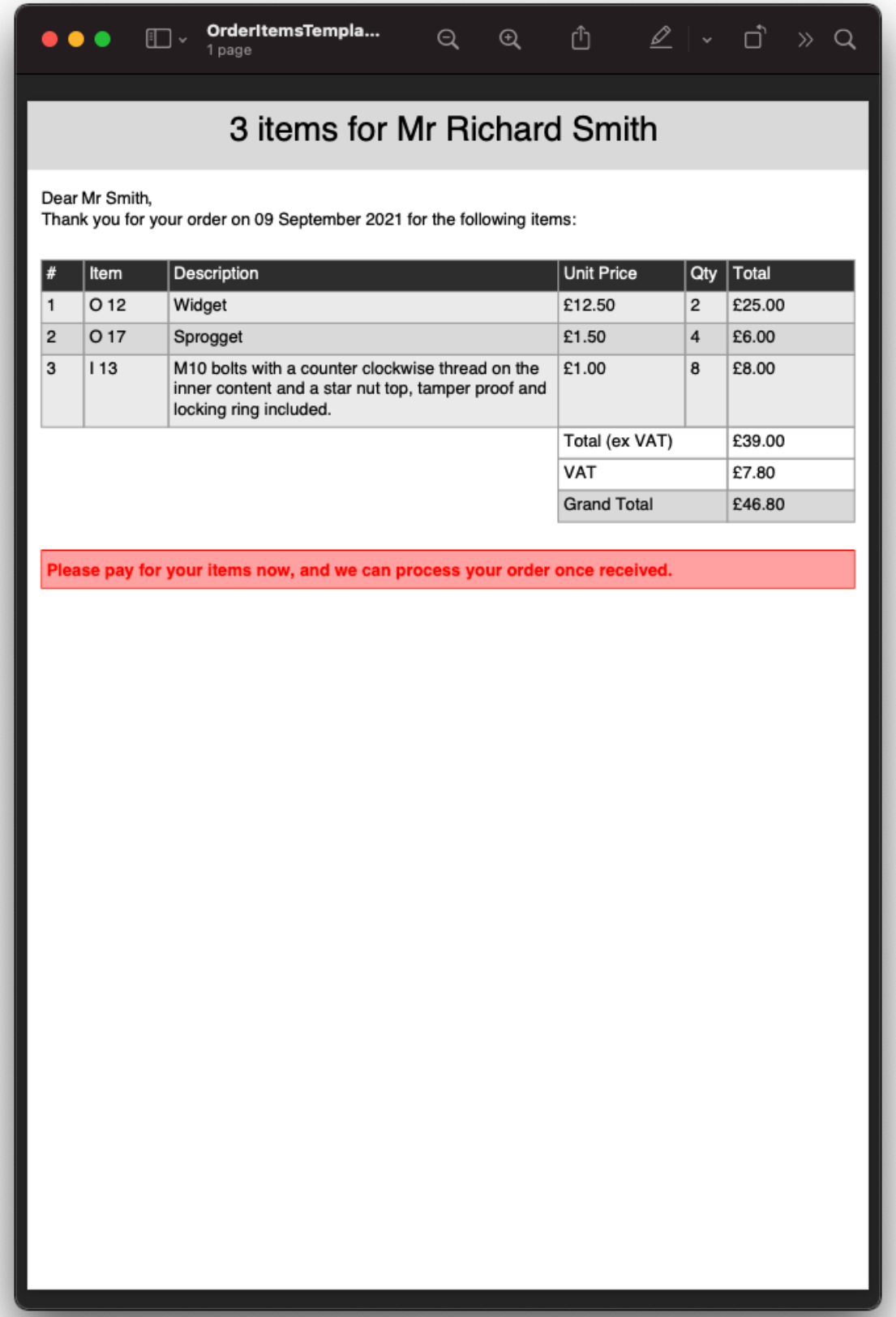
See 1_overview/9_document_output to understand more on the options for outputting a document to a stream or a file

See 4_styles/1_document_styles for a more detailed explation of each of the styling options within scryber.

See 6_binding/4_css_calc_reference to get a deep dive into the calc() and var() support in scryber.

See 2_document/3_drawing_units, 2_document/4_drawing_colors and 2_document/5_drawing_fonts for more on the support for measurements, colors and fonts.

Note: Remember, all of this is part of an object instance. The options to build a document are completely dynamic.


OrderItemsTempla...
1 page
3 items for Mr Richard Smith
Dear Mr Smith,
Thank you for your order on 09 September 2021 for the following items:
Item Description Unit Price Qty Total
1 O 12 Widget £12.50 2 £25.00
2 O 17 Sprogget £1.50 4 £6.00
3 I 13 M10 bolts with a counter clockwise thread on the inner content and a star nut top, tamper proof and locking ring included. £1.00 8 £8.00
Total (ex VAT) £39.00
VAT £7.80
Grand Total £46.80
Please pay for your items now, and we can process your order once received.

10.9 Output to files, streams or results - TD

We have seen 2 ways to create the resultant document from a template.

For MVC projects the extension methods for the controller allow assignment to a stream as a `FileResult`, and for console or GUI applications as an actual file on a file system `SaveAsPDF`.

10.10 Version History

The following change log is for developers upgrading from previous versions, or looking for new features

10.10.1 Version 5.1.0

Released 31st August 2021

Major update to the binding framework

- Added support for handlebar expressions {{}}, as well as calc() and var() in css
- Extensible function library based on the Expressive open source library
- Updated the library to use the document component for loading remote files, fonts and images.
- Added async support for document Generation with remote requests for images, fonts, etc.

Other minor enhancements and fixes

- Added caching support to the document via the Scryber.ServiceProvider.
- Fixed a bug on the font factory for multi-threading.

10.10.2 Version 5.0.7

Released 30th May 2021

Cool new features added

- Added support for float left and right within a single block (e.g. p, div)
- Added support for linear and radial gradients within css.
- A couple of other minor bug fixes.

10.10.3 Version 5.0.6

Released 30th March 2021

A catch up and fix release for the library, while we are building the docker images and playground.

Minor enhancements and bug fixes

- Support for parsed JSON objects in binding - along with std types and dynamic objects. (See: binding_model)
- Css 'margin:value' is applied to all margins even if explicit left, right etc. has been previously applied. (See: document_styles)
- Conformance is now carried through to templates, so errors are not indavertantly raised inside the template. (See: extending_logging)

- Missing background images will not raise an error. (See: drawing_images)
- Support for data images (src='data:image/..') within content - thanks Dan Rusu!
- Images are not duplicated within the output for the same source.

10.10.4 Version 5.0.5

Released 28th February 2021

Big Hitters

- Embed and iFrame support. (See: document_references)
- Support for border-left, border-right, etc (See: drawing_colors)
- Support for encryption and restrictions. (See: document_security)
- Support for base href in template files. (See: document_structure)
- Added em, strong, strike, del, ins elements. (See: document_textlayout)

Minor enhancements and bug fixes

- Classes and styles on template tags are supported.
- Html column width and break inside
- CSS and HTML Logging
- Binding speed improvements for longer documents.
- Fixed application of multiple styles with the same word inside
- Allow missing images on the document is now supported.
- Contain fill style for background images.

10.10.5 Version 5.0.4

**Initial SVG Support (See: drawing_paths) **

Local font urls along with some bug fixes.

10.10.6 Version 5.0.3

- Added @font-face, absolute, relative and display css. (See: drawing_fonts)
- Support for @page css directives for the whole document and section page sizes. (See: drawing_fonts)
- Support for <page /> tags with property or for attributes. (See: drawing_fonts)
- Added support for HTML binding with the template tag and data-bind attribute (See: binding_model)
- Fix for anchor links with internal and external href.
- Fixes for single character css values and other minor updates.

5.0.1-alpha

** Upgrade to support dotnet 5 **

Plus increased support for the HTML parsing with entities and DTD

10.10.7 Version 1.1 Core Change log

This is a breaking change for existing implementations, but represents a significant step foreward.

- XML content should now use the doc: prefix for the components namepsace
- The Scryber.Components namespace classes no longer have the PDF prefix i.e. PDFDocument is now Document.
- The output of a pdf method has changed SaveAsPDF
- Updated the schemas to match the new document structure

Other changes include the use of the match='[css selector]' on styles with priorities based on depth, and the support for xhtml as a root element in a document parsing.

10.10.8 Version 1.0 Core Change log

The first release of the library for DotNet Core

It includes the switch to a Document/Data element Improved layout capabilities The support for TTC (true type collection fonts) Various other enhancements

10.11 Document Overview - TD

We have seen 2 ways to create the resultant document from a template.

For MVC projects the extension methods for the controller allow assignment to a stream as a `FileResult`, and for console or GUI applications as an actual file on a file system `SaveAsPDF`.

10.12 Using units and measures - PD

Within scryber all drawing and positioning is based from the top left of the page. Scryber allows the definition of a dimension based on a number of positioning and sizing structures. All based around the **Unit** of measure.

In all the examples so far we have used pt (points) as the unit of measure, but scryber also supports the use of millimeters (mm), inches (in) and also pixels (px) as postfix units. A pixel is translated to 1/96 th of an inch for printing.

Note: Scryber does not support the use of relative dimensions: em, rem, vh etc. The only exception is the use of 100% on widths.

- **PDFUnit**
  - This is the base single dimension value.
  - Its default scale is the prinding standard points unit (1/72nd of an inch).
  - Values can also be specified in millimeters (mm) and inches (in) as well as explicitly in points.
  - Units are used in many places in xml templates
  - e.g. 72, 72pt, 1in, 25.4mm would all represent a 1.0 inch dimension.
  - Units can directly be cast and converted from integer and double values, or constructed in code.

- **PDFSize**
  - This is a width and height dimension with 2 PDFUnits.
  - Units can be mixed and matched within a size, but are generally only used internally for calculation
  - e.g. *72pt 1in* is a 1.0 inch wide and high
- **PDFPoint**
  - This represents a location on a page or container with an x and y component.
  - Again units can be mixed and matched within a point, but are generally only used internally for calculation
  - e.g. *72pt 1in* is 1 inch in from the left of the container and 1 inch down.
- **PDFThickness**
  - A thickness represent 4 dimensions around a square.
  - It follows the same order as html starting at the top and moving in a clockwise direction to the right, bottom and left.
  - It can be defined with 1, 2 or 4 values as a string, where 1 dimension refers to all values, 2 is the vertical and then horizontal and all 4 are explicit.
  - Thicknesses are used by the margins, padding, clipping attributes on components.
  - e.g. *25.4mm*, *1in 72pt* or *72 72 72 72* are all equivalent to 1 inch thickness all around.
- **PDFRect**
  - A rectagle is represented by 4 dimensions forming the x, y, width and height of a rectangle.
  - They can be mixed and matched in units, but are generally only used internally for calculation.
  - This should not be confused with the PDFRectangle used in drawing (see drawing_paths)

10.12.1 Units use in templates

When using units in xml templates its easy just to provide the values. For example the following will add an absolutely positioned Div on a page with some thickness padding textual content

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">
        body{
            background-color: aqua;
        }

        .positioned {
            position: absolute;
            top: 30mm;
            left: 40mm;
            width: 100mm;
            padding:20pt 0.25in 4mm 20pt;
            background-color: #AAAAFF;
```

(continues on next page)

(continued from previous page)

```
        }

    </style>
</head>
<body>
    <div class="positioned">
        20pt padding all around at 10pt, 20pt with a width of 100mm.
    </div>
</body>
</html>
```

2_document/images/drawingunits1.png

10.12.2 Units in code

The same could have be achieved in code using the Unit and Thickness constructors.

All the dimensions have a range of constructors, casting and parsing options as needed.

```
//using Scryber.Drawing

PDFUnit unit1 = 20; //implicit cast to 20pts
var unit2 = (PDFUnit)72; //explicit cast to 72 points (1 inch)
var unit3 = new PDFUnit(1, PageUnits.Inches); //explicit unit scale

var pt1 = new PDFPoint(20,72); //defaults to points
var pt2 = new PDFPoint(unit1, unit2); //explicit unit dimensions

var thick1 = new PDFThickness(unit3); //Applies to all with a PDFUnit value
var thick2 = new PDFThickness(10,20,10,20); //Applies explicit values to each␣
↪dimensions

var rect = PDFRect.Empty; //Set to Zeroed values.
rect.Inflate(thick2); //Then inflate the rectangle by the thickness.

var rect2 = PDFRect.Parse("12pr 10pt 100pt 2in"); //And all support parsing too.
```

10.12.3 One Hundred Percent

The special value of 100% for width applies true to the underlying FullWidth style value.

By default div's have a FullWidth of true, so they will be 100% wide, but tables, lists etc do not. By specifying a width of 100% on these, they will use all the available space.

See component_positioning for more information.

10.12.4 Overiding relative units

Finally: If there is an existing template or file being used, then overriding any relative styles can be done using the @media print rule - so it will only be used by scryber (or when the document is printed anyway).

10.13 Using Colours, fills, borders and backgrounds - PD

Colours are a standard drawing structure. They can be defined in the template and code in multiple ways.

10.13.1 Named Colours

Within a template the 140 standard HTML colours can be used These can be defined on any color style or css class,

```
<div style='background-color:red'>Red Content</div>
```

In the core library 16 pre-defined colors have been defined, and are available in the Scryber.Drawing.PDFColors class.

```
var div = new Scryber.Components.Div(){ BackgroundColor = Scryber.Drawing.PDFColors.
↪Red };
```

10.13.2 Colour Explicit Values

The colour also supports definition with explicit Hex, RGB or Grayscale values.

- **Grayscale**
  - Can be applied to an attribute with the format *G(255)*.
  - Can be applied to an attribute with the 2 character hex format *#FF*.
  - Can be assigned in code via the constructor *new PDFColor(1)*.
- **RGB color**
  - Can be applied to an attribute with the format *RGB(255,0,0)*.
  - Can be applied to an attribute with the 3 character format *#F00*
  - Can be applied to an attribute with the 6 character format *#FF0000*
  - Can be assigned in code via the constructor *new PDFColor(1,0,0)*
- **RGBA color**
  - Can only be applied to the css colors in the standard rgba(255,0,0,0.5)

```
<div style='background-color:#FF0000'>Red Content</div>
```

```
var div = new Scryber.Components.Div(){ BackgroundColor = new Scryber.Drawing.
↪PDFColor(1,0,0) };
```

```
<div style='background-color:rgba(255,0,0,0.5)'> 50% transparent red Content</div>
```

The use of transparency (opacity) is not part of the colour structure, but most drawing operations also allow a separate opacity value to be set.

Note: The internal styles have a color property and an opacity property (Background.Color, Background.Opacity) to allow image backgrounds and fills to obey this opacity.

10.13.3 Using background colors

All pages and block components (see component_positioning) support background colors and opacity. This can either be set via classes or explicitly on the style.

Inline components do not (currently) support rendering of a background colour.

The background is simply a solid colour from the Grayscale or RGB ranges. The background can also be specified with a fraction value from 0.0 to 1.0.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">
        body{ padding-top: 40pt;
            font-size:20pt;
            background-color: aqua;
        }
        .floating {
            position: absolute;
            top: 80pt;
            left: 230pt;
            background-color: rgba(255, 0, 255, 0.5);
            padding:10pt;
        }
    </style>
</head>
<body>
    <div style="background-color:lightpink; padding:20pt">
        Light pink full opacity background.
    </div>
    <div class="floating">
        This is the content in a semi-opaque fuschia background ontop of the page.
    </div>
</body>
</html>
```

2_document/images/documentbgcolor.png

Note: Backgrounds also support the use of single or repeating images. See drawing_images for details on using

images backgrounds.

10.13.4 Using border colors

Borders apply around the edges of block components. They can be solid or dashed (see below), and have color, opaciy and width values.

Margins are outside of the border, and padding is inside. But borders do not affect either, by design.

Scryber (v5.0.5) also supports the use of border sides (border-left etc.) and corner radius.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        body{ padding-top: 40pt;
            font-size:20pt;
            background-color: aqua;
        }

        .floating {
            position: absolute;
            top: 80pt;
            left: 230pt;
            background-color: rgba(255, 0, 255, 0.5);
            padding:10pt;
            /* Adding the border */
            border: 6pt rgba(127,0, 0, 0.7);
            margin-right:10pt;
        }
    </style>
</head>
<body>
    <div style="background-color:lightpink; padding:20pt;
                border-top: solid 3pt #C77; border-bottom: solid #C77 3pt;">
        Light pink full opacity background.
    </div>
    <div class="floating">
        This is the content in a semi-opaque fuschia background ontop of the page.
    </div>
</body>
</html>
```

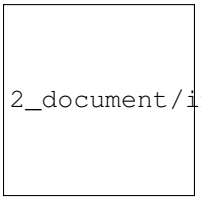
2_document/images/documentbordercolor.png

10.13.5 Using fill colors

The fill color applies to shapes and text. It is independent of background, however the same attributes apply to fills as to backgrounds.

See drawing_paths for more on using fills with shapes.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">
        body{ padding-top: 40pt;
            font-size:20pt;
            background-color: aqua;
        }
        .floating {
            position: absolute;
            top: 80pt;
            left: 230pt;
            background-color: rgba(255, 0, 255, 0.5);
            padding:10pt;
            /* Adding the border */
            border: 6pt rgba(127,0, 0, 0.7);
            margin-right:10pt;

            /* Adding a fill */
            color: #FFFFFF;
        }
    </style>
</head>
<body>
    <div style="background-color:lightpink; padding:20pt;
                border-top: solid 3pt #C77; border-bottom: solid #C77 3pt;
                color:aqua; fill-opacity: 0.7; font-weight:bold;">
        Light pink full opacity background.
    </div>
    <div class="floating">
        This is the content in a semi-opaque fuschia background ontop of the page.
    </div>
</body>
</html>
```

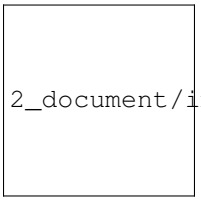


Note: Fills also support the use of single or repeating images. See drawing_images for details on using images for fills.

10.13.6 Using stroke colors

Finally stroke is around the shape or text. It supports the same properties as the border.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">
        body{ padding-top: 40pt;
            font-size:20pt;
            background-color: aqua;
        }
        .floating {
            position: absolute;
            top: 80pt;
            left: 230pt;
            background-color: rgba(255, 0, 255, 0.5);
            padding:10pt;
            /* Adding the border */
            border: 6pt rgba(127,0, 0, 0.7);
            margin-right:10pt;

            /* Adding a fill */
            color: #FFFFFF;

            /* Adding a stroke */
            stroke: #000;
            stroke-width: 1pt;
        }
    </style>
</head>
<body>
    <div style="background-color:lightpink; padding:20pt;
                border-top: solid 3pt #C77; border-bottom: solid #C77 3pt;
                color:aqua; fill-opacity: 0.7; font-weight:bold;">
        Light pink full opacity background.
    </div>
    <div class="floating">
        This is the content in a semi-opaque fuschia background ontop of the page.
    </div>
</body>
</html>
```



10.13.7 Border and stroke styles

Both the border and stroke styles support the use of dashes. Simply changing the stroke or border style to Dashed will apply a simple dash style.

```
border: dashed 3pt rgba(127,0, 0, 0.7);
```

2_document/images/documentborderdash.png

Note: scryber supports the solid, dashed, and dotted line styles only.

10.13.8 Binding Colors and fills

As with all things scryber. The styles are all bindable to parameters and data, so regular colours could be defined and then used in places throughout the styles and components.

See binding_model for an example.

10.14 Using fonts and text styles - PD

PDF supports standard fonts. Scryber also supports the standard names of 'sans-serif', 'serif', 'monospace'.

Fonts available in the current operating system that the application has access to, can also be used. They are referenced by their postscript name.

Fonts can also be dynamically included with the @font-face rule.

Scryber supports the use of the font fallback in styles.

Note: scryber currently only supports True type &tm; and Open Type fonts - ttf (otf) and ttc (otc)

10.14.1 Built in fonts

The standard built in fonts with PDF readers that can be used in documents. These are as follows

- sans-serif / Helvetica - Regular, Bold, Italic and Bold Italic.
- serif / Times - Regular, Bold, Italic and Bold Italic.
- monospace / Courier - Regular, Bold, Italic and Bold Italic.

If used, then if the font is available it will be embedded, as is best practice.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>{@:Content.Title}</title>
    <meta name="author" content="{@:DocAuthor}" />
```

(continues on next page)

(continued from previous page)

```
    <style type="text/css">

        .std-font{
            font-size: 20pt;
            background-color:#AAA;
            padding: 4pt;
            margin-bottom:10pt;
        }

    </style>
</head>
<body style="padding: 20pt">
    <div id="first" class="std-font" style="font-family:sans-serif">
        Helvetica is the default (sans-serif) font.<br/>
        Helvetica <b>Bold</b>, <i>Italic</i>, and <span style="font-weight:bold; font-
↪style:italic">Bold Italic</span> are available.
    </div>

    <div id="first" class="std-font" style="font-family:serif">
        Times is the serif font.<br />
        Times <b>Bold</b>, <i>Italic</i>, and <span style="font-weight:bold; font-
↪style:italic">Bold Italic</span> are available.
    </div>

    <div id="first" class="std-font" style="font-family:monospace">
        Courier is the monospaced font.<br />
        Courier <b>Bold</b>, <i>Italic</i>, and <span style="font-weight:bold; font-
↪style:italic">Bold Italic</span> are available.
    </div>


</body>
</html>
```

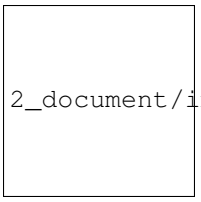
2_document/images/drawingfontsStandard.png

10.14.2 Using different fonts

Along with the standard fonts, scryber supports the systems fonts (the fonts in the Environment.SpecialFolder.Fonts). It does not support postscript font files but does support.

- ttf & otf - A truetype font file or opentype font file.
- ttc & otc - A truetype font collection (multiple styles) or open type collection

Fonts should be referred to by their Unicode Name, usually displayed through the font browser of the underlying operating system. Rather than the file name of the ttf or ttc file.



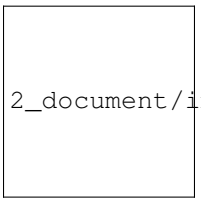
The following uses 3 different ttf fonts installed on the machine generating the document. But using the standard css font fallback if a font does not exist it can fall back to one of the known fonts

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>System Font Files</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font{
            font-size: 20pt;
            background-color:#AAA;
            padding: 4pt;
            margin-bottom:10pt;
        }

        .sans {
            font-family: 'Segoe UI', sans-serif;
        }

        .serif{
            font-family: Optima, Times, Times New Roman, serif;
        }

        .avenir{
            font-family:'Avenir Next Condensed', sans-serif;
        }

        .none {
            font-family: 'Made Up Font', monospace;
        }
    </style>
</head>
<body style="padding: 20pt">
    <div id="first" class="std-font sans">
        Segoe UI is used from a font style from the system fonts.<br />
        Segoe UI <b>Bold</b>, <i>Italic</i>, and <span style="font-weight:bold; font-
↪style:italic">Bold Italic</span> are also available.
    </div>

    <div id="first" class="std-font serif">
        Optima is used from a font style from the system fonts.<br />
        Optima <b>Bold</b>, <i>Italic</i>, and <span style="font-weight:bold; font-
↪style:italic">Bold Italic</span> are available.
    </div>

    <div id="first" class="std-font avenir">
```

(continues on next page)

(continued from previous page)

```
        Avenir Next Condensed is used from a font style from the system fonts.<br />
        Avenir Next Condensed <b>Bold</b>, <i>Italic</i>, and <span style="font-
↪weight:bold; font-style:italic">Bold Italic</span> are available.
    </div>

    <div id="first" class="std-font none">
        Fonts that are not avialable can use the fallback method.<br />
        They should also apply to <b>Bold</b>, <i>Italic</i>, and <span style="font-
↪weight:bold; font-style:italic">Bold Italic</span> styles.
    </div>
</body>
</html>
```



As the font is set to inherit, all child text components will use the specified font of the parent. If the font is changed, then all children will use the new font.

Note: .woff or woff files are not currently supported, but these can be easily converted to their ttf components online. They may be supported in future.

10.14.3 Specifying the font in code

If it is needed to set the actual font in code then internally the PDFFontSelector class can be used. It has a constructor that takes the name of the font, and an overload that can be used to chain multiple selectors together.

If the language supports it, then ther is also an explict cast available to convert to a PDFFontSelector.

```
page.Style.Font.FontFamily = new PDFFontSelector("sans-serif");
page.FontFamily = new PDFFontSelector("Arial", new PDFFontSelector("sans-serif"));
page.FontFamily = (PDFFontSelector)"Arial, sans-serif";
```

10.14.4 Font face loading

Along with the standard and system installed fonts, scryber supports the importing and declaration of custom fonts from specific files.

These can either be relative to the current file, or an absolute url.

This is also a good way of specifying various weights, as scryber (currently) only supports the bold variant. It is on our list of todo's.

```
<?xml version="1.0" encoding="utf-8" ?>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
```

(continues on next page)

(continued from previous page)

```
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Fonts loaded directly</title>
    <meta name="author" content="Scryber Team" />
    <!--
        link to google fonts API's.
      -->
    <link href="https://fonts.googleapis.com/css2?
↪family=Big+Shoulders+Inline+Display:wght@500;800&amp;display=swap" rel="stylesheet"␣
↪/>
    <style type="text/css">

        /* Open light font file on the local drive */

        @font-face {
            src: url(../../Resources/OpenSans-Light.ttf) format('truetype');
            font-family: 'Open Light';
        }

        /* Long Cang is downloaded from google but is cached locally */

        @font-face {
            font-family: 'Long Cang';
            font-style: normal;
            font-weight: 400;
            src: url(https://fonts.gstatic.com/s/longcang/v5/LYjAdGP8kkgoTec8zkRgrQ.
↪ttf) format('truetype');
        }

        .std-font {
            font-size: 20pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
        }

        /* Setting the classes to the fonts above */

        .sans {
            font-family: 'Open Light', monospace;
        }

        .grafitti {
            font-family: 'Long Cang', serif;
        }

        .broad {
            font-family: 'Big Shoulders Inline Display', sans-serif;
        }

    </style>
</head>
<body style="padding: 20pt">
    <div id="first" class="std-font sans">
        Open Sans Light is used from a font face declaration.<br />
        As we did not define <b>Bold</b>, <i>Italic</i>, or <span style="font-
↪weight:bold; font-style:italic">Bold Italic</span> they are <u>not</u> available␣
↪and will fallback.
```

(continues on next page)

(continued from previous page)

```
    </div>

    <div id="first" class="std-font grafitti">
        Long Kang is downloaded from the google fonts api.<br />
        No variations are idetnfied for the <b>bold</b> or <i>italic</i> are␣
↪available.
    </div>

    <div id="first" class="std-font broad">
        Big shoulders is used from a css file imported from the google fonts.<br />
        It does have a <b>Bold</b> variation, but not <i>Italic</i>.
    </div>

</body>
</html>
```

2_document/images/drawingfontsStyles.png

Warning: The link for the font css from google is not XHTML compliant. The & parameter separator should be escaped to & and the link tag closed '/>'

10.14.5 Text styles and decoration

Along with the bold and italic variants, scryber also supports underlines, strikethrough and overline text rendering features. As with HTML these are default styles, and can be altered as needed.

- **Bold**
  -
  -
  - css {font-weight:bold;}
- **italic**
  -
  -
  - css {font-style:italic;}
- **Underline**
  - <u></u>
  - <ins></ins>
  - css { text-decoration:underline;}
- **StrikeThrough**
  - <strike></strike>

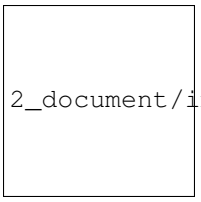
  - <del></del>
  - css { text-decoration: line-through; }

- **Overline**
  - css { text-decoration: overline; }

As with css text-decoration values can be combined e.g. 'line-through underline' , and the decorations will flow across lines.

Scryber does not (currently) support the text-decoration-color or text-decoration-style.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Fonts decorations</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font {
            font-size: 20pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
        }

        .railway{ text-decoration: overline underline; color: red;}

    </style>
</head>
<body style="padding: 20pt">
    <div id="first" class="std-font">
        Segoe UI is used from a system font<br />
        <strong>Strong is Bold</strong>, <em>Em(phasis) is Italic</em>.
        <ins>Ins(ert) is underlined</ins> and <del>del(eted) is strike through.</del>
    </div>

    <div id="first" class="std-font">
        The decorations can be combined by multiple tags<br />
        Such as <b><em><u>Bold italic underlined</u></em></b>
        or by the style <span class="railway" >over and under lined.</span>
    </div>

    <div class="std-font" style="font-weight:bold; text-decoration: underline;" >
        The decoration will flow down into child tags.
        <div style="margin:0 30pt 0 30pt; font-size:12pt">And any inner content can
            <span style="font-weight:normal; text-decoration: overline;">override the␣
↪settings</span>
            as needed.
        </div>
    </div>
</body>
</html>
```



10.14.6 Line height (leading)

The leading is the height of the lines including ascenders and descenders. The default is set by the font (usually about 120% of the font size), but can be manually adjusted as needed.

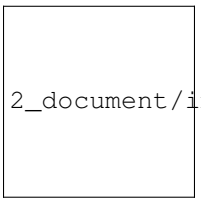
Inline components will ignore the block level style for leading. The value must be a unit value rather than a relative percent.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Line Height</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font {
            font-size: 20pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
            font-family: Geneva, Verdana, sans-serif;
        }

        .big{
            font-size: 40pt;
        }

        .high{
            line-height: 50pt;
        }

    </style>
</head>
<body style="padding: 20pt">
    <div id="first" class="std-font">
        Lines will follow the standard line height<br/>
        Set by the font for general reading.<br/>
        <span class="big">If a larger font-size is used, then this will increase the␣
↪line height.</span>
        Rolling back to the default size on any following new lines.
    </div>

    <div class="std-font high" >An explicit line height can be used to increase (or␣
↪decrease) the
    leading. <span class="big">This is not affected by a change in the font size</
↪span> and will continue
```

(continues on next page)

(continued from previous page)

```
    to maintain a standard height.</div>

    <div class="std-font" style="line-height: 22pt" >It is also allowed to be less␣
↪than
    the <span class="big">size of the font text</span> although this does affect␣
↪readability.</div>
</body>
</html>
```



10.14.7 Multi-byte Characters

Scryber supports multi-byte characters, anywhere in the document. Whether that is only a couple of characters, or whole paragraphs.

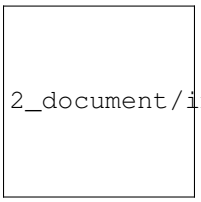
Note: The font used must also support the charcter glyphs that need to be drawn. If they are not in the font, then they cannot be rendered by the reader.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Line Height</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font {
            font-size: 20pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
            font-family: 'Microsoft JhengHei UI', sans-serif;
        }

        .big{
            font-size: 40pt;
        }

        .high{
            line-height: 50pt;
        }

    </style>
</head>
<body style="padding: 20pt">
```

(continues on next page)

(continued from previous page)

```
    <div class="std-font">




        <br />
        <br />
        <span>




        </span>
    </div>
    <!-- mixed character sets, with leading and spacing -->
    <div class="std-font">
        We can intermix the characters

        <span style='font-family:"Segoe UI";'>

            But the font must contain the glyphs.
        </span>
    </div>
</body>
</html>
```



Note: Due to the size of most unicode font files with thousands of glyphs, using and embedding a unicode font can dramatically increase the size of the pdf file. The example above came in at 23Mb without any images. Beware!

10.14.8 Right to Left

Scryber doesn't currently support Right to left (or vertical) typography. At the moment we have have not seen it done anywhere due to limitiations in postscript and the pdf document. But we will keep trying.

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.15 Securing Documents - PD

Scryber supports securing documents with restrictions (printing, copying and modifying) along with password protection.

10.15.1 Restrictions

The document restrictions are managed through the header meta tag with the name print-restrictions or print-encryption.

```
<meta name="print-restrictions" content="allow-forms allow-printing"  />
<!-- optional print encryption settings -->
<meta name="print-encryption" content="128bit" />
```

These restrictions can be applied directly to a document without an owner password if the parser-mode is Lax (default). However this will be a randomly generated password, and unretrieveable. (Although the document will still open and be secured).

If the parser-mode is strict then an error will be thrown. See (extending_logging)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">

<html lang='en' xmlns='http://www.w3.org/1999/xhtml'
    title="Root">
<head>
    <meta charset='utf-8' name='author' content='Richard Hewitson' />
    <title>{@:title}</title>
    <meta name="print-restrictions" content="allow-forms allow-printing" />
    <meta name="print-encryption" content="128bit" />
</head>
<body class="grey" title="Outer">
    <p class='highlight' title="Inner"
    style="margin:20pt; border-bottom:solid 1px red; padding:4px; margin-top:20pt;␣
↪font-family:sans-serif;  ">
    This pdf should only allow forms and printing, not copying or modifying without␣
↪the owner password.</p>
    <p>{@:title}</p>
</body>
</html>
```

2_document/./images/documentRestrictions.png

10.15.2 Restriction values

The following values are supported on the print-restrictions meta tag.

- allow-printing - The end user will be able to print the document.
- allow-accessibility - The end user can use accessibility tools to read the document.
- allow-annotations - The end user can add annotations and comments.
- allow-copying - The end user can copy content from the document
- allow-modifications - The end user can modify the document and add remove pages etc.
- allow-forms - The end user can complete forms, and sign content.

- all - All restrictions are applied.

The print-encryption values supported are

- 40bit - lowest level, and most basic restrictions.
- 128bit - more secure and supports enhanced restrictions (default if no print-encryption is set).

10.15.3 Adding Passwords

The best use of the document restrictions is when providing password(s) at generation time.

There are 2 passwords used

- owner - to unlock the restrictions and allow all actions.
- user - to allow opening of the document.

They can both be the same, if wanted, and are set on the Document.PasswordProvider property, through the implementation of an IPDFSecurePasswordProvider

The quickest set up is to use the DocumentPasswordProvider in Scryber.Secure. This can be initialized with one or two strings, or secure strings.

```
using (var doc = Document.ParseDocument(path))
{
    doc.PasswordProvider = new Scryber.Secure.DocumentPasswordProvider("Password",
↪"Password");
    doc.Params["title"] = "Hello World";

    using (var stream = DocStreams.GetOutputStream("ProtectedHtml.pdf"))
    {
        doc.SaveAsPDF(stream);
    }

}
```

Warning: The use of simple strings as in memory passwords onto documents could be a security risk, and the IDisposable System.Secure.SecureString is more appropriate.

10.15.4 Implementing the IPDFSecurePasswordProvider

The secure password provider interface is a trivial matter of implementing a class that can set some document password settings, based on the document path. The way they are loaded and how they are retuned is up to the implementor.

If the provider returns false then the document will not be secured, otherwise it will use the IDocumentPasswordSettings as needed.

```
//using System.Security

public interface IPDFSecurePasswordProvider : IDisposable
{
    /// <summary>
    /// Implementers should use this method to load specific security settings pased␣
↪on a specific path and return required values
    /// </summary>
```

(continues on next page)

(continued from previous page)

```
    /// <param name="documentpath">The source path the document was loaded from</
↪param>
    /// <param name="settings">Set to the security settings to be associated with␣
↪this document if IsSecure returns true.</param>
    /// <returns>Return true to assign security settings, or false to not.</returns>
    bool IsSecure(string documentpath, out IDocumentPasswordSettings settings);

}


public interface IDocumentPasswordSettings : IDisposable
{
    /// <summary>
    /// Gets the Owner Password for the document.
    /// If not set, then it must be set in code before a secure document is output
    /// </summary>
    public SecureString OwnerPassword { get; }

    /// <summary>
    /// Gets the User Password for the document. If null then NO password is required␣
↪to open and view the document
    /// </summary>
    public SecureString UserPassword { get; }

}
```

10.16 Processing instruction and logging - TD

We have seen 2 ways to create the resultant document from a template.

For MVC projects the extension methods for the controller allow assignment to a stream as a `FileResult`, and for console or GUI applications as an actual file on a file system `SaveAsPDF`.

10.17 Scryber Trace log details - PD

Internally scryber has lots of logging that can be used to understand what is going on underneath, without having to debug the libraries This is achieved with the scryber processing instruction that needs to be at the top of the file.

10.17.1 Example without logging

If we have the below example for our template.

```
<!DOCTYPE HTML >
<html xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>Hello World</title>
        <link href='../css/printstyles.css' rel='stylesheet' />
    </head>
    <body>
        <div style='padding:10px; text-align:center'>
            <img src='../images/sitelogo.png'>
```

(continues on next page)

(continued from previous page)

```
            <p>Hello World from scryber.</p>
        </div>
    </body>
</html>
```

Then when we generate, we will get the following output

2_document/images/HelloWorldNoLogging.png

It is there, but there is no image, and it's not looking like we expected.

However if we add our scryber processing intruction to append the log for messages then we see so much more going on.

10.17.2 Example with logging

```
<?scryber append-log='true' log-level='Messages' parser-log='true' ?>
<!DOCTYPE HTML >
<html xmlns='http://www.w3.org/1999/xhtml' >
    <head>
        <title>Hello World</title>
        <link href='../css/printstyles.css' rel='stylesheet' />
    </head>
    <body>
        <div style='padding:10px; text-align:center'>
            <img src='../images/sitelogo.png'>
            <p>Hello World from scryber.</p>
        </div>
    </body>
</html>
```

Then when we generate, we will get the following output

2_document/images/HelloWorldWithLogging.png

Here we can see that there are another 3 pages added to the document - and quite a few errors. Specifically the stylesheet could not be found, and nor could the image.

10.17.3 Scryber processing instruction

The following are the supported options on the processing instruction.

- **'append-log' - Controls the tracing log output for a single document**
  - false - This is the default and the document will be rendered as normal.

  - true - If set to true, then once the document has been generated, a trace log of output will be appended to the resultant file, containing all the recorded entries.

- **'log-level' - This is an enumeration of the granularity of the logging performed on the pdf file. Values supported (from lea**

  - Off - no entries be recorded.
  - Errors - only errors will be recorded (depending on the parser mode switch)
  - Warnings - warnings will occur if some of the contents cannot be loaded, or the parsing fails for a non-error condition.
  - Messages - This will output key stage messages for the generation of the file.
  - Verbose - A quantity of messages will be output for each of the compoents, and is a useful level to understand what is going wrong (if anything) with your document.
  - Diagnostic - This will generate a large log file and can slow the creation of a PDF file significnatly. But it's very informative.

- **'parser-log' - Controls the logging from the xml parser.**
  - true - then both the reading of the content, to create the document, as well as the output of the content to PDF will be recorded.
  - false - then only messages from the content creation and output will be recorded.

- **'parser-culture' - specifies the global culture settings when parsing a file for interpreting dates and number formats in the**

  - en-gb - This specifies the english, britsh culture. It can be useful for reading number formats or dates from files e.g.
  - es-es - This will read spanish nuber formats where . 'dot' is a thousand separator and , 'comma' is the decimal separator.

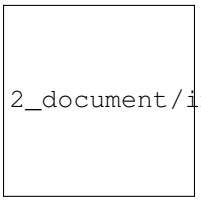
- **'parser-mode' - Defines how errors will be recorded if unknown or invalid attributes values are encountered.**

  - Strict - Will raise exceptions to the top of the stack and must be handled in your code. (Good for dev)
  - Lax - Default. If this is set then the parser is more complianant, where errors will be logged, but not cause the output to fail. (Good for Prod).

Note: If you set the log level to Diagnostic for the Hello World example, the appended log file is around 10 pages in length. If it's a long document - diagnostic is going to hurt.

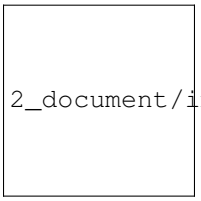
10.17.4 Tracing Details

There is some really good information available in the tracing output not just in the logging, but also on the metrics and overview.



The top section will give information on the versions, file sizes and generation time (for the document without the logging).

The middle section will give information on timings for each type fo activity. If the trace level is Verbose (or Diagnostic) then the performance metrics will detail specific areas, for eaxmple below we can see that the loading of the google font(s) was causing our template to increase generation time by 110 milliseconds to load the font css. Luckily the font files themselves are cached and did not need to be reloaded each time. But we could save that time by using a local css.



10.18 Document outline (or bookmarks) - PD

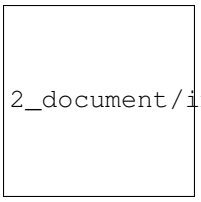
Structing a document also allows for the outline. This is effectively a table of contents available in PDF readers that support quick navigation of the whole document, to go to the section or content needed.

In Scryber the title attribute is converted to display as the outline of the document. Nested content will be under any parent title.

10.18.1 Simple Outline

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML>

<html xmlns='http://www.w3.org/1999/xhtml' >
<head>
    <link type="text/css" rel="stylesheet" href="https://fonts.googleapis.com/css2?
↪family=Roboto:ital,wght@0,100;0,700;1,100&amp;display=swap" />
    <style>

        body {
            font: 12pt 'Roboto';
            padding: 20pt;
        }

    </style>
</head>
<body title="Top Level" style="padding:20pt;">
    <div>
        <h1 title="Heading" >This is a heading</h1>
    </div>

    <section title="Page 2" >
        <h1 title="Heading 2" >This is the heading on the second page.</h1>
    </section>
</body>
</html>
```

When output the reader application or browser can show the content of the outline. Selecting any of the bookmark items should navigate directly to the page the content is on.



10.19 Class Hierarchy - PD

When working with the code or adding objects to the code once parsed it is good to understand the hierarchy.

Scryber has a top level of Components. These create the basic level of document structure.

10.19.1 Base Component Classes

The base classes form the foundation of the functionality for each of the main concrete classes. It's much easier to create your own funtionality using one of these classes.

- Component - Implements the IPDFComponent interface along with IPDFBindable and
- ContainerComponent - Holds child instances. Implements the 'InnerContent' collection as a protected property and the IPDFContainer.Contents implementation. Also passes the lifecycle methods to children.
- **VisualComponent - Extends a container to be styled with the IPDFStyledComponent interface and a lot of properties for i**
  - Panel - Base class for the standard container componnents (Div, Span, etc) that implements the IPDFViewPortComponent interface to do the laying out of content.
  - PageBase - Base class for all page types.
  - ImageBase - Base class for all the image types.
  - TextBase - Base class for the textual components and implements the IPDFTextComponent interface.
  - ShapeComponent - Base class for drawing components, implementing the IPDFGraphicPathComponent interface.

All components inheriting from VisualComponent have a virtual method for GetBaseStyle() which returns the default style for that component before anything is applied.

The PDFObjectType used in constructors is simply a 4 character struct that can identify the type of component, and can also be used for generating ID's. It can be directly cast from a string value or const string.

Note: Any custom classes should include a parameterless constructor if they sould be parsed as part of some other xml/xhtml content.

10.19.2 Collection classes

Scryber maintains a bi-directional structure to the document content graph. When a component is added as a child to a container, that child's Parent value is updated to the container. As such, each child knows it's parent, page and document.

In order to maintian this the ContainerComponent creates the ComponentList class with itself as an owner.

Scryber then provides the abstract ComponnetWrappingList and ComponentWrappingList<T> classes for stronger typing on the contents of the collection and implementing the ICollection<T> interface.

e.g. The TableGrid has a Rows property of TableRowCollection which inherits from ComponentWrappingList<TableRow> and wraps the ContainerComponent.InnerContent.

```
public class TableGrid : ContainerComponent
{
    //Strongly typed collection of Rows that will have their parent set automatically.
    public TableRowList Rows
    {
        get
        {
            if (this._rows == null)
                this._rows = new TableRowList(this.InnerContent);
            return this._rows;
        }
    }

}

//The strongly typed collection for TableRows.
public class TableRowList : ComponentWrappingList<TableRow>
{
    public TableRowList(ComponentList content)
        : base(content)
    {
    }
}
```

10.19.3 Concrete Component Classes

All the components are in the Scryber.Components namespace and inherit from VisualComponent

- **PageBase**
  - **Page - a single page by default style, with a Header and Footer.**
    - Section - allows multiple layout pages by default, with a continuation header and footer.
  - PageGroup - a set of PageBases instances but also a Header, Footer and continuation header and footer that are passed down.
- **Panel**
  - Div - basic concrete panel implementation with contents and a full width.
  - BlockQuote - basic block quote implementation with a custom style of 10pt margins.
  - Canvas - a container where all content will be relatively positioned and the content clipped to the canvas bounds.
  - **List - contains inner list items.**
    - ListOrdered - a list that has a default decimal numbering.
    - ListUnordered - a list that has a default bullet addornment.
    - ListDefinition - a list that has terms and content.
  - ListItem - the individual items in a list.

  - UserComponent - allows the dynamic loading of content from a remote source.
  - Paragraph - a block of inner content, with a 4pt margin at the top as a default style.
  - Preformatted - a block of inner content, with a default style for rendering code.
- **TextBase**
  - Date - renders the current of defined date in a specific format.
  - Number - renders a numeric value in any specific format.
  - PageNumberLabel - renders the current page (along with totals) in any specific format.
  - PageOfLabel - renders the page number of another component.
- TextLiteral - A non-visual component for text strings, including assigment within the constructor.
- TableGrid - A layout of content in a tabular way.
- TableRow - A single row of cells within a grid.
- TableCell - the final content of the cells in a table grid.
- **ShapeComponent**
  - HorizontalRule - basic flat line.
  - Line - Line that supports a position and size.
  - Path - Complex path definition with M(oves), L(ines to) etc.
  - **PolygonBase**
    * Polygon - Multi-sided shape with style.
    * Rectangle - A 4 sided shape with style.
    * Triangle - Just the 3 sides.
    * Ellipse - A box bounded circle or ellipse with style.
- PageBreak - Forces the flow onto the next page if possible.
- ColumnBreak - Forces the flow onto the next column or page if possible.
- LineBreak - Forces the flow onto a new line.

10.19.4 Html Classes

When parsing content from HTML the document component graph will be constructed from subclasses of the main components in the Scryber.Html.Components namespace.

```
namespace Scryber.Html.Components {

    [PDFParsableComponent("div")]
    public class HTMLDiv : Scryber.Components.Div
    {

    }
}
```

The HTML components generally map directly onto their superclass Component, with added support for the html specific attributes (title, hidden, etc).

10.19.5 Layout content

In the creation of a PDF document, the components above are used to create the actual layout items. These are much more basic, but know how to generate the pdf content streams and data used by PDF readers.

If a document has a Page, and then a Section with 2 page breaks - the layout will be 4 pages long with all the text and runs in the respective pages.

If needed any component can implement or override the IPDFViewPortComponent interface and return a new LayoutEngine for that component. The LayoutEngineBase and LayoutEnginePanel are good starting points to layout your own custom content.

- PDFLayoutDocument - Top level holding font references, image resource references and the list of layout pages.
- PDFLayoutPage - A single page of a content block, with an optional header content block and or footer content block, and any absolutely positioned regions.
- PDFLayoutBlock - A grouping of one or more column regions along with any relatively positioned regions, that will render the style.
- PDFLayoutRegion - A single continuous set of lines and/or other blocks.
- PDFLayoutLine - A single line of content runs.
- **PDFLayoutRun - A single lightweight atomic graphical content operation.**
  - **PDFTextRun - Textual operation**
    - PDFTextRunBegin - Start of the text, includes setting the font etc.
    - PDFTextRunCharacter - Text Drawing operation
    - PDFTextRunNewLine - Simple line break operation
    - PDFTextRunProxy - Placeholder for text to come from the owning component.
    - PDFTextRunEnd - Completion of text.
    - PDFTextRunSpacer - Offset of a line run to allow for other content.
  - PDFLayoutXObject - Renders PDF content as a separate stream, return the reference to that stream.
- PDFLayoutComponentRun - allows the owning component to render it's own content explicitly (e.g. Paths).

10.19.6 Content Styles

The style classes are based around a dictionary of inherited and direct style item keys with storongly typed style value keys. All of the standay ones are defined in the Scryber.Styles.StyleKeys static class.

If a style value is inherited, the it will be copied to any descendent element (e.g. FontFamily) and any direct value will only be used on the component it is defined on (e.g. BackgroundColor)

Implementor can create their own style items and keys as needed using the static constructor methods with distinct object types (use mixed case to ensure they are unique).

```
const bool INHERITED = true;
var tocStyle = StyleKey.CreateStyleItemKey((PDFObjectType)"Ctoc", INHERITED);
var tocLeader = StyleKey.CreateStyleValue<LineStyle>((PDFObjectType)"Ctld", tocStyle);
```

This can then be used on any style definition or styled component to get or set a value, it can be bound to a value, and as it is inherited, will flow down with the content (merged).

```
var styleDefn = new StyleDefn();
styleDefn.SetValue(tocLeader, LineStyle.Dotted);

LineStyle default = LineStyle.None;
var defined = styleDefn.GetValue(tocLeader, default);

if(styleDefn.TryGetValue(tocLeader, out defined)
{
    //Do something with the defined style.
}
```

The style class hierarchy is as follows.

- **StyleBase - root abstract class that holds the actual values.**
  - **Style** [StyleBase - the main class used on components themselves directly.]
    - StyleDefn : Style - has a class matcher property that will ensure that this style is only applied to Components that match.
    - StyleFull : Style - a readonly, locked set of style values with known values - position, font, padding etc.
  - StyleGroup : StyleBase - a collection of style base items, that can be treated as one item in an outer collection.

The document has a Styles property which is a StyleCollection, so any of the above can be added to the the document. Each VisualComponent has a Style property where these values can be directly applied.

The flow for creating a full style for a component is linear.

1. The GetBaseStyle returns a new instance the standard style for a component. 1. If the component inherits from a super class VisualComponent then it should call the base.GetBaseStyle() and apply any styles to that before returning. 1. The GetAppliedStyle is then called with the base style. 1. This traverses up the component hierarchy, finally reaching the document. 1. The document calls MergeInto on its style collection with the base style. 1. Each style within the collection is MergedInto the style. 1. If that style is a StyleDefn it is checked to make sure it is matched, before being merged. 1. If that style is a StyleGroup, the it calls MergeInto on its own collection of styles. 1. If it should be merged, then each style value is assessed to see if it exists and compares the priority. 1. If the style that should be merged is a higher priority then the value is replaced. 1. We then come back to the original component and any direct styles are applied to to orginal base. 1. Once this is done it is pushed onto the StyleStack, where the hierarchy of styles from parent components are. 1. And finally a full style is built based on inherited and direct values. 1. That full style is retained and used through the rest of the layout and rendering.

Despite the number of steps, the build of styles is usually not an issue, compared to extracting font files, image binary data or encrypting streams. However for some documents with a large number or containers e.g. a very long table with many rows it can become the limiting factor as well as memory intensive.

The template element automatically caches the style for each of the inner contents, rather than building every time. This can speed the generation, but if it causeing issues can be switched off using the data-cache-styles=false attribute. This will force the styles to be built each and every time.

```
<templalate data-bind='{@:Model.Items}' data-cache-styles='false' >
    <tr>
        <td class='desc-cell' >{@:.Description}</td>
        <!-- can be applied individually so that they are cached -->
        <td class='val-cell' data-style-identifier='boundcellValue' >{@:.Value}</td>
    </tr>
</template>
```

10.19.7 Why and when to implement

A lot of the time, it is easier to use compound components to build all the main characteristics of the content needed. However sometimes there is a need to use explict functionality or capabilities that are not currently available.

At scryber we also use this framework extensively to provide new top level features with safe knowledge the lower engine layers can deal with the grunt work.

See extending_logging and extending_scryber along with namespaces_and_assemblies for more on this.

Scryber does not rely on xml / xhtml, but it makes life easier and is more visual and structured.

When ever you parse a Document or component you are simply creating the same as you could in code.

10.20 XHTML Template

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml' >

<head>
    <title>HTML Document</title>
    <style>
        .grey{ background-color: grey; }
    </style>
</head>

<body class="grey" title="Page 1">
    <p title="Inner">Hello World, from scryber.</p>
</body>
</html>
```

10.21 The same in code

```
var doc = new Document();
doc.Info.Title = "Coded Document";

var style = new StyleDefn(".grey");
style.Background.Color = (PDFColor)"grey";
doc.Styles.Add(style);

var pg = new Page();
pg.StyleClass = "grey";
pg.OutlineTitle = "Page 1";

var para = new Paragraph();
para.Contents.Add(new TextLiteral("Hello World From scryber"));

pg.Contents.Add(para);
doc.Pages.Add(pg);

return doc.ProcessDocument();
```

10.22 The same in XLinQ

```
XNamespace html = "http://www.w3.org/1999/xhtml";
XElement root = new XElement(html + "html",
    new XElement(html + "head",
        new XElement(html + "title", "XElememt Document"),
        new XElement(html + "style", ". grey { background-color: gray;}")
    ),
    new XElement(html + "body",
        new XAttribute("class", "grey"),
        new XAttribute("title", "Page 1"),
            new XElement(html + "p",
            new XAttribute("title", "Inner"),
            new XText("Hello World from Scryber"))
        )
    );

string basePath = string.Empty;
doc = Document.ParseDocument(root.CreateReader(), basePath, ParseSourceType.
↪DynamicContent);
```

Note: The use of the base path allows relative images and styles to be used from a reader in dynamic content.

10.23 Loading partial content

Scryber supports the loading of partial content (not a whole document) through the use of the instance method ParseTemplate(), or the Document static Parse method.

There are a number of overloads for the parse template method that use streams, text readers and xml readers. But as a simple loading mechanism it can be useful to include some custom stored content.

```
//This content can be loaded from any source.

var content = "<p xmlns='http://www.w3.org/1999/xhtml' >" +
                "This <b>Is my content</b>" +
                "</p>";

using (var reader = new StringReader(content))
{
    var comp = doc.ParseTemplate(doc, reader) as Component;
    (doc.Pages[0] as Page).Contents.Add(comp);
}
```

The use of the first component argument in ParseTemplate is to provide the source path for any relative references. It can also be called with the owner and a base path.

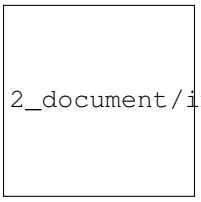
10.24 Why use one over the other

We always think that the declarative is better for what you need, but sometimes building in code works. See the document_code_classes for a break down of the class heierarchy.

In this documentation, we will concentrate on the use of the declarative html with code where appropriate, but remember that everything that is declared can be coded too.

10.25 Lifecycle of a document creation - PD

During the creation and output of a document, there are a number of stages that the processing goes through.



10.25.1 Wrapping it up

Using the wrapper methods for *PDFDocument.ProcessDocument(output)* or the MVC *this.PDF(doc)* all the stages will be automatically executed and the resulting content rendered to a stream.

It makes the output of a document simple and discreet.

```
using(var doc = PDFDocument.Parse(path))
{
    doc.ProcessDocument(output);
}
```

10.25.2 Initialize and Load

The initialize and load method, *InitializeAndLoad()*, makes sure that the document components are ready. It allows opportunity to set up the defaults, prepare for loading of resources, and perform any start up operation.

After initialization and loading external code should be able to interact and alter the propperties etc.

10.25.3 Data Binding

The data binding stage is important for the creation of any content from a datasource. There is a flag on the document class (see reference/pdf_document) that defines whether databinding should automatically be done during the lifecycle.

By default this is set to true, but if manual invocation of databinding is required, then it can be set to false and databing can be invoked later.

10.25.4 Layout and Rendering

The output stage is rendering the content to a stream or file. This is where all the compoments are laid-out to their text runs, image binarty details, fonts and string measured etc.

And then finally they are written to the data stream (compressed and ordered).

10.25.5 Disposing

It's a good idea to dispose of your document once created. Scryber itself does not use any unmanaged resources, but other components or extensions build could allocate significant memory blocks, leading to a leak over time.

10.25.6 Manual Execution

```
using(var doc = PDFDocument.Parse(path))
{
     //make any component changes needed

     doc.InitializeAndLoad();

     // between each of the stages

     doc.DataBind();

     // access any components in templates that
     // are created during databinding

     // all set and ready to output

     doc.RenderToPDF(output);
}
```

10.25.7 Reuse of a document object

It is not recommended to load or create a documment and execute the rendering multiple times. Because state and output information can be stored on the components themselves it may have unpredictable output.

The parsing or creation of a document object is not usually the most exhaustive process.

10.26 Splitting into multiple files - PD

For large documents or projects, it's often easier to split your templates into multiple files. These can be separate stylesheets, pages, components and the top level document.

As a converions the files should have the following extensions.

- Documents - [MyTemplate].html
- Stylesheets - [MyStyles].css
- Pages and content - [MyInnerContent].html

It just makes life easier.

10.26.1 4 file example

As an example we can split a single document into 4 files. Here we will take the top level document and reference a stylesheet, a page header component and a cover page.

10.26.2 DocumentRefs.html

At the top level is the Document - *DocumentRefs.html*

```
<?xml version="1.0" encoding="utf-8" ?>
<?scryber append-log='true' log-level='verbose' ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>My multi file document</title>
    <!-- Stylesheet links -->
    <link rel="stylesheet" media="screen" href="./css/includeScreen.css" />
    <link rel="stylesheet" media="print" href="./css/includePrint.css" />
</head>
<body style="margin:20pt; font-size:20pt">
    <header>
        <embed src="./fragments/pageheader.html" />
    </header>

    <section style="page-break-before:avoid; page-break-after:always;">
        <embed src="./fragments/coverpage.html">
    </section>

    <div>
        <h1 class="title" >This is the second page</h1>
    </div>

</body>
</html>
```

This contains a reference to *includeScreen.css* and *includePrint.css* in the *css* folder. As the first link is specified as for screen only it will not be loaded, and only the includePrint.css will be loaded.

An embedded reference to a *PageHeader.html* in the *Fragments* folder for a standard document header, and a reference to a *CoverPage.html* in the *Fragments* folder for the cover page content.

The path references are relative to the current document (but could be absolute urls)

10.26.3 includePrint.css

This is the content of the *includePrint.css*

```
.title{
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    font-weight:bold;
    font-size:60pt;
    margin: 20pt;
    padding:10pt;
    text-align:center;
}

.page-head {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    font-weight: normal;
    font-size: 14pt;
```

(continues on next page)

(continued from previous page)

```
    margin: 20pt 10pt 10pt 10pt;
    padding: 10pt;
    border-bottom:solid 1pt black;
}
```

This file declares 2 style classes that can be applied to any element with class names *title* and *page-head* For more info about styles see document_styles

10.26.4 CoverPage.html

This is the content of the *CoverPage.html*, which will be directly included in the content of the document, so should not start with the HTML of body tag, but go directly to the actual content used.

As this is intended to be the first page, and always a page, the page-break-before and page-break-after have been switched.

The namespace is important on includes, just as with top-level documents, the namespace is **critical**

```
<?xml version="1.0" encoding="utf-8" ?>
<div xmlns='http://www.w3.org/1999/xhtml' >
    <h1 class="title">Heading Page</h1>
</div>
```

Note: These are just samples and can be as complex as you like, but to be good xml it should still only have a single root.

10.26.5 PageHeader.html

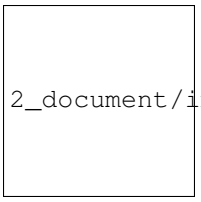
The component is referenced from link in the *DocumentRefs.pdfx*. This file is just used as the content for the header of the pages.

```
<?xml version="1.0" encoding="utf-8" ?>
<div  xmlns='http://www.w3.org/1999/xhtml' class="page-head" style="column-count:2">
    <span class="head-text" style="break-after:always;" >Referenced File Example</
↪span>
    <time date-format="dd MMM yyyy" />
</div>
```

The content could be anything, but for this time we are using it as a standard header. It has 2 columns with a title on one side and then a date label on the other.

10.26.6 Bringing it all together

These are all the files, and we just need to generate them. All being well, then when we bring it together we will get a 2 page document with consistent headers and content.



The styles are used across all content even referenced files, and the layout flows just as you would expect.

10.26.7 Circular references

Scryber will not allow circular references. i.e. files that reference either themselves, or other files that reference back to the original as it could create an infinie parsing loop.

Whilst a file can be embedded from multiple places in multiple documents, each time it will be loaded as new content. Once loaded changes to one instance will not affect any other instances loaded from that file.

10.26.8 iFrame support

Along with the embed option, scryber supports the use of iFrames with a src.

```
<iframe src='Fragments/PageHeader.html' />
```

The frame is not isolated, or independent of the main document, and styles will be transferred down into the content of the frame. This gives the wrong usage impression - but is supported as a tag element.

10.27 Components overview - TD

10.27.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

See the tables_reference for more details on what is supported in tables.

10.27.2 Binding Simple numbers

10.27.3 Binding Dates and Times

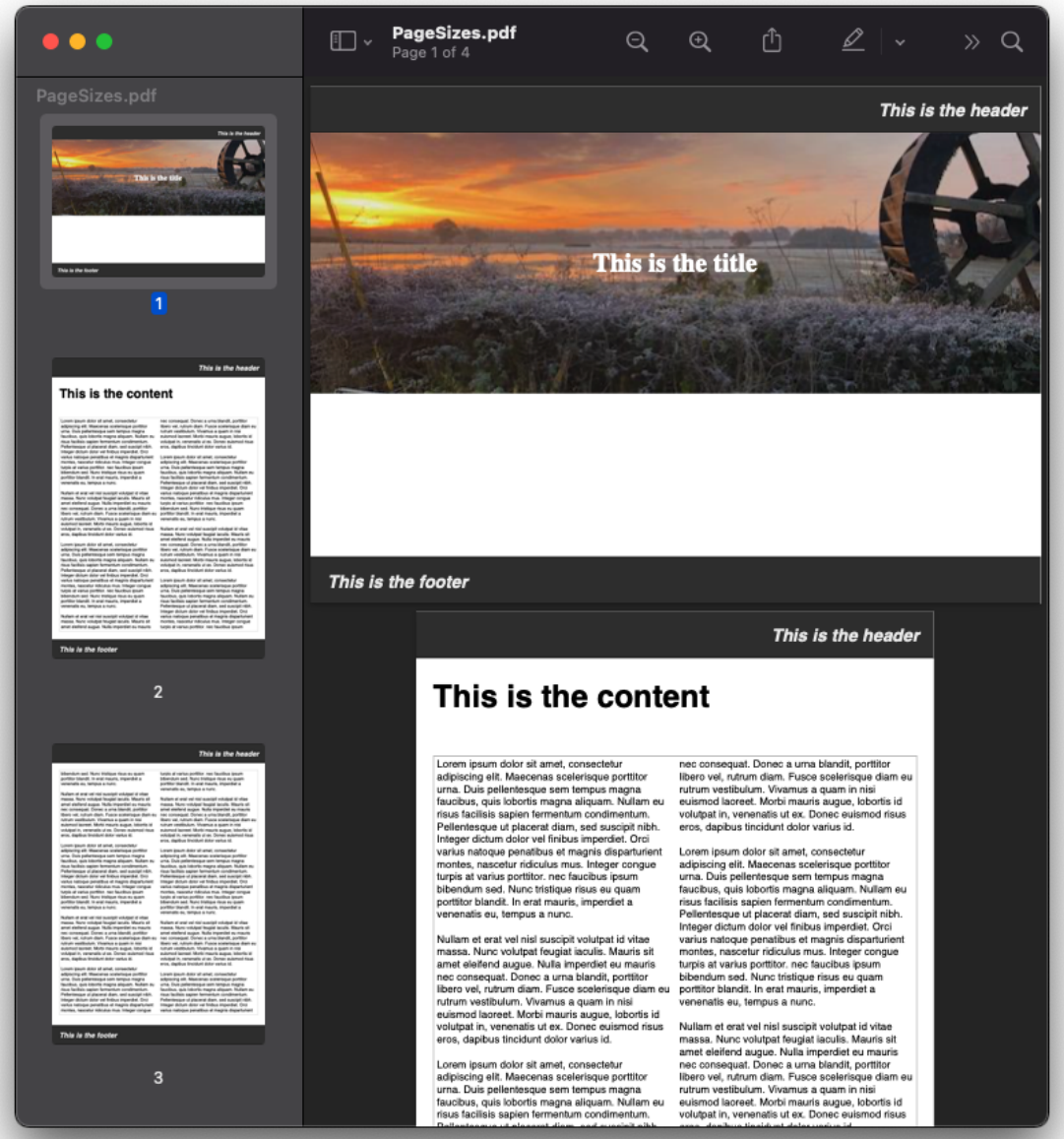
10.27.4 Calculating values

10.27.5 Building in code

10.28 Body, Pages, breaks and sizes

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it.

The use of the *page-break-before* or *page-break-after* is supported on any content to force a new page when set to 'always' on any component tag


PageSizes.pdf
Page 1 of 4
This is the header
This is the title
This is the footer
This is the content
1
2
3

The body has an optional header and footer that will be used on every page if set.

Scryber also supports the use of the @page rule to be able to change the size and orientation of each of the pages either as a whole, or within a section or tag.

```
<body>
    <header>Header on every page</header>
    <div>On the first page</div>
    <div class='next-page' >On the second page in landscape</div>
</body>
```

```
@page{ size: A4 portrait }

@page landscape { size: A4 landscape }

.next-page {
    page: landscape;
    page-break-before: always;
}
```

In code a document can have `Page``s, ``Section``s and ``PageGroup``s added to it that allow inner content to be split over different parts of the document. A styled component can also have it's ``Style.Page.BreakBefore` or `Style.Page.BreakBefore` set to *true* and flow onto a new page (if allowed).

```
using(var doc = new Document())
{
    var sect = new Section();

    var div1 = new Div();
    div1.Contents.Add(new TextLiteral("On the first page"));
    sect.Contents.Add(div1);
    doc.Pages.Add(sect);

    sect = new Section();
    sect.PaperSize = PaperSizes.A4;
    sect.PaperOrientation = PaperOrientation.Landscape;

    var div2 = new Div();
    div2.Contents.Add(new TextLiteral("On the second page"));
    sect.Contents.Add(div2);
    doc.Pages.Add(sect);
}
```

10.28.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

10.28.2 The body and its content

A body section has a structure of optional elements

- header - Optional, but always sited at the top of a page
- Sited between the Header and Footer is any content to be included within the page.

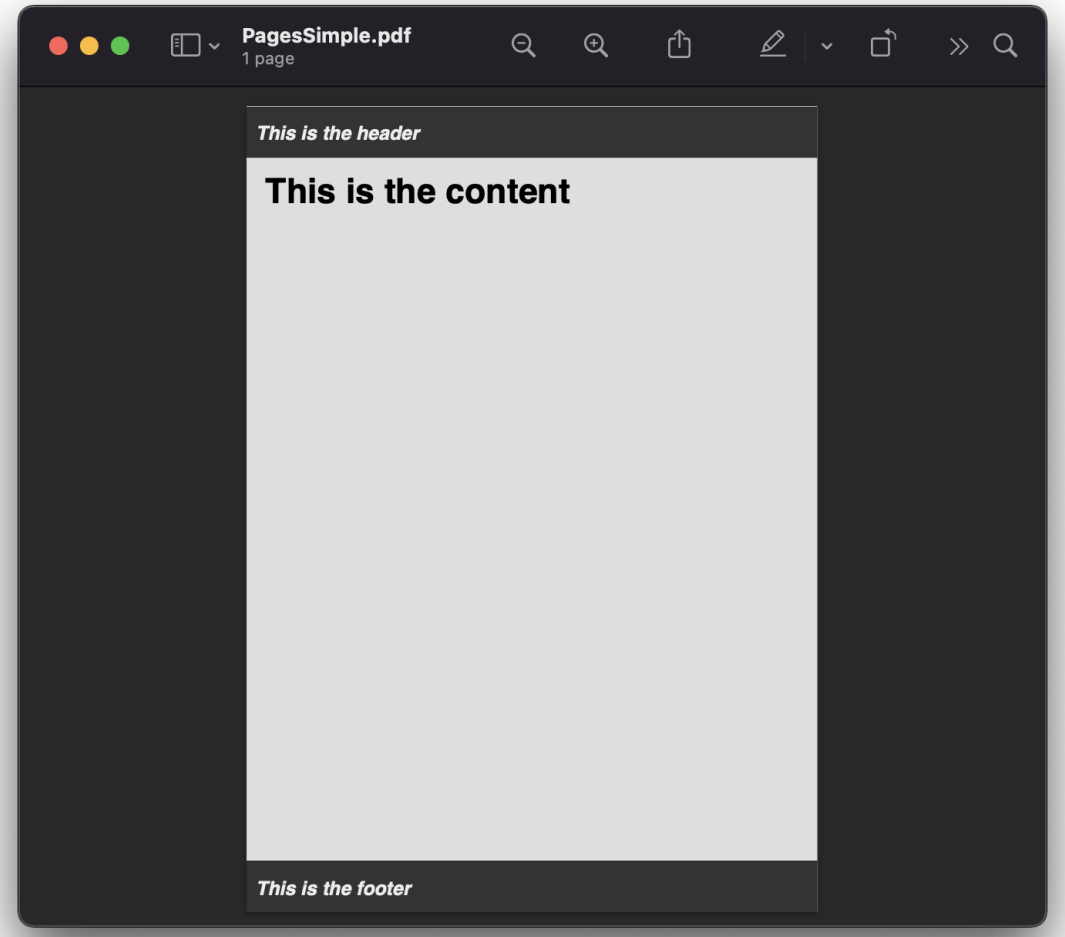
- footer - Optional, but always sited at the bottom of a page

If a page has a header or footer the available space for the content will be reduced. Headers and footers can contain any content in the same way as any other block.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style>

    body {
        background-color: #DDD;
    }

    header, footer{
            padding: 10pt;
            background-color: #333;
            color: #EEE;
            border-bottom: 1px solid black;
            border-top: 1px solid black;
    }

    h1{
        padding: 20pt;
    }

    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>
    <h1>This is the content</h1>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```

```
//Scryber.UnitSamples/PagesSamples.cs

public void SimpleNavigationLinks()
{
    var path = GetTemplatePath("Pages", "PagesSimple.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Pages", "PagesSimple.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version


PagesSimple.pdf
1 page
This is the header
This is the content
This is the footer

Note: Any styles set on the body will be applied to the header and footer as well. e.g. padding or margins. But they can have their own (overriding) styles as well.

10.28.3 Single body structure

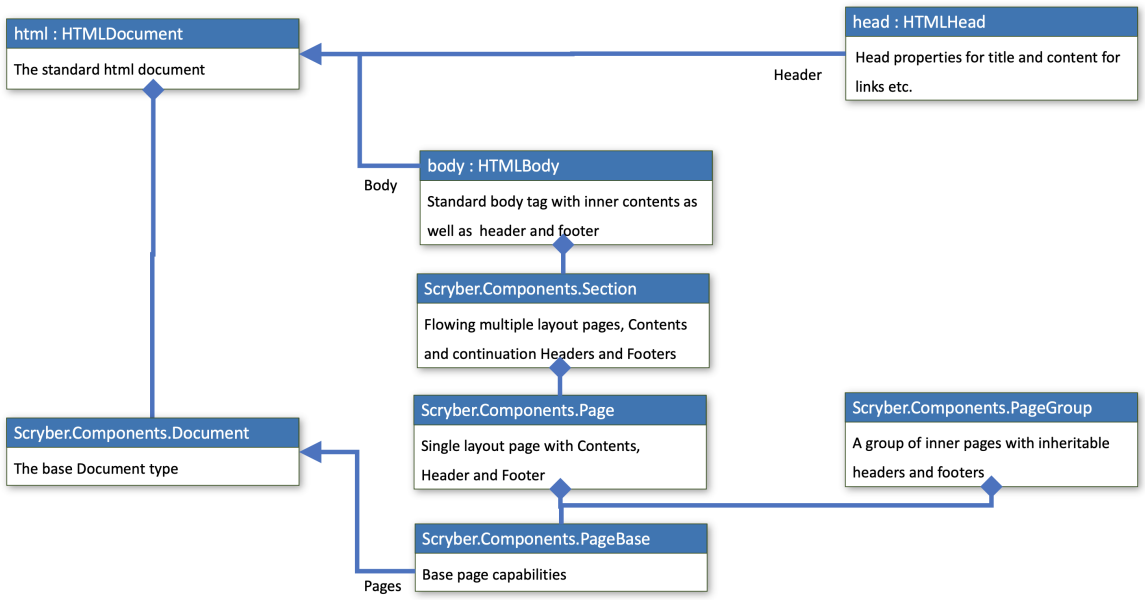
In the example above the `html` tag references the `Scryber.Html.Components.HTMLDocument` class that inherits from the `Scryber.Components.Document` class.

See ../overview/scryber_parsing for more information on how instances are created from elements.

The `HTMLDocumemt` has 2 properties on it for the *head* (`HTMLHead`) and *body* (`HTMLBody`) that are matched to the content of the template.

The `HTMLBody` inherits from the `Scryber.Components.Section` which in itself inherits from the `Scryber.Components.Page` class.and supports multiple pages, and then the `Scryber.Components.PageBase` that all page components should inherit from.

The `HTMLHead` is a specific html component that wraps the title and *Contents* for links, styles etc.



Full size version

10.28.4 Flowing Pages

If the size of the content is more than can fit on a page it will overflow onto another page. Repeating any header or footer.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style>
```

(continues on next page)

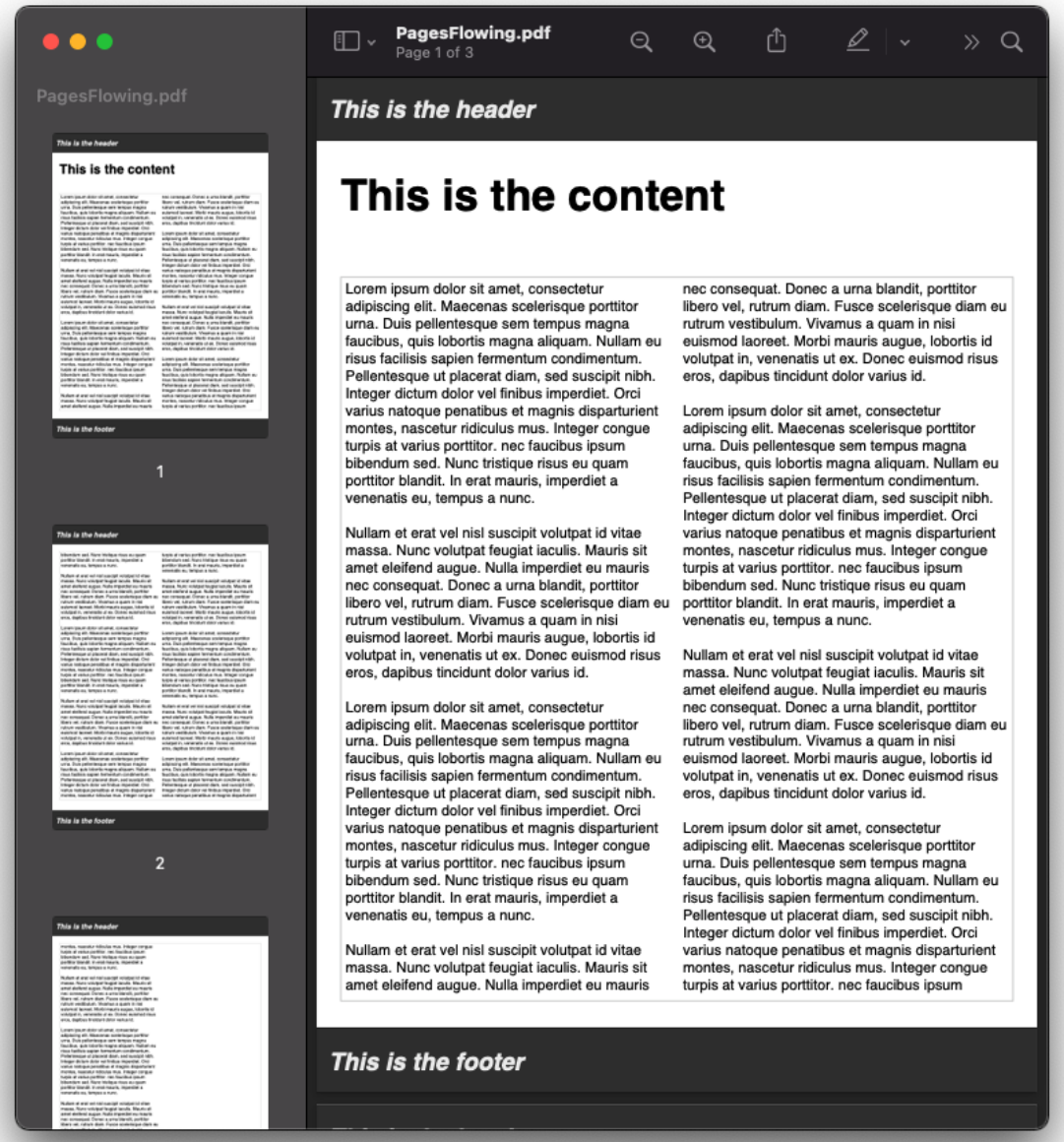
(continued from previous page)

```
        header, footer {
            padding: 10pt;
            background-color: #333;
            color: #EEE;
            border-bottom: 1px solid black;
            border-top: 1px solid black;
        }

        body h1, body div {
            margin: 20pt;
        }

        body div.content {
            font-size: 12pt;
            padding: 4pt;
            border: solid 1px silver;
            column-count: 2;
        }
    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>
    <h1>This is the content</h1>
    <!-- main content in the document
        bound from the parameter 'content' -->
    <div class='content' style="white-space: pre-wrap">{{content}}</div>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```

Loading a long text file and binding to the *content* parameter, we use the `white-space: pre-wrap` style so the carriage returns are preserved, but the text will flow in the columns and over multiple pages.

```
//Scryber.UnitSamples/PagesSamples.cs

public void PagesFlowing()
{
    var path = GetTemplatePath("Pages", "PagesFlowing.html");

    var txtPath = GetTemplatePath("Pages", "LongTextFile.txt");
    doc.Params["content"] = System.IO.File.ReadAllText(txtPath);

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Pages", "PagesFlowing.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Here we can see that the content flows naturally onto the next pages, including the padding and borders. And the header and footer are shown on the following pages.



Full size version

10.28.5 Page breaks

Using the *page-break-before: always* and *page-break-after: always* css properties, we can force content onto a new page in the flow.

In this example we have set up a `h1` to force the break after so the rest of the content will be on a new page.

```
body h1.title {
    page-break-after : always;
}
```

The breaking can be at any depth, and borders; padding; margins; etc. should be preserved.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style>

        header, footer {
            padding: 10pt 20pt 10pt 20pt;
            background-color: #333;
            color: #EEE;
            border-bottom: 1px solid black;
            border-top: 1px solid black;
        }

        header{
            text-align: right;
        }

        body h1, body div {
            margin: 20pt;
        }

        body div.content {
            font-size: 12pt;
            padding: 4pt;
            border: solid 1px silver;
            column-count: 2;
        }

        /* title page with background image
            and page-break-after */
        body h1.title{
            background-image: url(../../images/landscape.jpg);
            background-size: cover;
            font: 30pt serif;
            color: white;
            height: 300pt;
            margin: 0;
            vertical-align:middle;
            text-align:center;
            page-break-after: always;
        }

    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>

    <!-- title content that forces a
    page break after -->
```

(continues on next page)

(continued from previous page)

```
    <h1 class="title">
        This is the title
    </h1>

    <h1>This is the content</h1>
    <div class='content' style="white-space: pre-wrap">{{content}}</div>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```

```
public void PagesBreaks()
{
    var path = GetTemplatePath("Pages", "PagesBreaks.html");

    using (var doc = Document.ParseDocument(path))
    {
        var txtPath = GetTemplatePath("Pages", "LongTextFile.txt");
        doc.Params["content"] = System.IO.File.ReadAllText(txtPath);

        using (var stream = GetOutputStream("Pages", "PagesBreaks.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

10.28.6 Page sizes

The default page size for a layout in scryber is A4 portrait. Scryber supports the use of the `@page` directive to alter the size of the layout page in the document.

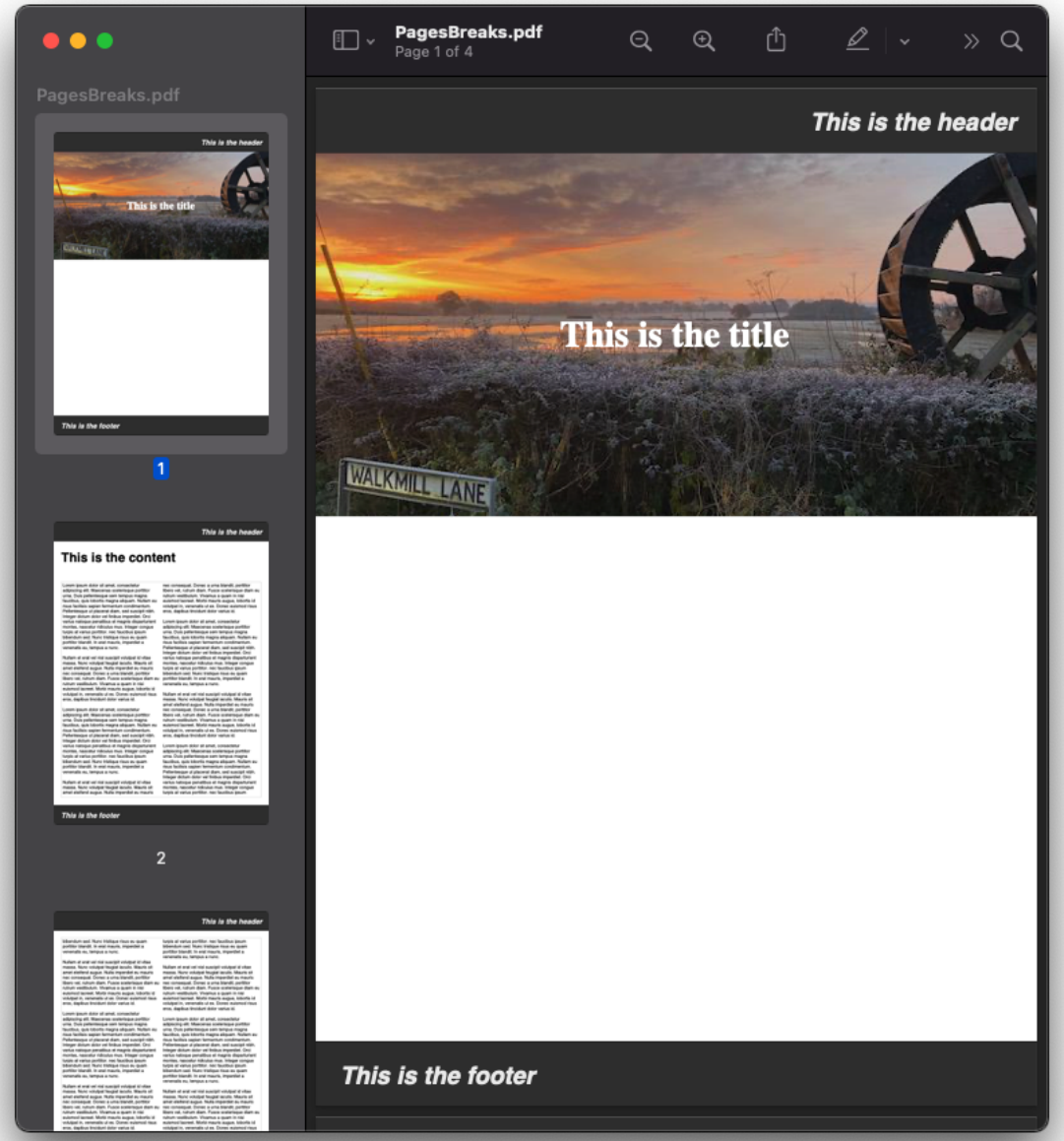
```
@page {
    size: A4 landscape;
}
```

This will change **all** the pages to use landscape layout.

To define specific page sizes the *@page* directive can be followed by a label and then that label applied to the style of the component that is currently forcing a new page.

```
@page main-body {
    size: A4 portrait;
}

.main {
    page: main-body;
```

(continues on next page)


PagesBreaks.pdf
Page 1 of 4
This is the header
This is the title
WALKMILL LANE
This is the footer
This is the content
1
2

(continued from previous page)

```
    page-break-before: always;
}
```

Note: As the layout page will be created when a page-break property css is met, the *page* property should be set at that level. This means that a component that has the page-break-after property, should also stipulate which page size to use.

Scryber supports the use of the following page sizes.

- **ISO 216 Standard Paper sizes**
  - A0 to A9
  - B0 to B9
  - C0 to C9
- **Imperial Paper Sizes**
  - Quarto, Foolscap, Executive, GovermentLetter, Letter, Legal, Tabloid, Post, Crown, LargePost, Demy, Medium, Royal, Elephant, DoubleDemy, QuadDemy, Statement,

But custom values can be used for a specific width or height on the *size* property.

```
@page {
    size: 200mm 200mm;
}
```

Putting this together with the example above, the title page uses the default A4 landscape size, and following pages use the portrait size.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style>

        header, footer {
            padding: 10pt 20pt 10pt 20pt;
            background-color: #333;
            color: #EEE;
            border-bottom: 1px solid black;
            border-top: 1px solid black;
        }

        header{
            text-align: right;
        }

        body h1, body div {
            margin: 20pt;
        }

        body div.content {
            font-size: 12pt;
            padding: 4pt;
            border: solid 1px silver;
            column-count: 2;
```

(continues on next page)

(continued from previous page)

```
        }

        body h1.title{
            background-image: url(../../images/landscape.jpg);
            background-size: cover;
            font: 30pt serif;
            color: white;
            height: 300pt;
            margin: 0;
            vertical-align:middle;
            text-align:center;
        }

        /* The main will force a new page
            of style main-content
        */
        body h1.main{
            page-break-before: always;
            page: main-content;
        }

        /* default */
        @page {
            size: A4 landscape;
        }

        /* main content specific */
        @page main-content {
            size: A4 portrait;
        }

    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>

    <h1 class="title">
        This is the title
    </h1>

    <!-- this now forces a break before and
        specifies the page orientation of portrait -->
    <h1 class="main">This is the content</h1>

    <div class='content' style="white-space: pre-wrap">{{content}}</div>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```

```
public void PagesSizes()
```

(continues on next page)

(continued from previous page)

```
{
    var path = GetTemplatePath("Pages", "PageSizes.html");

    using (var doc = Document.ParseDocument(path))
    {
        var txtPath = GetTemplatePath("Pages", "LongTextFile.txt");
        doc.Params["content"] = System.IO.File.ReadAllText(txtPath);

        using (var stream = GetOutputStream("Pages", "PageSizes.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }
}
```

Full size version

10.28.7 Creating pages in code.

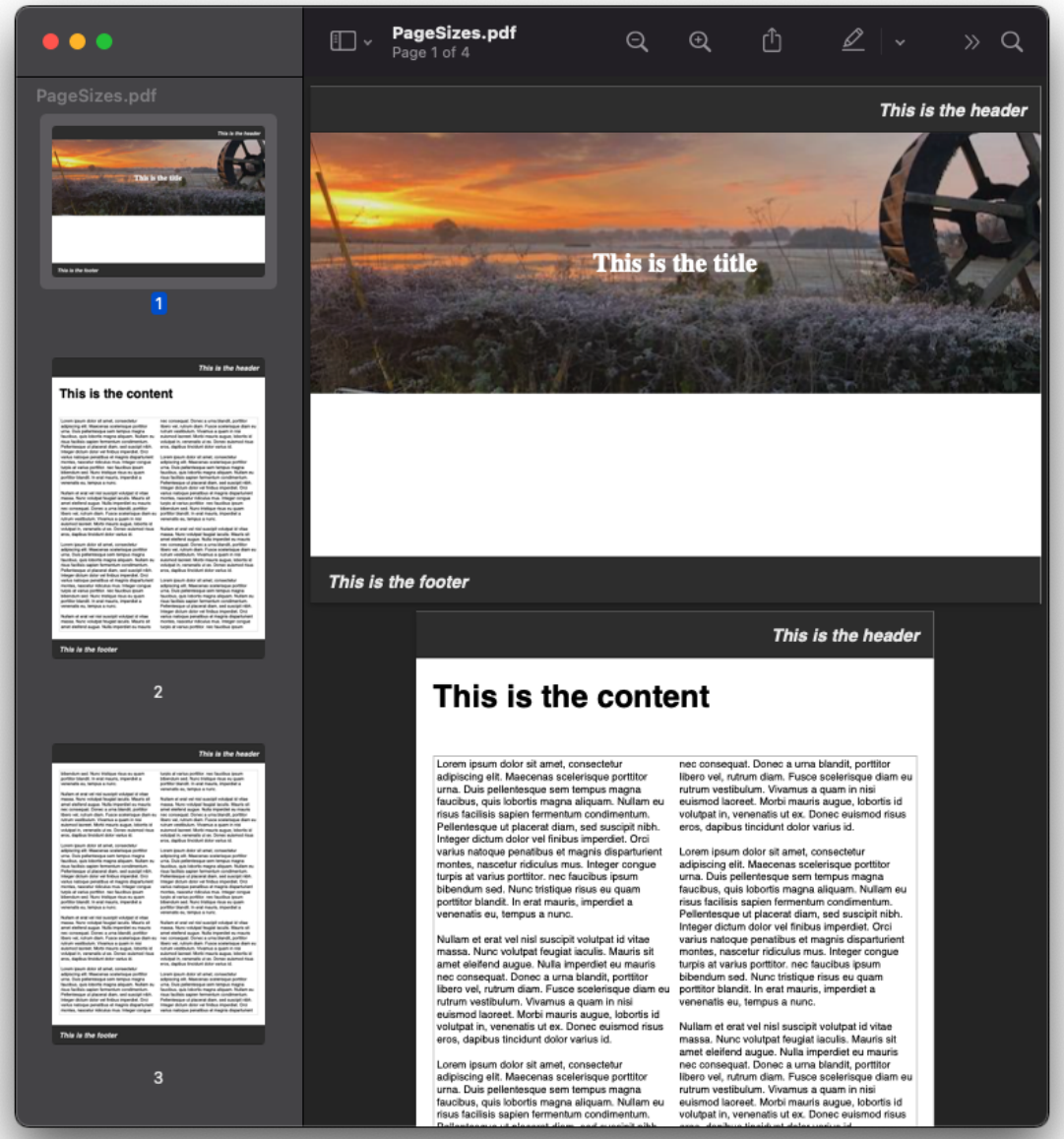
As with everything else in scryber, it is simple and easy to create pages in code from the document and pagebase classes.

It is also possible to insert pages, sections and page groups to an existing parsed template. As the `body` inherits from `Scyber.Components.Section` this will be parsed as a single section.

For headers and footers, these are supported through the `IPDFTemplate` interface. See page_headers_reference for more on this topic.

```
public void PagesCoded()
{
    using(var doc = new Document())
    {
        //Define the title style that matches onto the '.title' style class.
        var titleStyle = new StyleDefn(".title");

        titleStyle.Background.ImageSource = "../../../Images/Landscape.jpg";
        titleStyle.Background.PatternRepeat = PatternRepeat.Fill;
        titleStyle.Position.VAlign = VerticalAlignment.Middle;
        titleStyle.Position.HAlign = HorizontalAlignment.Center;
        titleStyle.Size.Height = 300;
        titleStyle.Font.FontSize = 30;
        titleStyle.Fill.Color = PDFColors.White;
        titleStyle.Font.FontFamily = new PDFFontSelector("serif");


        //Define the body style that matches onto the '.body' style class
        var bodyStyle = new StyleDefn(".body");
        bodyStyle.Font.FontSize = 12;
        bodyStyle.Padding.All = 10;
        bodyStyle.Border.Color = (PDFColor)"#AAA";
        bodyStyle.Columns.ColumnCount = 2;

        var textStyle = new StyleDefn(".preserve");
        textStyle.Text.PreserveWhitespace = true;

        //Add the styles to the document
```

(continues on next page)


PageSizes.pdf
Page 1 of 4
This is the header
This is the title
This is the footer
This is the content
1
2
3

(continued from previous page)

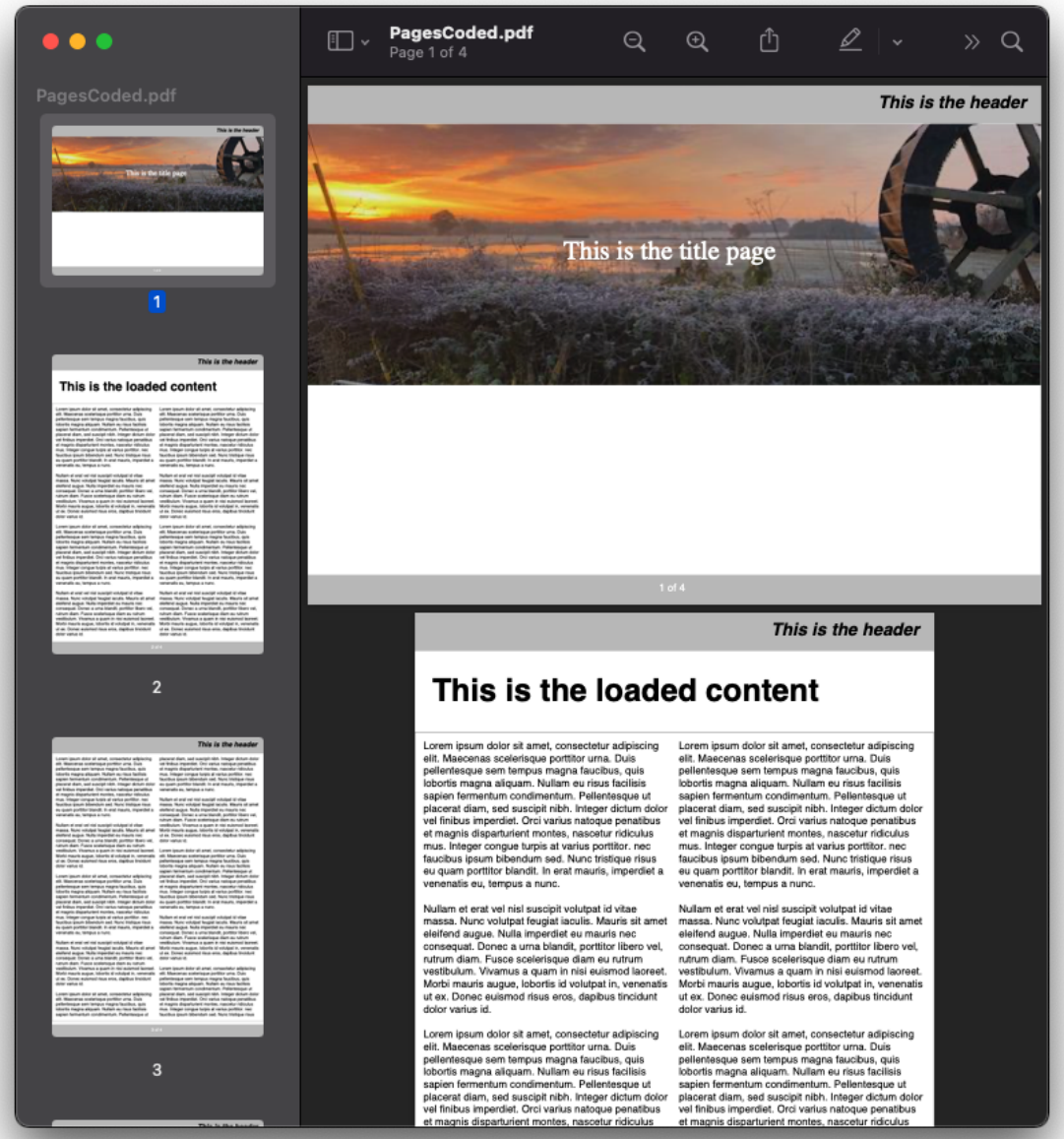
```
        doc.Styles.Add(bodyStyle);
        doc.Styles.Add(titleStyle);
        doc.Styles.Add(textStyle);

        //Create a page with a size
        var pg = new Page()
        {
            PaperSize = PaperSize.A4,
            PaperOrientation = PaperOrientation.Landscape
        };

        //add it to the document Pages collection
        doc.Pages.Add(pg);

        //Create new instances of the header and footer classes that implement
        //The IPDFTemplate interface and set to the header and footer.
        pg.Header = new CodedHeader();
        pg.Footer = new CodedFooter();

        //Create the title div and add it to the first page
        var div = new Div();
        div.StyleClass = "title";
        pg.Contents.Add(div);

        //With some text in it.
        var txt = new TextLiteral("This is the title page");
        div.Contents.Add(txt);

        //Now add a section to the document
        var sect = new Section()
        {
            PaperOrientation = PaperOrientation.Portrait
        };
        doc.Pages.Add(sect);

        //Set the header and footer (to the same as the page)
        sect.Header = pg.Header;
        sect.Footer = pg.Footer;

        //Add a header
        var contentTitle = new Head1() { Text = "This is the loaded content", Margins␣
↪= new PDFThickness(20) };
        sect.Contents.Add(contentTitle);

        //And add the body content to the section.
        var body = new Div();
        //Add the body class, and preserve so extra returns are retained
        //Will still wrap text.
        body.StyleClass = "body preserve";
        sect.Contents.Add(body);

        //Read some long plain text from a file into a text literal
        var path = GetTemplatePath("Pages", "LongTextFile.txt");
        var content = new TextLiteral();
        content.Text = System.IO.File.ReadAllText(path);

        //We set the style to preserve, so that the white space in the content is␣
↪retained
```

(continues on next page)

(continued from previous page)

```
        content.StyleClass = "preserve";

        //Add it to the body.
        body.Contents.Add(content);

        //And process in the same way
        using (var stream = GetOutputStream("Pages", "PagesCoded.pdf"))
        {
            doc.SaveAsPDF(stream);
        }
    }


}

/// <summary>
/// IPDFTemplate for the header
/// </summary>
private class CodedHeader : IPDFTemplate
{
    public IEnumerable<IPDFComponent> Instantiate(int index, IPDFComponent owner)
    {
        return new IPDFComponent[]
        {
            new Head4(){
                Text = "This is the coded header",
                Padding = new PDFThickness(10, 20, 10, 20),
                Margins = PDFThickness.Empty(),
                BackgroundColor = PDFColors.Silver,
                HorizontalAlignment = HorizontalAlignment.Right
            }
        };
    }
}

/// <summary>
/// IPDFTemplate for the footer
/// </summary>
private class CodedFooter : IPDFTemplate
{
    public IEnumerable<IPDFComponent> Instantiate(int index, IPDFComponent owner)
    {
        var div = new Div() {
            BackgroundColor = PDFColors.Silver,
            FillColor = PDFColors.White,
            FontSize = 12,
            HorizontalAlignment = HorizontalAlignment.Center,
            Padding = new PDFThickness(10)
        };
        div.Contents.Add(new PageNumberLabel() { DisplayFormat = "{0} of {1}" });

        return new IPDFComponent[] { div };
    }
}
```

Full size version


PagesCoded.pdf
Page 1 of 4
PagesCoded.pdf
This is the header
This is the title page
1
This is the header
This is the loaded content
2
This is the header
3
This is the header
This is the loaded content
1 of 4

10.28.8 Coded page breaks

The components in code support the page break before and page break after style.

```
content.Style.Page.BreakBefore = true;
```

To add an explicit page break in a `Section` the `PageBreak` component can be added to the content.

```
var pbreak = new PageBreak();
body.Contents.Add(pbreak);

//this can also be disabled with the Visible property

pbreak.Visible = false;
```

10.29 Pages Numbers

Putting numbers in pages is often a requirement, but honestly we have never liked the CSS approach.

At scryber we have taken a slightly more declarative approach with the 'page' tag. Browsers do not understand this tag, and will ignore it. The scryber engine will understand and output the current page number

```
<!-- xmlns='http://www.w3.org/1999/xhtml' -->

<!-- current page number -->
<page />

<!-- total number of pages -->
<page property='total'>

<!-- page number of another component -->
<page for='#id' />

<!-- page with custom format -->
<page data-format='Page {0} of {1}' />
```

The page number component classes are `PageNumberLabel` and `PageOfLabel` in the `Scryber.Components` namespace.

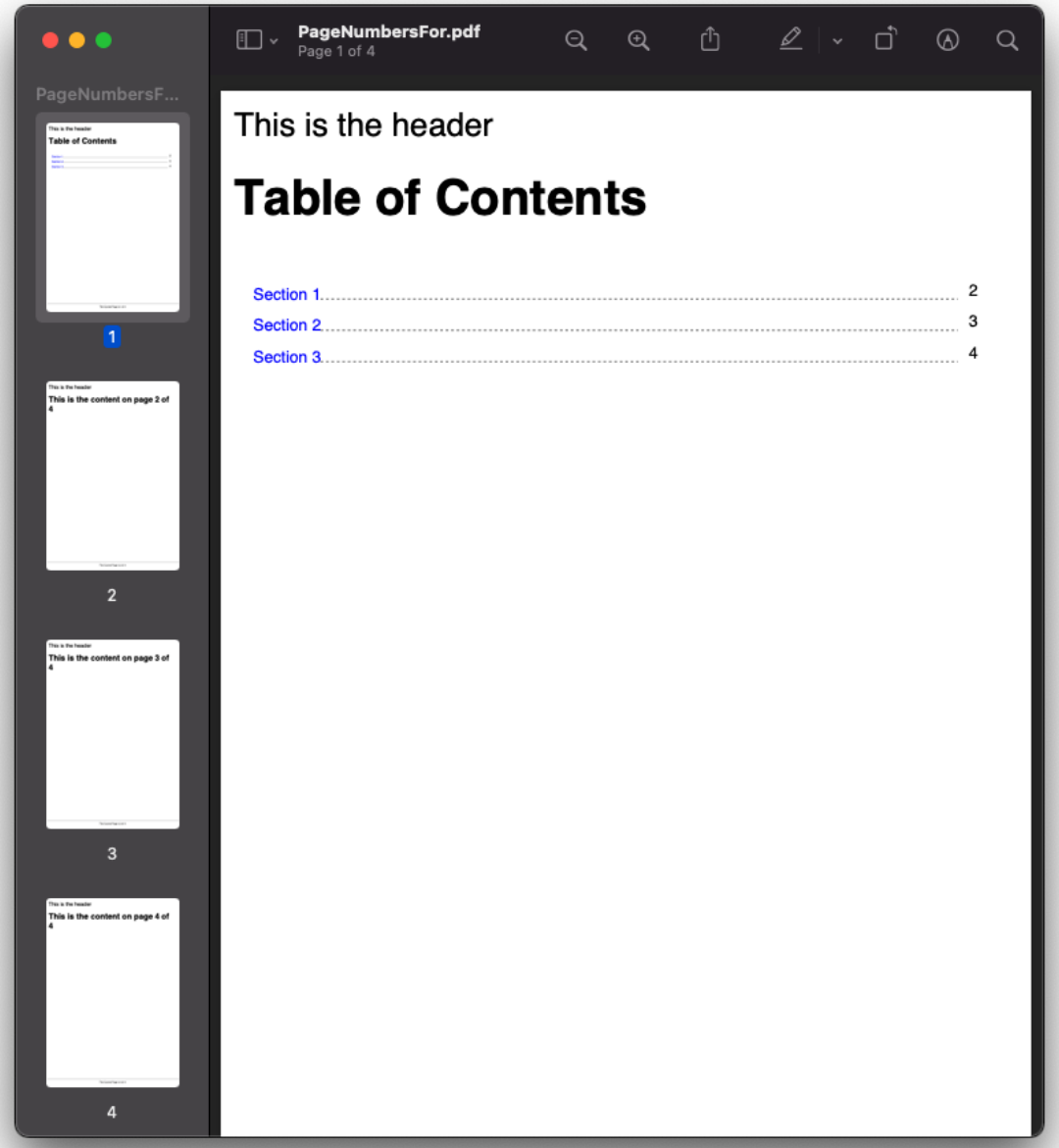
```
//using Scryber.Components

var pgNum = new PageNumberLabel(){ DisplayFormat = "{0} of {1}" };

var pgOf = new PageOfLabel() { ComponentName = "#id", NotFoundText = "Oops!" };
```

10.29.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

10.29.2 Current Page Numbers

The <page/> tag can be placed anywhere within the text of a document, and will render in the current style. Although it does also support the inline style options from as with any other span.


PageNumbersFor.pdf
Page 1 of 4
PageNumbersF...
This is the header
Table of Contents
Section 1 2
Section 2 3
Section 3 4
1
2
3
4

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>My Document</title>
    <style>

        p, h1 {
            padding: 10pt;
        }

        .print-only{
            display:none;
        }

        @media print{

            .print-only{ display: block; }

            .foot{
                border-top: solid 1pt gray;
                text-align:center;
                font: 10pt sans-serif;
                margin: 5pt;
            }
        }

    </style>
</head>
<body>
    <header>
        <p>This is the header</p>
    </header>

    <!-- Page number within the content -->
    <h1>This is the content on page <page /></h1>

    <footer>
        <!-- a page number using the current font style in a footer -->
        <p class='print-only foot'> The Current Page is <page /></p>
    </footer>

</body>

</html>
```

```
public void CurrentPageNumber()
{
    var path = GetTemplatePath("PageNumbers", "PageNumbersCurrent.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Links", "PageNumbersCurrent.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
```

(continues on next page)

(continued from previous page)

```
}
```

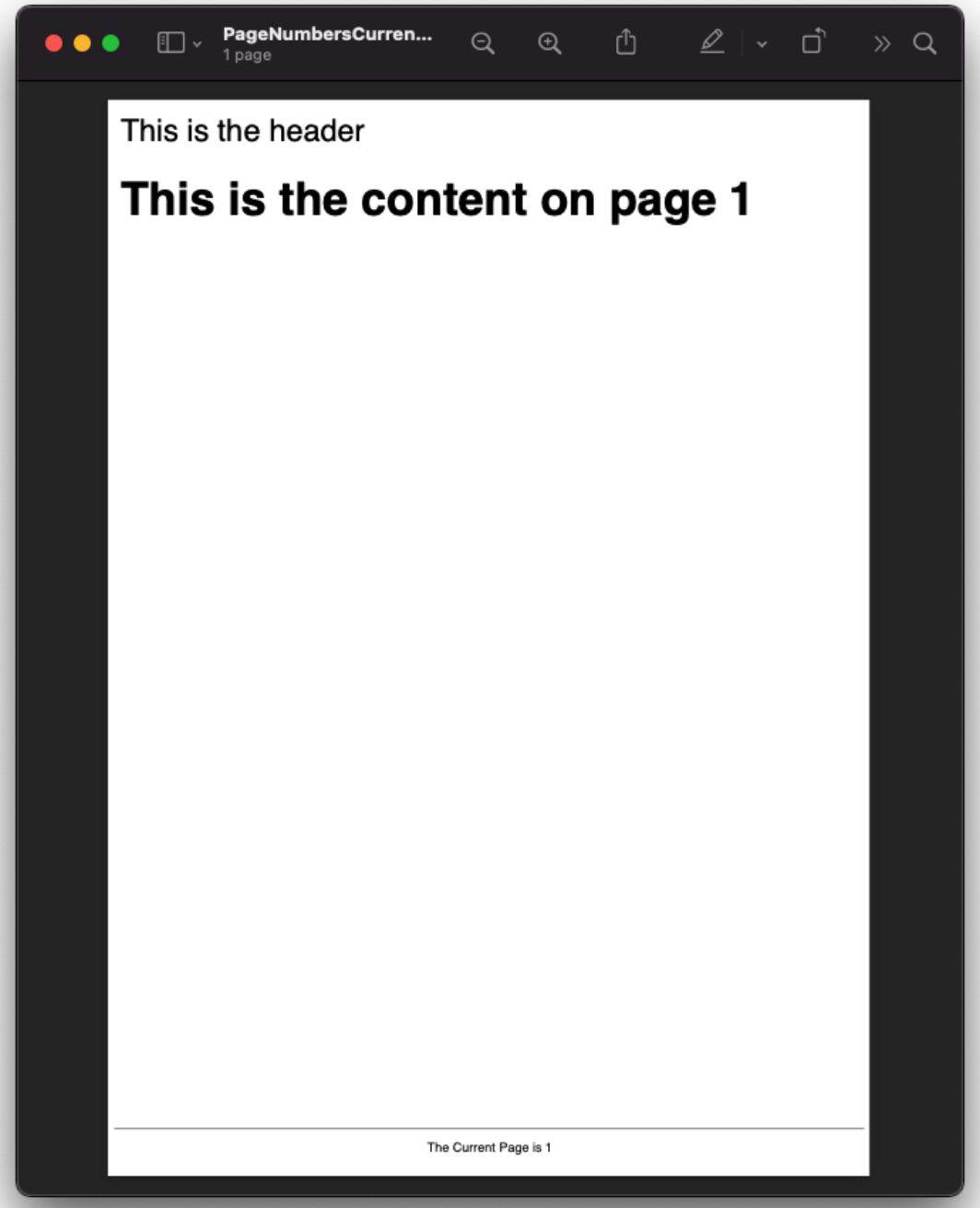
Full size version

10.29.3 Total number of pages

The page tag also supports the property attribute for displaying the 'total' number of pages.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>My Document</title>
    <style>

        p, h1 {
        padding: 10pt;
        }

        .print-only{
        display:none;
        }

        @media print{

            .print-only{ display: block; }

            .foot{
            border-top: solid 1pt gray;
            text-align:center;
            font: 10pt sans-serif;
            margin: 5pt;
            }

            .break{
            page-break-before:always;
            }
        }

    </style>
</head>
<body>
    <header>
        <p>This is the header</p>
    </header>
    <h1 id='First'>This is the content on page <page /> of <page property='total' /></
↪h1>
    <h1 id='Second' class='break'>This is the content on page <page /> of <page␣
↪property='total' /></h1>
    <h1 id='Third' class='break'>This is the content on page <page /> of <page␣
↪property='total' /></h1>
    <h1 id='Fourth' class='break'>This is the content on page <page /> of <page␣
↪property='total' /></h1>
    <footer>
        <p class='print-only foot'> The Current Page is <page /> of <page property=
↪'total' /></p>
```

(continues on next page)


PageNumbersCurren...
1 page
This is the header
This is the content on page 1
The Current Page is 1

(continued from previous page)

```
    </footer>

</body>

</html>
```

```
public void TotalPageNumbers()
{
    var path = GetTemplatePath("PageNumbers", "PageNumberTotal.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("PageNumbers", "PageNumberTotal.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

10.29.4 The page *for* another component

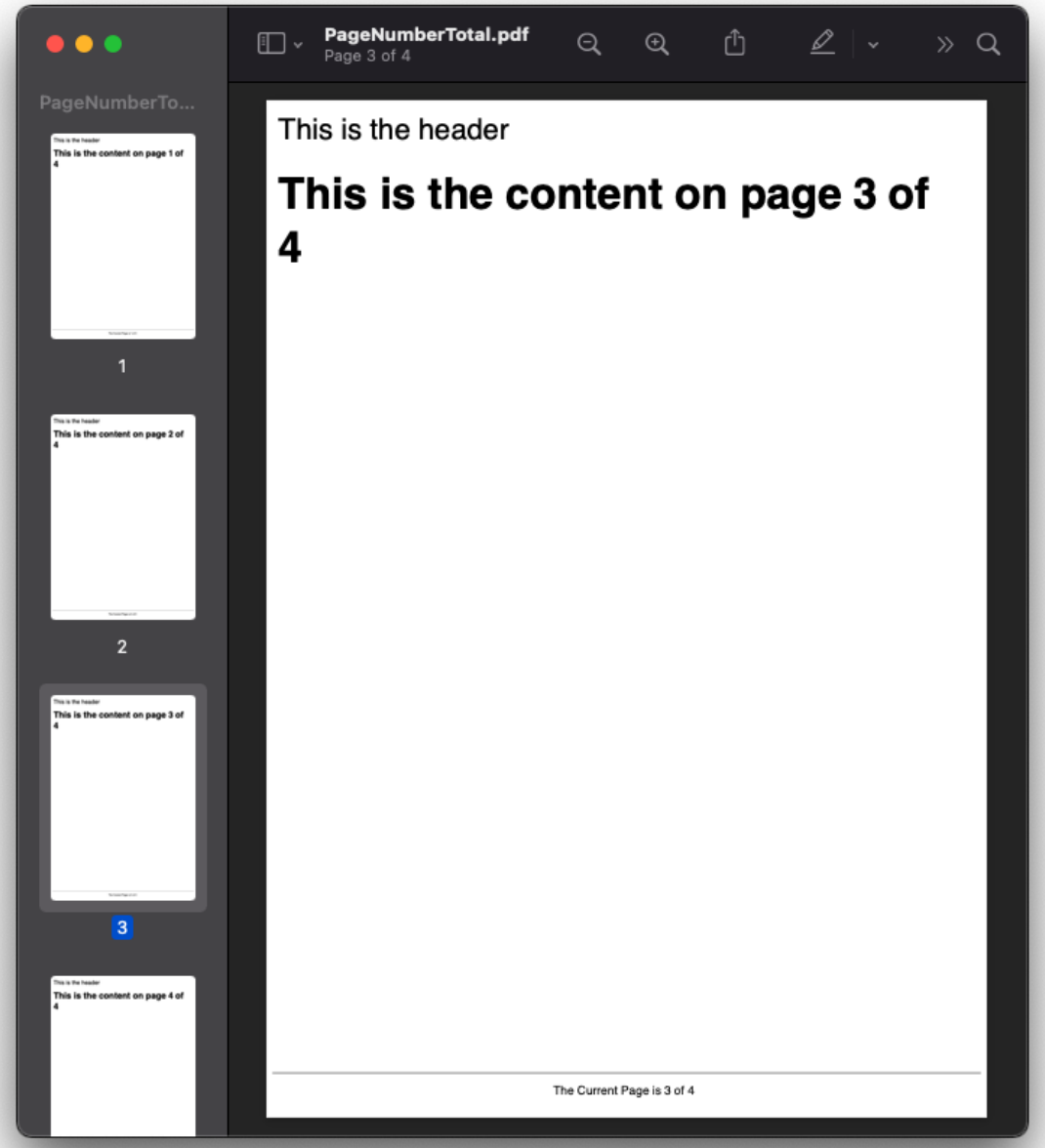
Conversly to the current page number, it is also possible to get the page number of another element. By using the `for` attribute.

The example below is a table of contents with links to sections based on their ID and a line leading to the page numbers on the right cell.

Note: The for referenced component can be following the current content, and not yet laid out. It is only once everything is laid out would the page numbers for another component be evaluated.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>My Document</title>
    <style type="text/css">

    p, h1 {
        padding: 10pt;
    }

    .print-only{
        display:none;
    }

    @media print{

        .print-only{ display: block; }

        .foot{
        border-top: solid 1pt gray;
```

(continues on next page)


PageNumberTotal.pdf
Page 3 of 4
PageNumberTo...
This is the header
This is the content on page 3 of 4
The Current Page is 3 of 4

(continued from previous page)

```
        text-align:center;
        font: 10pt sans-serif;
        margin: 5pt;
        }

        .break{
        page-break-before:always;
        }

        /* Table of Contents Styling */

        table.toc{
        font-size:12pt;
        margin-left:30pt;
        }

        table.toc thead{
        font-weight:bold;
        text-decoration:underline;
        }

        /*  Remove the underline from a hyperlink */

        table.toc a{
        text-decoration:none;
        }

        /*  a horizontal rule, inline dashed with a
            margin to push down to the baseline */

        table.toc hr{
        display:inline;
        margin-top:12pt;
        stroke: gray;
        stroke-dasharray: 2;
        }

        /* remove the default borders from the cells */

        table.toc td{
        border:none;
        }

        /* Explicit width on the last cell */

        table.toc td.pg-num {
        width:30pt;
        }
    }

    </style>
</head>
<body>
    <header>
        <p>This is the header</p>
    </header>
    <h1 id='First'>Table of Contents</h1>
```

(continues on next page)

(continued from previous page)

```
    <table class="toc" style="margin:20pt; width:100%;">
        <tr>
            <td><a href="#Second">Section 1</a><hr class="tab-spacer" /></td>
            <td class="pg-num"><page for="#Second" /></td>
        </tr>
        <tr>
            <td><a href="#Third">Section 2</a><hr class="tab-spacer" /></td>
            <td class="pg-num"><page for="#Third" /></td>
        </tr>
        <tr>
            <td><a href="#Fourth">Section 3</a><hr class="tab-spacer" /></td>
            <td class="pg-num"><page for="#Fourth" /></td>
        </tr>
    </table>
    <h1 id='Second' class='break'>This is the content on page <page /> of <page␣
↪property='total' /></h1>
    <h1 id='Third' class='break'>This is the content on page <page /> of <page␣
↪property='total' /></h1>
    <h1 id='Fourth' class='break'>This is the content on page <page /> of <page␣
↪property='total' /></h1>
    <footer>
        <p class='print-only foot'> The Current Page is <page /> of <page property=
↪'total' /></p>
    </footer>

</body>

</html>
```

```
public void ForComponentPageNumbers()
{
    var path = GetTemplatePath("PageNumbers", "PageNumbersFor.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("PageNumbers", "PageNumbersFor.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

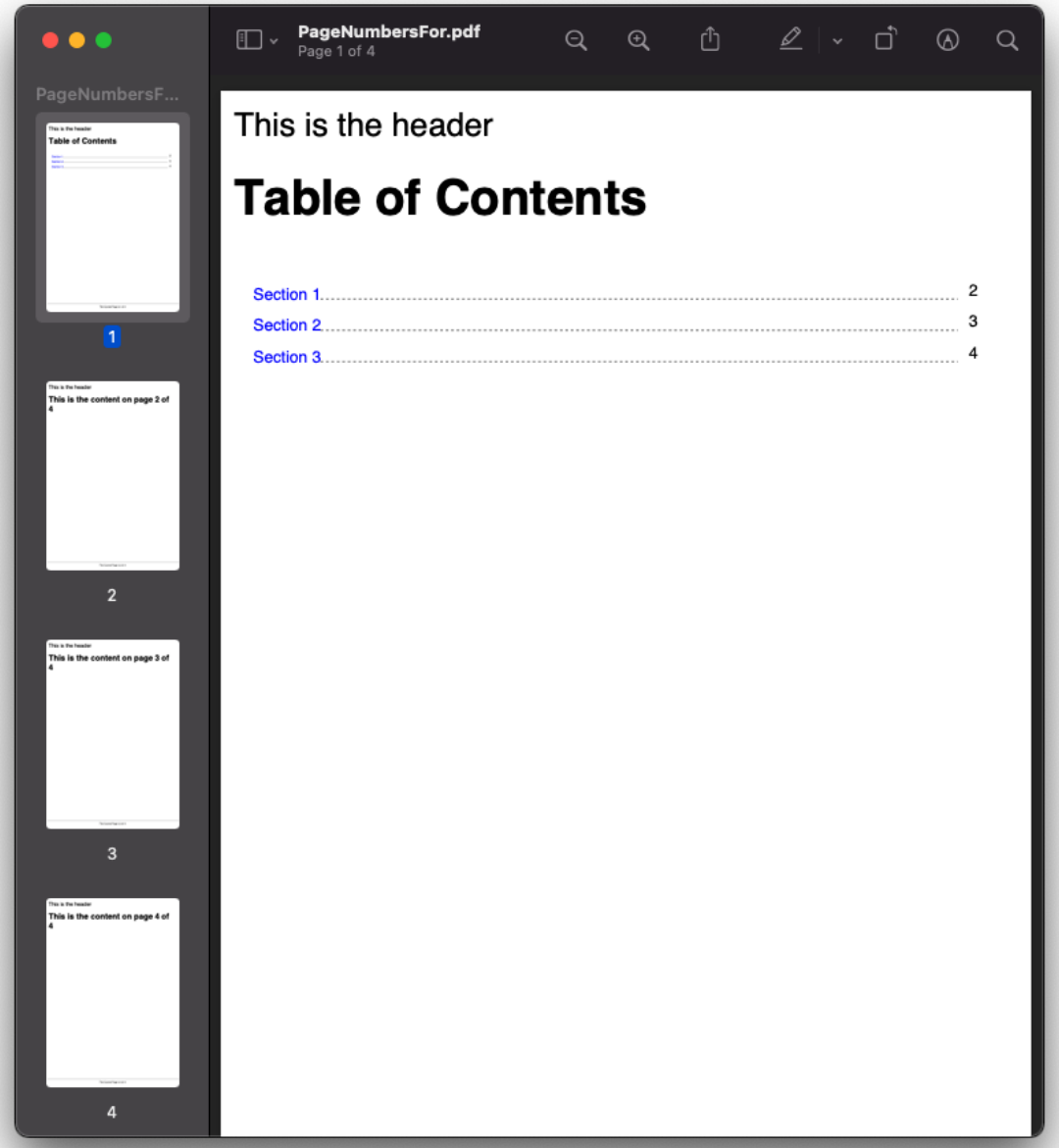
Is is also possible to use also databinding to achieve this (see links_reference in the next section for an example of this).

Full size version

Note: The page index of a component can be forward as in this case, as well as backward looking, but will always be the very first page the component is laid out at, even if it overflows onto another page.

10.29.5 Page Numbers in code

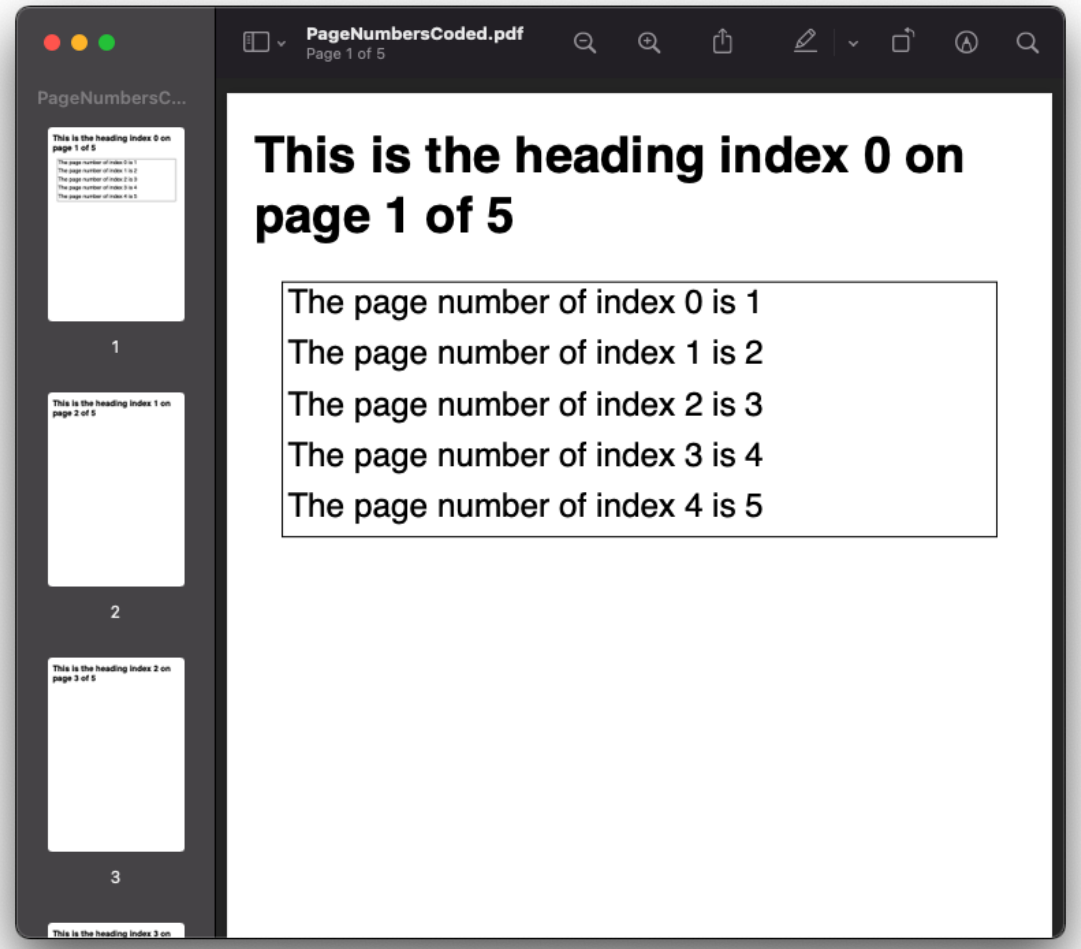
The use of the `PageNumberLabel` and `PageOfLabel` in coded documents is just the same as in templates.


PageNumbersFor.pdf
Page 1 of 4
PageNumbersF...
This is the header
Table of Contents
Section 1 2
Section 2 3
Section 3 4
1
2
3
4

Creating a five page document with headings on each and a reference to each of the the headings on the first page. The spans are added as individual blocks, showing the page numbers of following headings.

```
public void CodedPageNumbers()
{


    using (var doc = new Document())
    {
        for(var i = 0; i < 5; i++)
        {
            var pg = new Page();
            var head = new Head1() { ID = "Item" + i };
            var lit = new TextLiteral() { Text = "This is the heading index " + i + "␣
↪on page " };
            var num = new PageNumberLabel() { DisplayFormat = "{0} of {1}" };
            pg.Style.Margins.All = 20;

            doc.Pages.Add(pg);
            pg.Contents.Add(head);
            head.Contents.Add(lit);
            head.Contents.Add(num);

            if(i == 0) //First page add links to components on the nex
            {
                var div = new Div();
                div.Style.Margins.All = 20;
                div.Style.Border.Color = PDFColors.Black;
                pg.Contents.Add(div);

                for (int j = 0; j < 5; j++)
                {
                    var span = new Span() { PositionMode = PositionMode.Block,␣
↪Padding = new PDFThickness(4) };
                    span.Contents.Add(new TextLiteral("The page number of index " + j␣
↪+ " is "));
                    span.Contents.Add(new PageOfLabel() { ComponentName = "#Item" + j␣
↪});
                    div.Contents.Add(span);
                }
            }

        }


        using (var stream = GetOutputStream("PageNumbers", "PageNumbersCoded.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version


PageNumbersCoded.pdf
Page 1 of 5
PageNumbersC...
This is the heading index 0 on page 1 of 5
The page number of index 0 is 1
The page number of index 1 is 2
The page number of index 2 is 3
The page number of index 3 is 4
The page number of index 4 is 5
1
2
3

10.29.6 Page number spacing

Because the page numbers are calculated at the end of the layout, the spacing needed for the total number of pages (or the page number of a following component) is deferred to the end of the layout. Before then a proxy value is used.

By default this is '99', so enough space will be left for the number '99' to be rendered in the content. For smaller numbers, very long documents, or very large font sizes this may alter the layout too much and potentially casue character clashes.

The `<page />` element supports the *data-page-hint* attribute.

And the `PageNumberLabel` and `PageOfLabel` support the *TotalPageCountHint* properties that can be set to an integer value where clashes need to be fixed.

```
<page property='total' data-page-hint='9999' />
```

```
var pglbl = new PageOfLabel() { ComponentName = "#VeryLastComponent",␣
↪TotalPageCountHint = 9999 };
```

10.30 Panels, Divs and Spans - TD

Preamble

10.30.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

See the tables_reference for more details on what is supported in tables.

10.30.2 Binding Simple numbers

10.30.3 Binding Dates and Times

10.30.4 Calculating values

10.30.5 Building in code

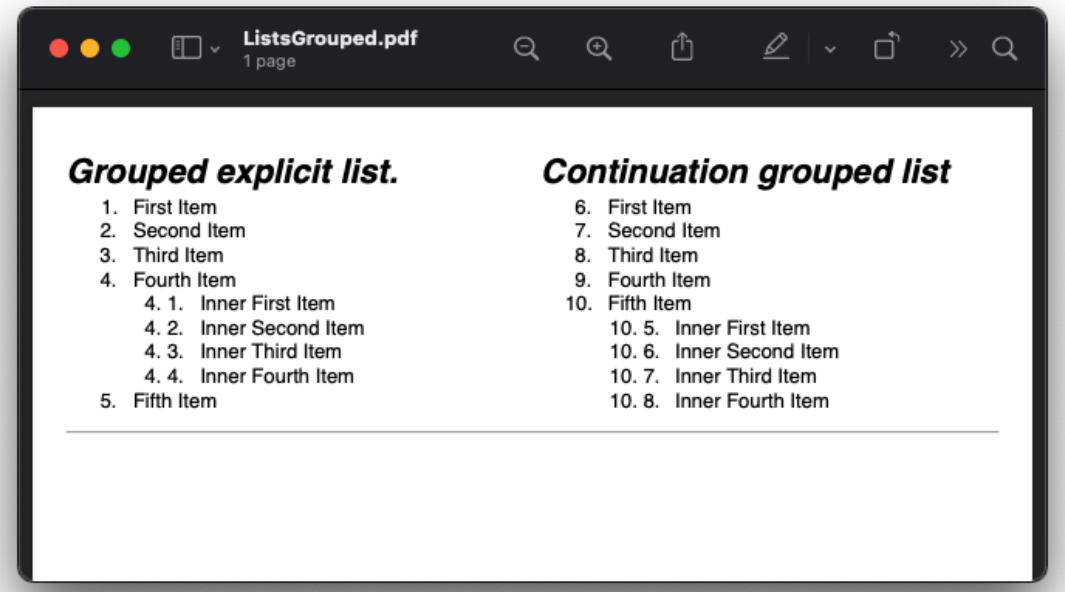
10.31 Headings 1 to 6 and numbers - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.32 Unordered, Ordered and Definition Lists

Scryber supports the use of lists both ordered and unordered and allows nesting, overflow, and definition lists. It also supports the use of binding and repeating on list items.

A list item is just a container for other content, and can contain any children.




```
<!-- xmlns='http://www.w3.org/1999/xhtml' -->

<ol style='list-style-type: lower-roman'>
    <li>First Item</li>
    <li>Second Item</li>
    <li>Third Item</li>
</ol>

<ul>
    <li>First Item</li>
    <li>Second Item</li>
    <li>Third Item</li>
</ul>
```

The `ol` and `ul` lists inherit from `ListOrdered` and `ListUnordered`

```
//using Scryber.Components

var list = new ListOrdered() { NumberingStyle = ListNumberingGroupStyle.
↪LowercaseRoman };
for(var i = 0; i < 3; i++)
{
    var li = new ListItem();
    li.Contents.Add(new TextLiteral("Item #" + i));
    list.Items.Add(li);
}
```

10.32.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

10.32.2 Unordered and ordered lists

Scryber uses the same tags as Html for the Ordered Lists `ol` and Unordered Lists `ul` as per html, this is also possible to alter the list style type using the `list-style` option.

The contents of a list item `li` can be any form of content (inline or otherwise).

```
<!-- /Templates/Lists/ListsSimple.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
        }
    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <div class="separator">
        <h4>An unordered list.</h4>
        <ul style="break-after: always;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>Fifth Item</li>
        </ul>

        <h4>An ordered list</h4>
        <ol>
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>Fifth Item</li>
        </ol>
    </div>

    <div class="separator">
        <h4>A list with lower alpha.</h4>
        <ul style="break-after: always; list-style:lower-alpha;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>Fifth Item</li>
        </ul>
```

(continues on next page)

(continued from previous page)

```
        <h4>A list with upper romam</h4>
        <ol style="list-style: upper-roman;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>Fifth Item</li>
        </ol>
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/ListSamples.cs

public void SimpleList()
{
    var path = GetTemplatePath("Lists", "ListsSimple.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Lists", "ListsSimple.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

10.32.3 Supported list numbering types

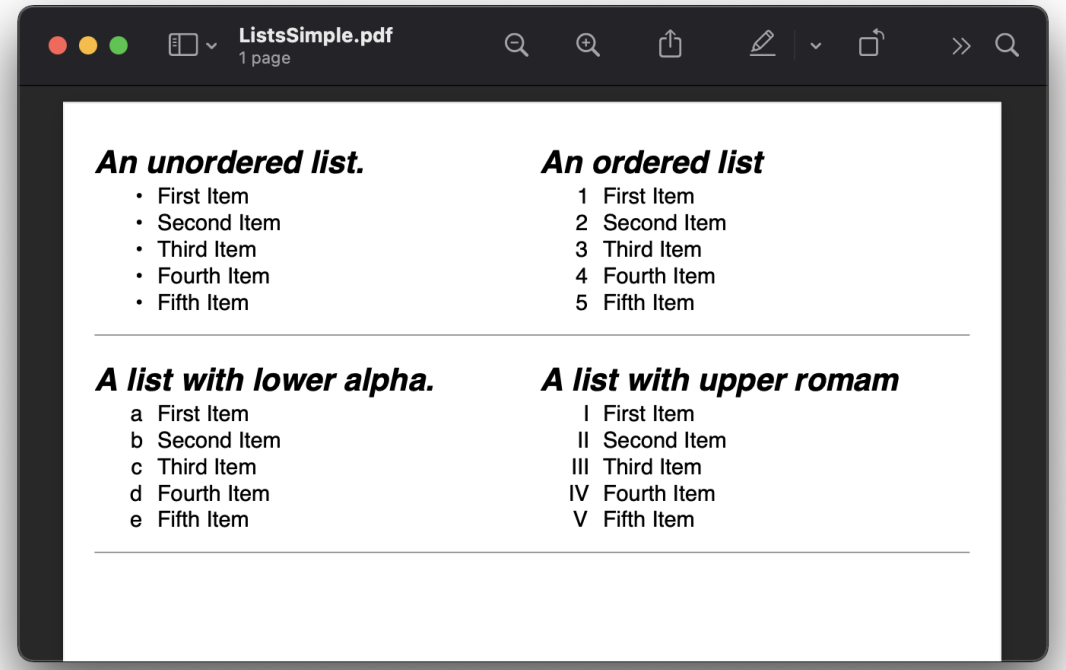
The following types of numbering are supported for lists. This is based on the numbering supported in the PDF Specification.

- disc or circle - this will be a bulleted list.
- decimal - this will be a number 1,2,3,4, etc.
- upper-roman - Roman numerals I, II, III, IV, etc.
- lower-roman - Roman numerals i, ii, iii, iv, etc.
- upper-alpha - Letters A, B, C, D, etc.
- lower-alpha - Letters a, b, c, d, etc.
- none - No list numbering will be shown.

Any other values will be output as decimals.

10.32.4 Overflowing list items

The content of list items will flow niely onto new columns and pages, and are not designed to be split.



However, if this is not the desired effect then they can be moved as a single block onto a new column or page with the `break-inside: avoid` selector, as with other block components.

This can be put any of the items individually, or as in the case below - as a css selector.

```
<!-- /Templates/Lists/ListsOverflow.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
            height: 200pt;
        }

         .keep-together > li{
            break-inside: avoid;
        }
    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <div class="separator">
```

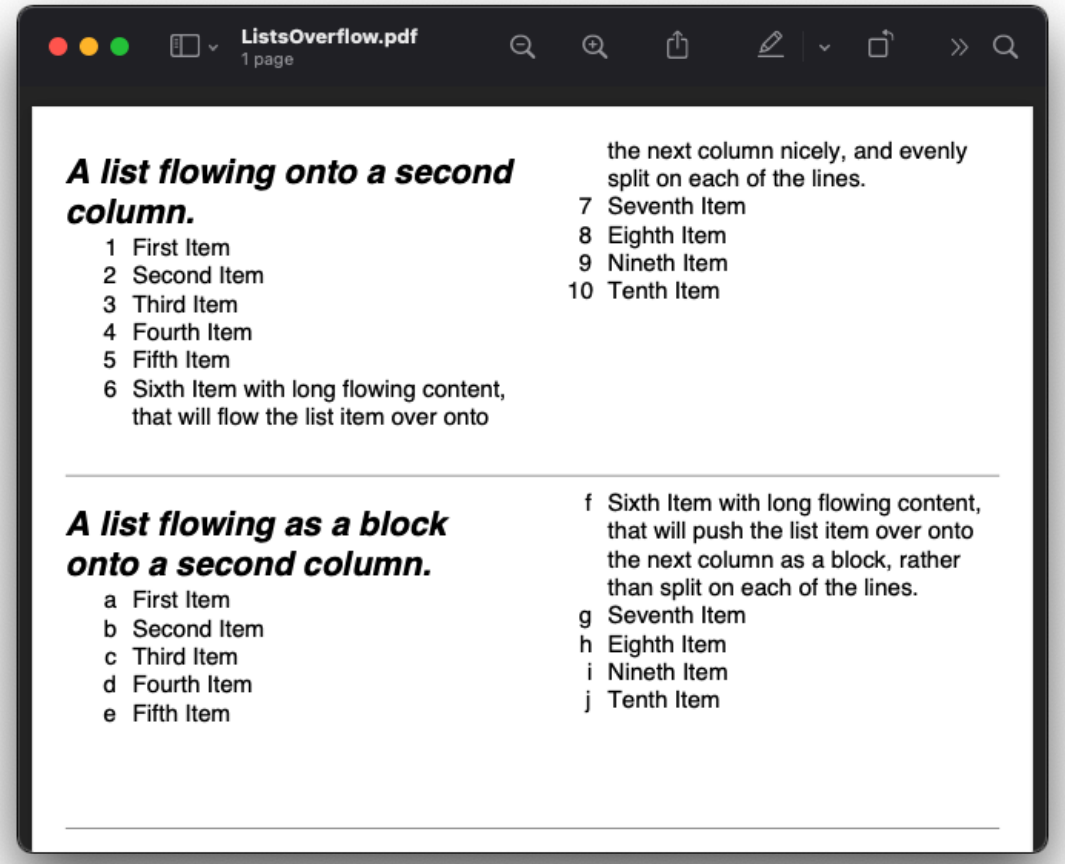
(continues on next page)

(continued from previous page)

```
        <h4>A list flowing onto a second column.</h4>
        <ol>
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>Fifth Item</li>
            <li>Sixth Item with long flowing content, that will flow the
                list item over onto the next column nicely, and evenly
                split on each of the lines.</li>
            <li>Seventh Item</li>
            <li>Eighth Item</li>
            <li>Nineth Item</li>
            <li>Tenth Item</li>
        </ol>
    </div>

    <div class="separator">
        <h4>A list flowing as a block onto a second column.</h4>
        <ul class="keep-together" style="list-style:lower-alpha;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>Fifth Item</li>
            <li>Sixth Item with long flowing content, that will push the
            list item over onto the next column as a block, rather than
            split on each of the lines.</li>
            <li>Seventh Item</li>
            <li>Eighth Item</li>
            <li>Nineth Item</li>
            <li>Tenth Item</li>
        </ul>
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/ListSamples.cs

public void OverflowingList()
{
    var path = GetTemplatePath("Lists", "ListsOverflow.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Lists", "ListsOverflow.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version


ListsOverflow.pdf
1 page
A list flowing onto a second column.
1 First Item
2 Second Item
3 Third Item
4 Fourth Item
5 Fifth Item
6 Sixth Item with long flowing content, that will flow the list item over onto the next column nicely, and evenly split on each of the lines.
7 Seventh Item
8 Eighth Item
9 Nineth Item
10 Tenth Item
A list flowing as a block onto a second column.
a First Item
b Second Item
c Third Item
d Fourth Item
e Fifth Item
f Sixth Item with long flowing content, that will push the list item over onto the next column as a block, rather than split on each of the lines.
g Seventh Item
h Eighth Item
i Nineth Item
j Tenth Item

10.32.5 Definition Lists

Definition lists allow terms and contents to be set out with a term and a definition. Whilst not expressly a list, they are covered here as part of our list building.

The <dl></dt> top level tag defines the list, and the inner <dt></dt> terms and <dd></dd> definitions supporting any inner content. The definitions are margins inset by 100pt's to the left.

As the definitions are simply blocks, they support all style and class properties of block_styles, and will split across columns and pages. Unlike list items, the terms and the definitions are separate blocks, so the term is be independent of the definition.

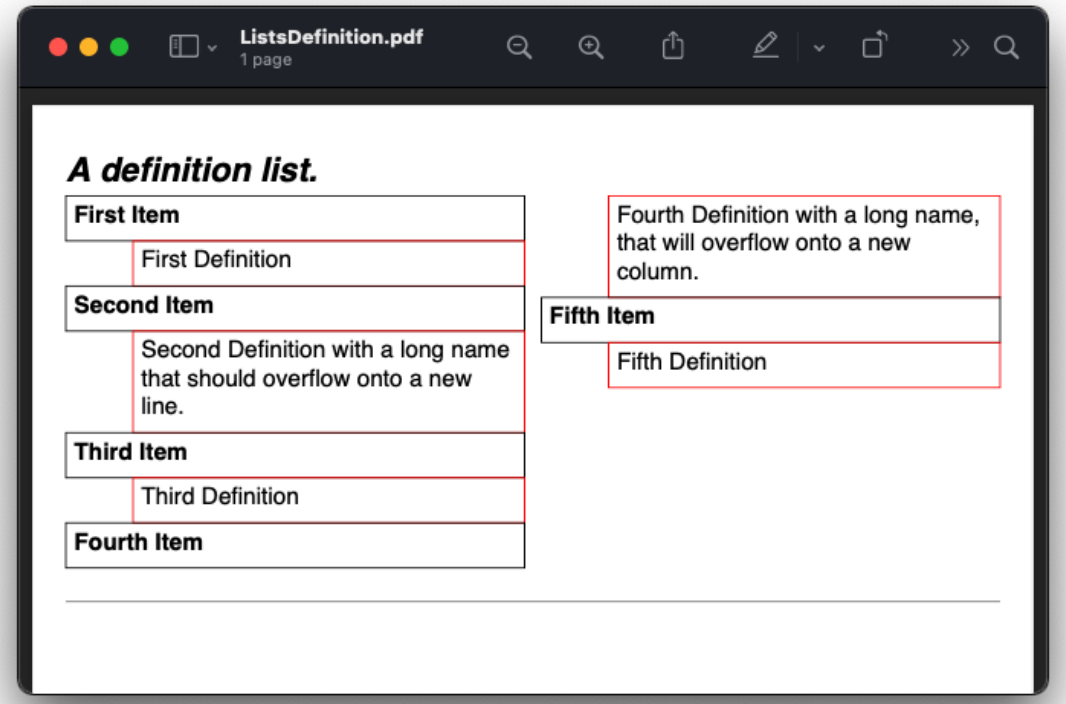
```
<!-- /Templates/Lists/ListsDefinition.html -->

<!DOCTYPE html>
<?scryber append-log=false parser-log=true ?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Definition Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            max-height: 250pt;
            column-count: 2;
        }

        dt{ border: solid 1px black; padding:5pt; break-inside: avoid; }
        dd{ border: solid 1px red; padding:5pt;  break-inside: avoid; }

    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <h4>A definition list.</h4>
    <div id="sep1" class="separator">
        <dl id="dl1">
            <dt>First Item</dt>
            <dd id="def1">First Definition</dd>
            <dt id="term1">Second Item</dt>
            <dd id="def2">Second Definition with a long name that should overflow␣
↪onto a new line.</dd>
            <dt>Third Item</dt>
            <dd>Third Definition</dd>
            <dt id="dt4">Fourth Item</dt>
            <dd id="dd4">Fourth Definition with a long name, that will overflow onto␣
↪a new column.</dd>
            <dt>Fifth Item</dt>
            <dd>Fifth Definition</dd>
        </dl>
    </div>
</body>
</html>
```

```
//cryber.UnitSamples/ListSamples.cs

public void DefinitionList()
```

(continues on next page)

(continued from previous page)

```
{
    var path = GetTemplatePath("Lists", "ListsDefinition.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Lists", "ListsDefinition.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

10.32.6 Nested Lists

Scryber supports the nesting of lists within each other. The number type and style can change with inner lists. As above the overlow of list items can be avoided as needed.

```
<!-- /Templates/Lists/ListsNested.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
```

(continues on next page)

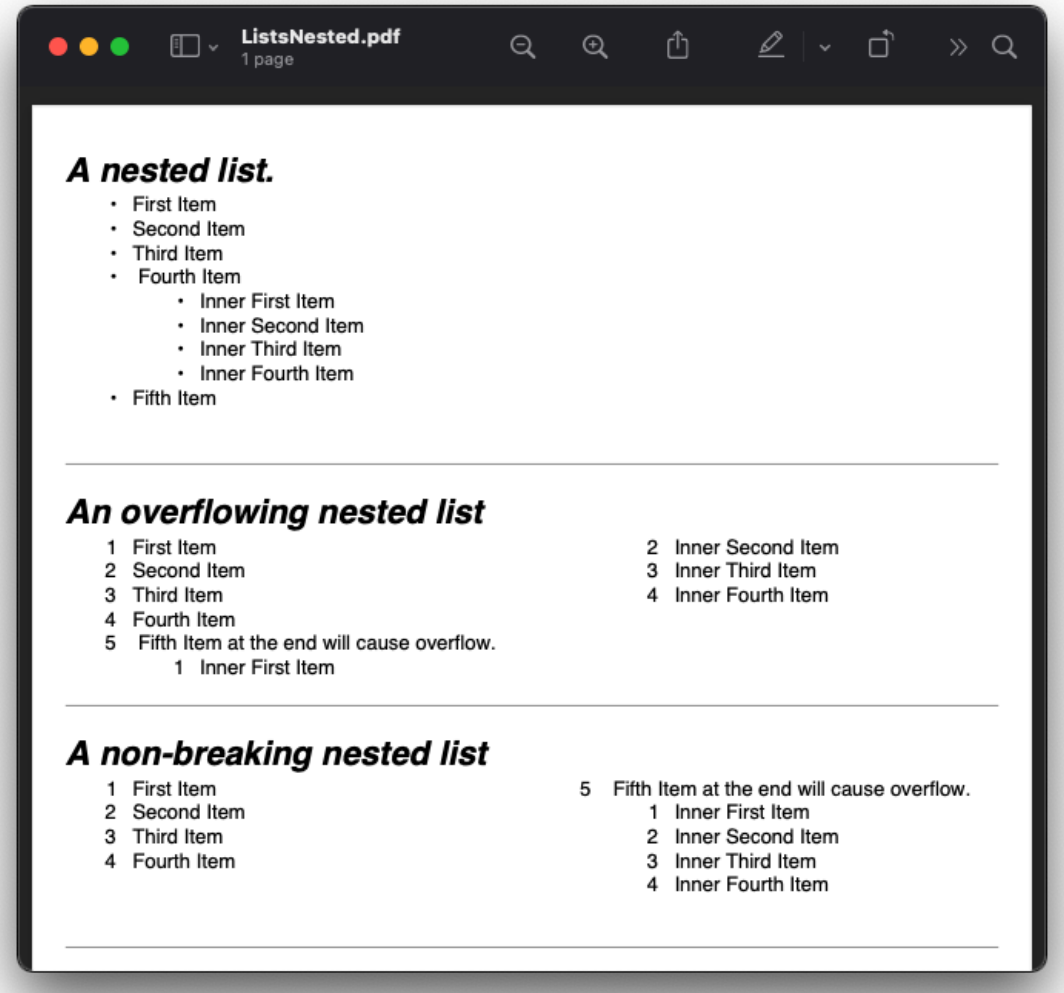
(continued from previous page)

```
<head>
    <meta charset="utf-8" />
    <title>Nested Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            font-size: 12pt;
            column-count:2;
            height: 100pt;
        }


    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <h4>A nested list.</h4>
    <div class="separator" style="height: 160pt;" >
        <ul>
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>
                Fourth Item
                <ul>
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ul>
            </li>
            <li>Fifth Item</li>
        </ul>
    </div>
    <h4>An overflowing nested list</h4>
    <div class="separator">

        <ol class="top">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>
                Fifth Item at the end will cause overflow.
                <ol class="inner">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
        </ol>
    </div>
    <h4>A non-breaking nested list</h4>
    <div class="separator">
        <ol class="top">
```

(continues on next page)

(continued from previous page)

```
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li style="break-inside:avoid;">
                Fifth Item at the end will cause overflow.
                <ol class="inner">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
        </ol>
    </div>
</body>
</html>
```



Full version

10.32.7 Prefix and postfix

Lists support a pre-fix, and a post-fix string that can be applied to the numbering. This will add a string value to either before the list number and/or after the list number.

As this is a *non-standard* html capability, the values can be specified in 2 ways:

1. As an attribute on the list itself with the `data-li-prefix` and `data-li-posfix` values.
2. As a custom css property with the `-pdf-li-prefix` and `-pdf-li-postfix` values either on the tag style, or on the CSS *styles*.

If it is set in the css, then the value can be wrapped in single or double quotes, so preceding or trailing spaces are not removed. This can also be combined with nesting, concatenation and grouping, as in the examples below.

```
<!-- /Templates/Lists/ListsPrePostFix.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
            font-size: 12pt;
            height: 180pt;
        }

        /* Style properties without quotes,
           the value will be truncated     */
        .top{
            -pdf-li-postfix: .;
        }

        /* Style properties with quotes,
           can have spaces in.          */
        .inner{
            -pdf-li-prefix: '# ';
            -pdf-li-postfix: ".";
        }

    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <div class="separator">
        <h4>Pre/Post explicit list.</h4>
        <ol data-li-postfix="." style="break-after: always;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>
                Fourth Item
```

(continues on next page)

(continued from previous page)

```
                <ol data-li-prefix="# " data-li-postfix=".">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
            <li>Fifth Item</li>
        </ol>

        <h4>Pre/Post styled list</h4>
        <ol class="top">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>
                Fifth Item
                <ol class="inner">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
        </ol>
    </div>
</body>
</html>
```

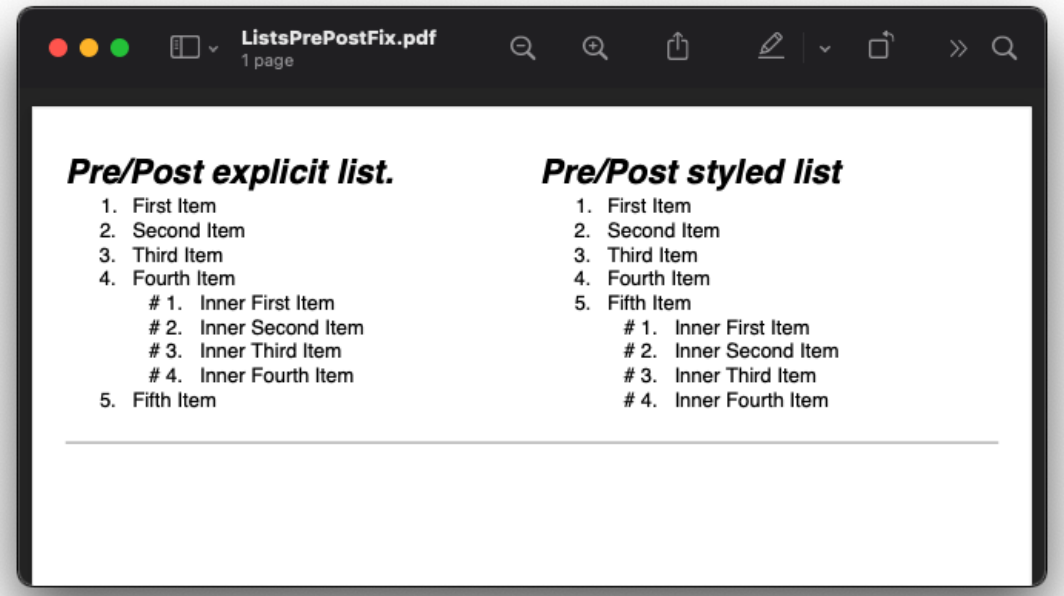
```
//Scryber.UnitSamples/ListSamples.cs

public void PrePostFixList()
{
    var path = GetTemplatePath("Lists", "ListsPrePostFix.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Lists", "ListsPrePostFix.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full version

10.32.8 Concatenated List numbers

The `-pdf-li-concat` css property or `data-li-concat` attribute control if nested list numbers are concatenated with their parents. The concatenation value can be true, 1, or *concatenate* in the css property. **Any** other value will be treated as false.

For the data attribute, the concatenation value can only be *true* or *false* (as it is directly on the boolean class property - see ../overview/scryber_parsing).




```
<!-- /Templates/Lists/ListsNestedConcatenated.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
            font-size: 12pt;
        }


        .top {
            -pdf-li-postfix: .;
        }

        .inner {
            /*using the 'concatenate' option*/
            -pdf-li-concat: concatenate;
            list-style-type: lower-alpha;
            -pdf-li-prefix: ' ';
            -pdf-li-postfix: ".";
        }
    </style>
</head>
```

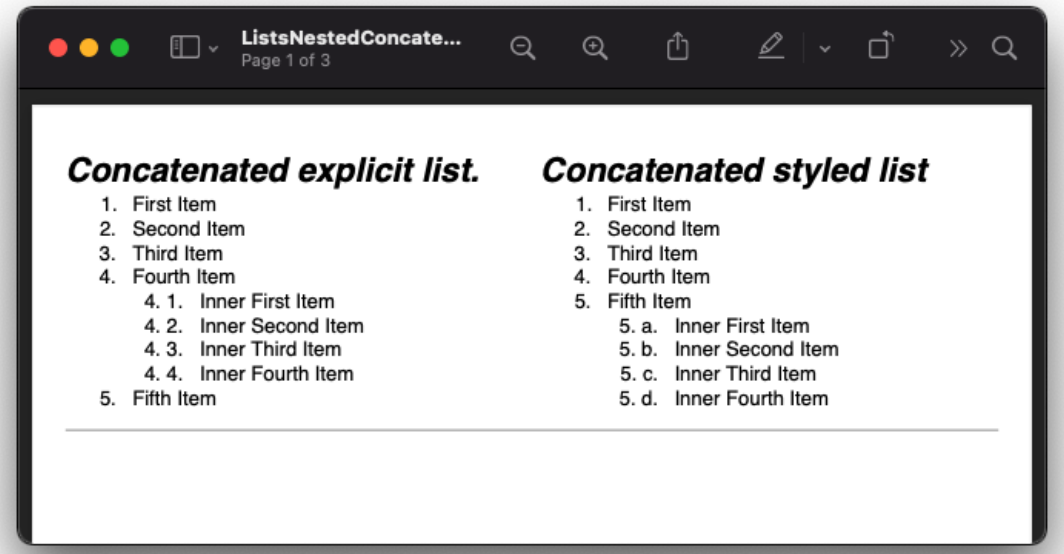
(continues on next page)

(continued from previous page)

```
<body style="padding:20pt; font-size: 14pt;">
    <div class="separator">
        <h4>Concatenated explicit list.</h4>
        <ol data-li-postfix="." style="break-after: always;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>
                Fourth Item
                <ol data-li-prefix=" " data-li-concat="true" data-li-postfix=".">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
            <li>Fifth Item</li>
        </ol>

        <h4>Concatenated styled list</h4>
        <ol class="top">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>
                Fifth Item
                <ol class="inner">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
        </ol>
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/ListSamples.cs

public void ConcatenatedList()
{
    var path = GetTemplatePath("Lists", "ListsNestedConcatenated.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Lists", "ListsNestedConcatenated.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version



10.32.9 List grouping

Number groups can be used so the values increment outside of the list using the `-pdf-li-group` css property, or if preferred, the `data-li-group` attribute on the list tag itself. Group names can be any valid string, but *are* case sensitive

A group will maintain the index number across the whole document, and each list item will increment the number.

When grouped the style type can still be updated, without affecting the numbering.

```
<!-- /Templates/Lists/ListsGrouped.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
            font-size: 12pt;
        }

        .top {
            -pdf-li-postfix: .;
            /*group name is 'one'*/
            -pdf-li-group: one;
        }
```

(continues on next page)

(continued from previous page)

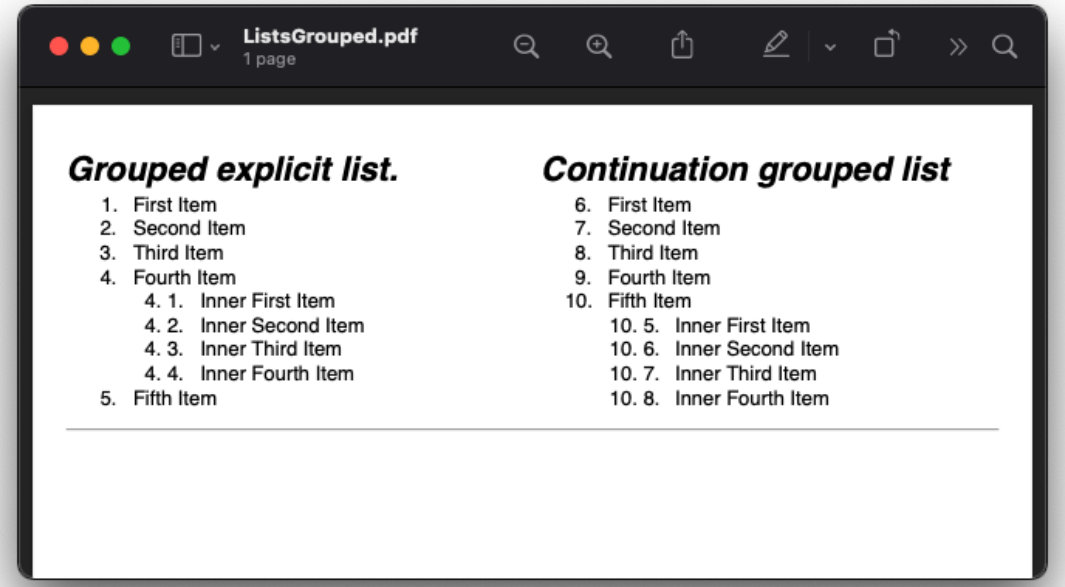
```
        .inner {
            -pdf-li-concat: concatenate;
            /*group name is 'two'*/
            -pdf-li-group: two;
            list-style-type: lower-alpha;
            -pdf-li-prefix: ' ';
            -pdf-li-postfix: ".";
        }
    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <div class="separator">
        <h4>Grouped explicit list.</h4>
        <!-- start group 'one' -->
        <ol data-li-group="one" data-li-postfix="." style="break-after: always;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>
                Fourth Item
                <!-- Start group 'two' -->
                <ol data-li-group="two" data-li-prefix=" " data-li-concat="true" data-
↪li-postfix=".">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
            <li>Fifth Item</li>
        </ol>

        <h4>Continuation grouped list</h4>
        <!-- continue group 'one' -->
        <ol class="top">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>
                Fifth Item
                <!-- continues group 'two' -->
                <ol class="inner">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item</li>
                </ol>
            </li>
        </ol>
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/ListSamples.cs

```

(continues on next page)

(continued from previous page)

```
public void NumberGroupedList()
{
    var path = GetTemplatePath("Lists", "ListsGrouped.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Lists", "ListsGrouped.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

Note: This is now similar to the css counter-reset and counter-increment options. It may be implemented in the future to allow numbering on any tag, but both counted and -pdf-li- options can be used together as needed.

10.32.10 Number alignment and inset.

The list items number block is right align by default with a width of 30pts with an alley of 10pt (between the number and the content). If lists numbers are concatenated, are deeply nested, or have long pre-fixes etc. then this may cause the numbers to flow onto multiple lines.

As such the `data-li-inset` (or `-pdf-li-inset`) will take a unit value as the effective width of the number, and then at 10pt for the start of the item content block.

If needed, then the number can be aligned with `data-li-align` (or `-pdf-li-align`) to alter the alignment of the number text to *left* or even *center*.

The inset is available for list items as well, to affect the layout and ensure items can fit.

```
<!-- /Templates/Lists/ListsInsetAndAlign.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
            font-size: 12pt;
        }

        .top {
            -pdf-li-postfix: .;
        }

        .inner {
            list-style-type: lower-alpha;
            -pdf-li-concat: concatenate;
            -pdf-li-prefix: ' ';
            -pdf-li-postfix: ".";
        }


        ol.left{
            /* left aligned */
            -pdf-li-align:left;
        }

        ol.wide {
            /* increase the li number width */
            -pdf-li-inset: 50pt;
            list-style-type:lower-roman;
        }

        li.v-wide{
            /* increase an item explicitly */
            -pdf-li-inset: 80pt;
        }
    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <div class="separator">
        <h4>Wide explicit list.</h4>
        <ol data-li-align="Left"  data-li-postfix="." style="break-after: always;">
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>
```

(continues on next page)

(continued from previous page)

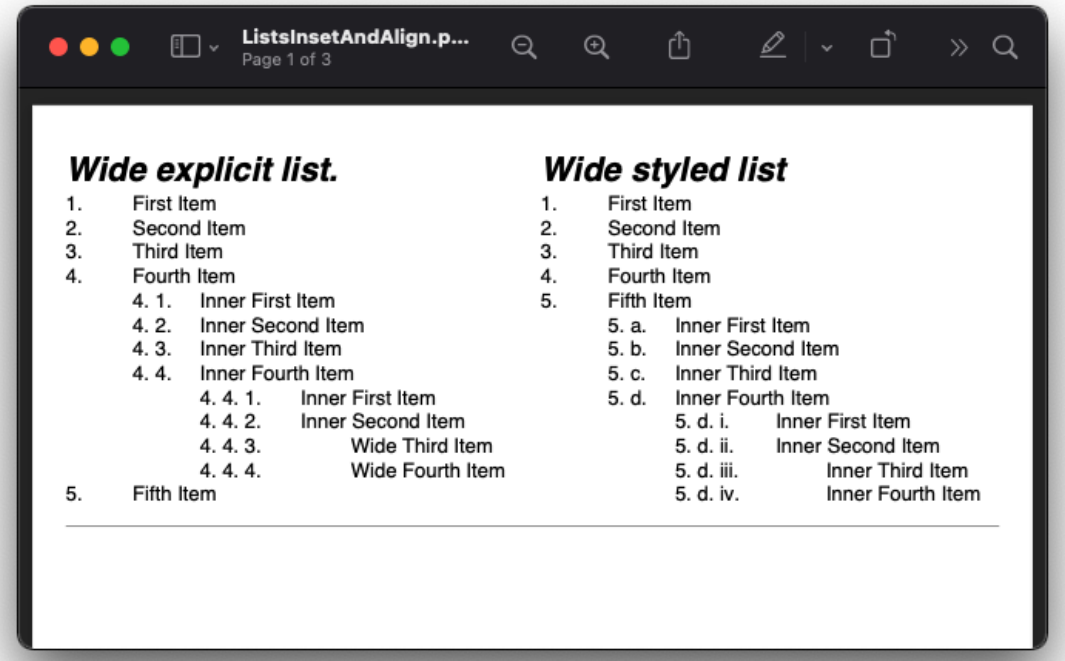
```
                Fourth Item
                <ol data-li-align="Left" data-li-prefix=" "
                    data-li-concat="true" data-li-postfix=".">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item
                        <ol data-li-inset="50pt" data-li-align="Left"
                            data-li-prefix=" " data-li-concat="true"
                            data-li-postfix=".">
                            <li>Inner First Item</li>
                            <li>Inner Second Item</li>
                            <li data-li-inset="80pt">Wide Third Item</li>
                            <li data-li-inset="80pt">Wide Fourth Item</li>
                        </ol>
                    </li>
                </ol>
            </li>
            <li>Fifth Item</li>
        </ol>

        <h4>Wide styled list</h4>
        <ol class="top left" >
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
            <li>Fourth Item</li>
            <li>
                Fifth Item
                <ol class="inner left">
                    <li>Inner First Item</li>
                    <li>Inner Second Item</li>
                    <li>Inner Third Item</li>
                    <li>Inner Fourth Item
                        <ol class="inner left wide">
                            <li>Inner First Item</li>
                            <li>Inner Second Item</li>
                            <li class="v-wide">Inner Third Item</li>
                            <li class="v-wide">Inner Fourth Item</li>
                        </ol>
                    </li>
                </ol>
            </li>
        </ol>
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/ListSamples.cs

public void NumberInsetAndAlignList()
{
    var path = GetTemplatePath("Lists", "ListsInsetAndAlign.html");

    using (var doc = Document.ParseDocument(path))
    {
```

(continues on next page)

(continued from previous page)

```
        using (var stream = GetOutputStream("Lists", "ListsInsetAndAlign.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

10.32.11 Building Lists in code

Lists and list items are just as easy to define in code. The base class in the `Scryber.Components` namespace is `List`, with `ListOrdered`, `ListUnordered` and `ListDefinition` inheriting from the base class and applying their own base style.

The list items `Scryber.Components.ListItem` can be added to the list `Items` collection, and adds some extra style properties for the ItemLabelText (for definition lists), the NumberAlignment and the NumberInset.

The properties for the specific list styles on the `List` class are

- NumberingStyle
- NumberingGroup
- NumberPrefix
- NumberPostfix

- NumberInset
- NumberAlignment

or they can be set on the components `Style` property, or on a `StyleDefn` properties. (see: ../overview/styles_and_classes)

- comp.Style.List.NumberingStyle
- comp.Style.List.NumberingGroup
- comp.Style.List.NumberPrefix
- comp.Style.List.NumberPostfix
- comp.Style.List.NumberInset
- comp.Style.List.NumberAlignment

```
<!-- /Templates/Lists/ListsCoded.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Coded Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
            height: 170px;
        }
    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">

    <h4>Add a list after</h4>
    <div id="TopDiv" class="separator">
    </div>

    <h4>Add another list after</h4>
    <div id="SecondDiv" class="separator">
    </div>
</body>
</html>
```

```
// Scryber.UnitSamples/ListSamples.cs

public void CodedList()
{
    var path = GetTemplatePath("Lists", "ListsCoded.html");

    using (var doc = Document.ParseDocument(path))
    {

        if (doc.TryFindAComponentById("TopDiv", out Div top))
        {
            ListOrdered ol = new ListOrdered() { NumberingStyle =␣
↪ListNumberingGroupStyle.LowercaseLetters };
```

(continues on next page)

(continued from previous page)

```
            for(var i = 1; i < 10; i ++)
            {
                ListItem li = new ListItem();
                li.Contents.Add(new TextLiteral("Item #" + i));

                //Setting the item number alignment to left individually
                if (i == 5)
                    li.NumberAlignment = HorizontalAlignment.Left;

                ol.Items.Add(li);
            }
            top.Contents.Add(ol);
        }

        if (doc.TryFindAComponentById("SecondDiv", out Div second))
        {
            ListDefinition dl = new ListDefinition();

            for (var i = 1; i < 10; i++)
            {
                ListDefinitionTerm term = new ListDefinitionTerm();
                term.Contents.Add(new TextLiteral("Term " + i));
                dl.Items.Add(term);

                ListDefinitionItem def = new ListDefinitionItem();
                def.Contents.Add(new TextLiteral("Definition for term " + i));

                //Setting the item number inset to 100 with margins
                if (i == 5)
                    def.Style.Margins.Left = 100;

                dl.Items.Add(def);

            }
            second.Contents.Add(dl);
        }

        using (var stream = GetOutputStream("Lists", "ListsCoded.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }

    }
}
```

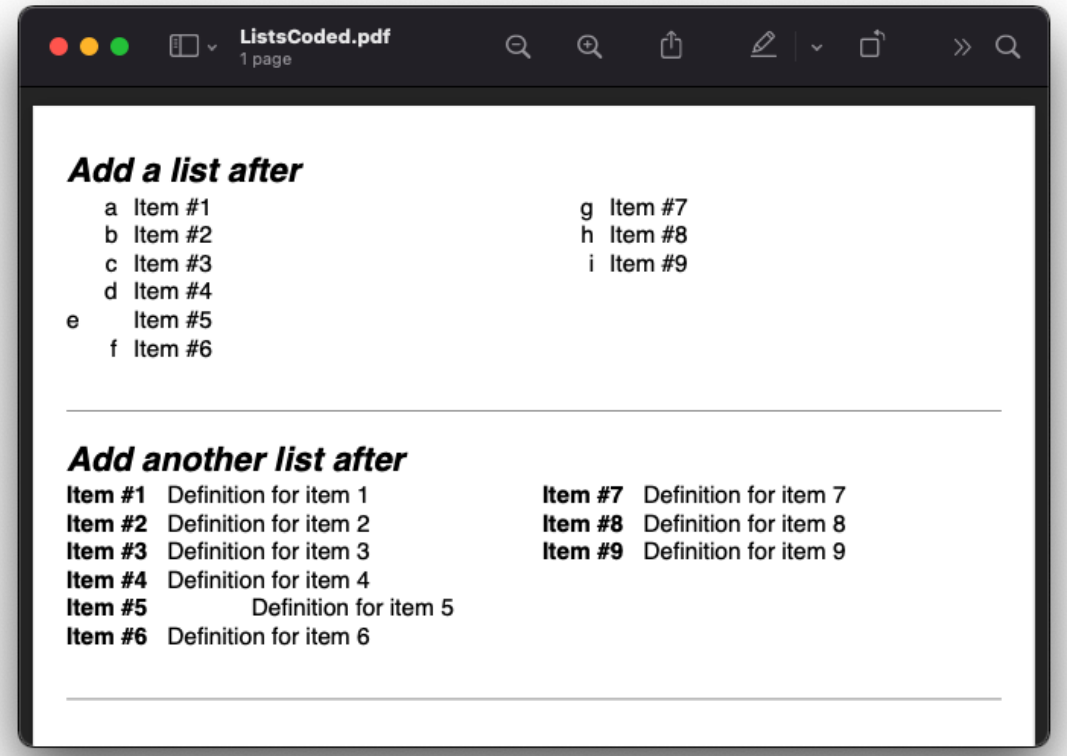
Full size version

It is also just as easy to look up an existing list and add or remove items, or alter contents as needed.

10.32.12 Any list contents

So far all list content has been text (or other list items), however the content for a list item can be any visual content. Tables, paragraphs, images, divs, spans etc are all supported.

Scryber will attempt to lay them out appropriately.


ListsCoded.pdf
1 page
Add a list after
a Item #1
b Item #2
c Item #3
d Item #4
e Item #5
f Item #6
g Item #7
h Item #8
i Item #9
Add another list after
Item #1 Definition for item 1
Item #2 Definition for item 2
Item #3 Definition for item 3
Item #4 Definition for item 4
Item #5 Definition for item 5
Item #6 Definition for item 6
Item #7 Definition for item 7
Item #8 Definition for item 8
Item #9 Definition for item 9

```
<!-- /Templates/Lists/ListsComplexContent.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
            column-count: 2;
        }

        .mixed {
            background-image: radial-gradient(farthest-corner at 40px 40px, #f35 0%,
↪#43e 100%);
            padding:10pt;
        }

        .mixed table {
            background-color: rgba(255,255,255,0.2);
        }

        .mixed > li > p{
            font: bold 20pt;
            margin-top:0;
            padding-bottom: 5pt;
        }

        .mixed .last {
            font-size: 20pt;
            font-style: italic;
        }

    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <div class="separator">
        <h4>A mixed content list.</h4>
        <ol class="mixed">
            <li>First <b>Strong</b> Item </li>
            <li>
                <img src="../../images/landscape.jpg" style="width: 50pt; opacity: 0.5
↪" />
            </li>
            <li>
                <table style="width:100%">
                    <tr><td>One</td><td>Two</td><td>Three</td></tr>
                    <tr><td>Four</td><td>Five</td><td>Six</td></tr>
                </table>
            </li>
            <li style="margin-top:10pt">
                <p>This is a paragraph of content with a specific style</p>
                <p>A following paragraph</p>
            </li>
```

(continues on next page)

(continued from previous page)

```
            <li class="last">Normal list item where the style is applied to the␣
↪content <b>and</b> the number.</li>
        </ol>


    </div>

</body>
</html>
```

```
//Scryber.UnitSamples/ListSamples.cs

public void ComplexListContent()
{
    var path = GetTemplatePath("Lists", "ListsComplexContent.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Lists", "ListsComplexContent.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

10.32.13 Inline-Block None and image Style

A common scenario with html list items is to use them as navigation elements.

We are getting there with our html support for design of content, but at the moment fixed width and floating is probably the best available option for this scenario.

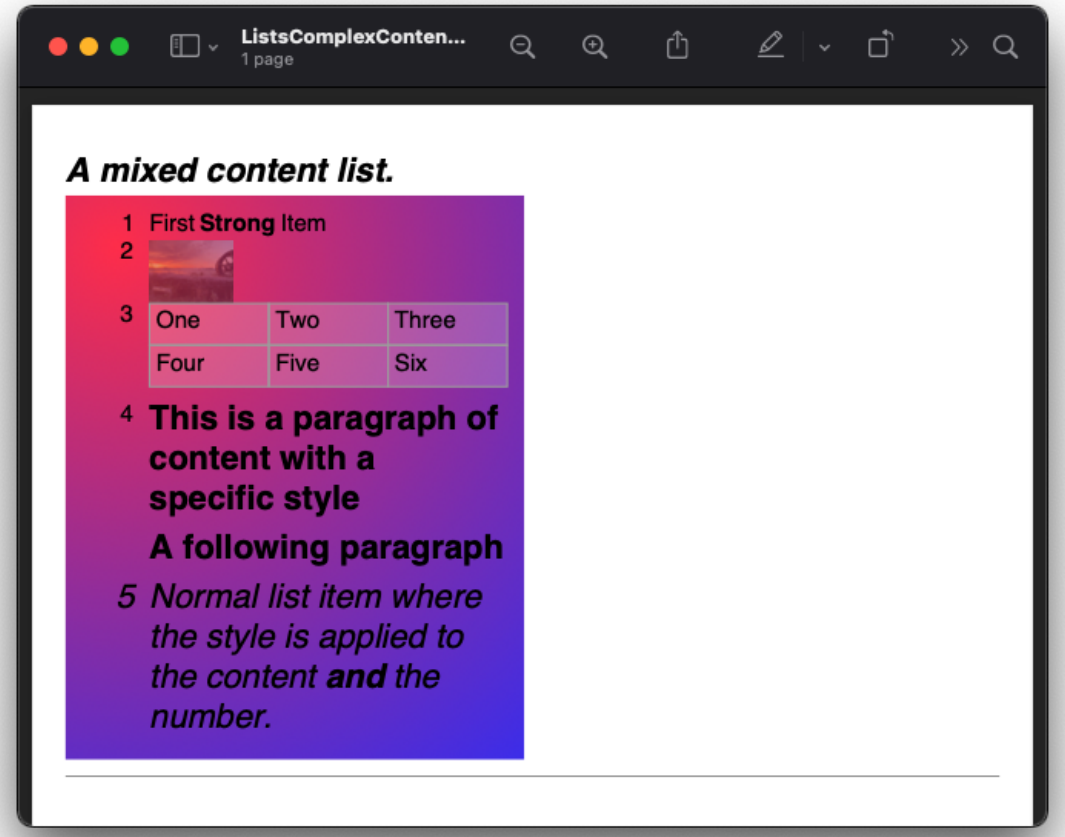
10.32.14 Binding List items

Just as with tables and any other content, lists fully support data binding (at any level), and can take data from either the parameters or the current data, using the `template` component

See ../overview/parameters_and_expressions for more on how to set up sources and get data into a document.

```
<!-- /Templates/Lists/ListsDataBound.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Nested Lists</title>
    <style>
        .separator {
            border-bottom: solid 1px gray;
            margin-bottom: 10pt;
            padding-bottom: 10pt;
```

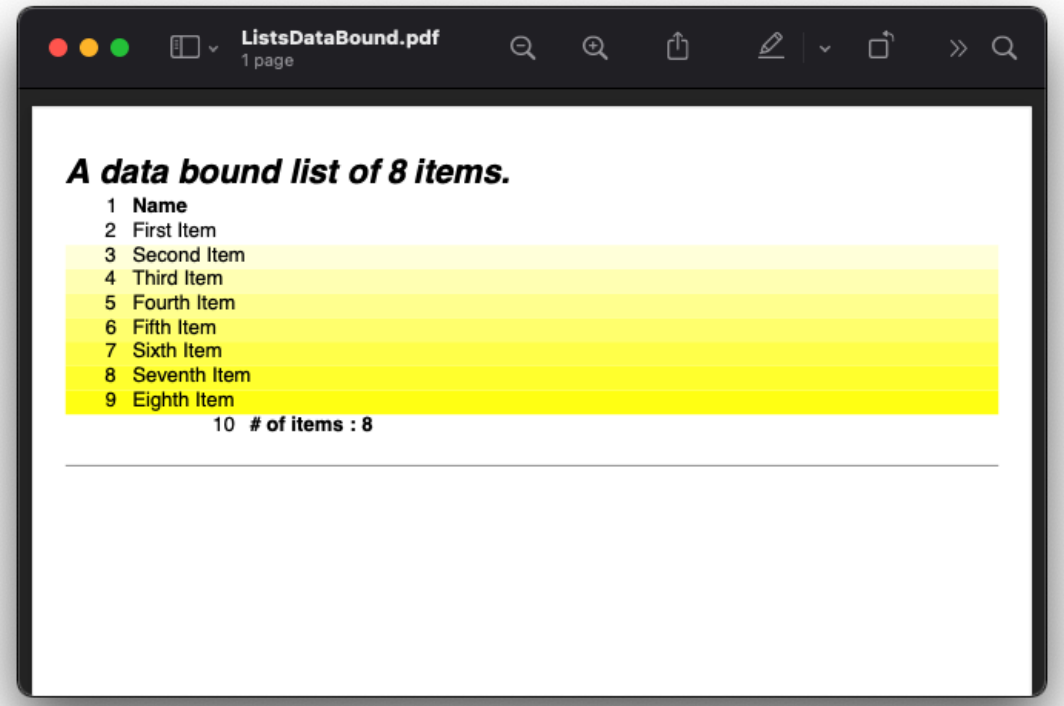
(continues on next page)


ListsComplexConten...
1 page
A mixed content list.
1 First Strong Item
2
3
One
Two
Three
Four
Five
Six
4 This is a paragraph of content with a specific style
A following paragraph
5 Normal list item where the style is applied to the content and the number.

(continued from previous page)

```
            font-size: 12pt;
            height: 100pt;
        }

        li.full { width: 100%}

    </style>
</head>
<body style="padding:20pt; font-size: 14pt;">
    <h4>A data bound list of {{count(model.items)}} items.</h4>
    <div class="separator" style="height: 160pt;" >
        <ol style="width:100%">
            <li><b>Name</b></li>
            <template data-bind="{{model.items}}">
                <li class="full" style="background-color: calc(.color);">
                    <span>{{.name}}</span>
                </li>
            </template>
            <li style="-pdf-li-inset:100pt;"><b>{{concat('# of items is ',
↪count(model.items))}}</b></li>
        </ol>
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/ListSamples.cs

public void BoundListData()
{
    var path = GetTemplatePath("Lists", "ListsDataBound.html");

    var model = new
    {
        items = new []
        {
            new { name = "First Item", color = "#FFF"},
            new { name = "Second Item", color = "#FFD"},
            new { name = "Third Item", color = "#FFB"},
            new { name = "Fourth Item", color = "#FF9" },
            new { name = "Fifth Item", color = "#FF7" },
            new { name = "Sixth Item", color = "#FF5" },
            new { name = "Seventh Item", color = "#FF3"},
            new { name = "Eighth Item", color = "#FF1"}
        }

    };
    using (var doc = Document.ParseDocument(path))
    {
        doc.Params["model"] = model;
        using (var stream = GetOutputStream("Lists", "ListsDataBound.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

10.32.15 Other numbering components

All other container content also supports numbers and numbering, not just lists - see headings_reference for more information.

10.33 Tables, Rows and Cells

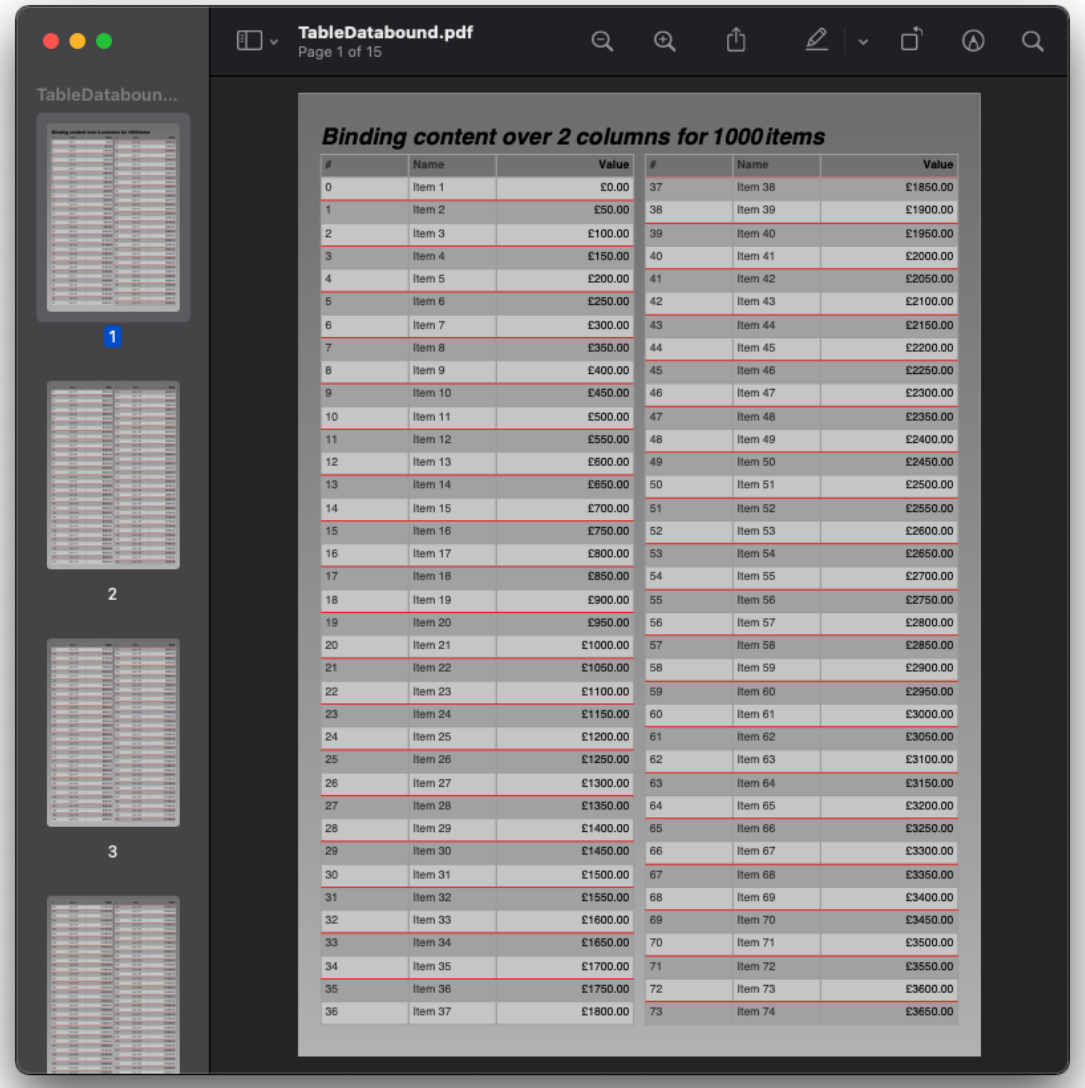
Scryber supports the use of tables with rows, cells and allows nesting, overflow, headings, footers and column-spans.

It also supports the use of binding and repeating at the row and/or the cell level.

By default a table will consume as much width as needed, measured for the first 5 rows, but can be full width (*width:100%*'). The thead and tbody are optional as in html, but help separate the content, and header rows will by default repeat across breaks.

```
<!-- xmlns='http://www.w3.org/1999/xhtml' -->

<table style='width:100%'>
    <thead>
        <tr>
            <td>Header 1</td>
            <td>Header 2</td>
```

(continues on next page)



Binding content over 2 columns for 1000 items

#	Name	Value	#	Name	Value
0	Item 1	£0.00	37	Item 38	£1850.00
1	Item 2	£50.00	38	Item 39	£1900.00
2	Item 3	£100.00	39	Item 40	£1950.00
3	Item 4	£150.00	40	Item 41	£2000.00
4	Item 5	£200.00	41	Item 42	£2050.00
5	Item 6	£250.00	42	Item 43	£2100.00
6	Item 7	£300.00	43	Item 44	£2150.00
7	Item 8	£350.00	44	Item 45	£2200.00
8	Item 9	£400.00	45	Item 46	£2250.00
9	Item 10	£450.00	46	Item 47	£2300.00
10	Item 11	£500.00	47	Item 48	£2350.00
11	Item 12	£550.00	48	Item 49	£2400.00
12	Item 13	£600.00	49	Item 50	£2450.00
13	Item 14	£650.00	50	Item 51	£2500.00
14	Item 15	£700.00	51	Item 52	£2550.00
15	Item 16	£750.00	52	Item 53	£2600.00
16	Item 17	£800.00	53	Item 54	£2650.00
17	Item 18	£850.00	54	Item 55	£2700.00
18	Item 19	£900.00	55	Item 56	£2750.00
19	Item 20	£950.00	56	Item 57	£2800.00
20	Item 21	£1000.00	57	Item 58	£2850.00
21	Item 22	£1050.00	58	Item 59	£2900.00
22	Item 23	£1100.00	59	Item 60	£2950.00
23	Item 24	£1150.00	60	Item 61	£3000.00
24	Item 25	£1200.00	61	Item 62	£3050.00
25	Item 26	£1250.00	62	Item 63	£3100.00
26	Item 27	£1300.00	63	Item 64	£3150.00
27	Item 28	£1350.00	64	Item 65	£3200.00
28	Item 29	£1400.00	65	Item 66	£3250.00
29	Item 30	£1450.00	66	Item 67	£3300.00
30	Item 31	£1500.00	67	Item 68	£3350.00
31	Item 32	£1550.00	68	Item 69	£3400.00
32	Item 33	£1600.00	69	Item 70	£3450.00
33	Item 34	£1650.00	70	Item 71	£3500.00
34	Item 35	£1700.00	71	Item 72	£3550.00
35	Item 36	£1750.00	72	Item 73	£3600.00
36	Item 37	£1800.00	73	Item 74	£3650.00

(continued from previous page)

```
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Content 1</td>
            <td>Content 2</td>
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <td colspan='2'>Wide footer</td>
        </tr>
    </tfoot>
</table>
```

Using the header, row and cell classes in code allows creation of tables too.

```
//using Scryber.Components;

var table = new TableGrid() { FullWidth = true };
var head = new TableHeaderRow();
var h1 = new TableHeaderCell();
h1.Contents.Add(new TextLiteral("Header 1"));
head.Cells.Add(h1);
var h2 = new TableHeaderCell();
h2.Contents.Add(new TextLiteral("Header 2"));
head.Cells.Add(h2);
table.Rows.Add(head);

for(var r = 0; r < 2; r++)
{
    var row = new TableRow();
    for(var c = 0; c < 2; c++)
    {
        var cell = new TableCell();
        cell.Contents.Add(new TextLiteral("Content " + c));
        row.Cells.Add(cell);
    }
    table.Rows.Add(row);
}
var foot = new TableFooterRow();
var fd = new TableFooterCell() { CellColumnSpan = 2 };
fd.Contents.Add(new TextLiteral("Wide footer"));
foot.Cells.Add(fd);
table.Rows.Add(foot);
```

10.33.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

10.33.2 Simple Tables

A simple table with no style or formatting will be output with a single point gray border and 4pt padding on each cell.

Each column will take up as much room as needed (or possible). And the table will be be sized for the widths.

```
<!-- /Templates/Tables/TableSimple.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Tables</title>
</head>
<body style="padding:20pt">
    <table id='FirstTable' >
        <tr>
            <td>Cell 1.1</td>
            <td>Wider Cell 1.2</td>
            <td>Cell 1.3</td>
        </tr>
        <tr>
            <td>Cell 2.1</td>
            <td>Cell 2.2</td>
            <td>Cell 2.3</td>
        </tr>
        <tr>
            <td>Cell 3.1</td>
            <td>Cell 3.2</td>
            <td>Cell 3.3</td>
        </tr>
    </table>
</body>
</html>
```

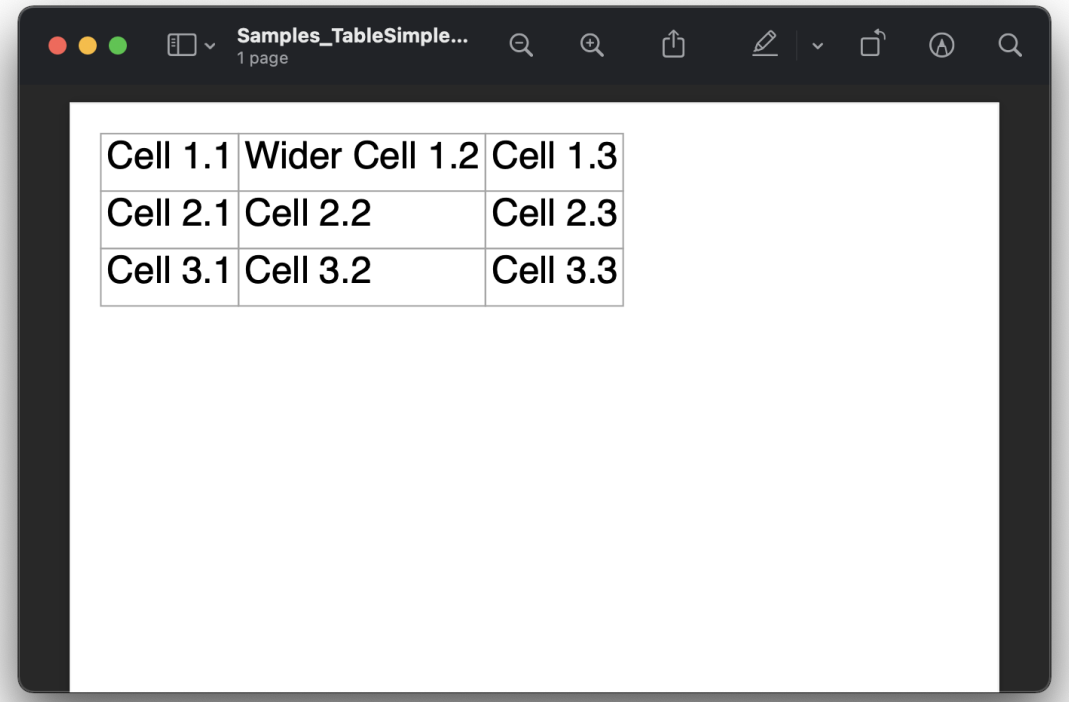
```
//using Scryber.Components;
//Scryber.UnitSamples/TableSamples.cs

public void Table1_SimpleTable()
{
    var path = GetTemplatePath("Tables", "TableSimple.html");

    using (var doc = Document.ParseDocument(path))
    {
        using(var stream = GetOutputStream("Tables", "TableSimple.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

For speed the first 5 rows are tested for desired width, if they are not explicitly set. This allows for giving good measurement of a desired layout without having to double measure an entire table. If the 6th row on a table has a particularly large flowing content, then this will be ignored - set an explicit width on that column, or all the others.

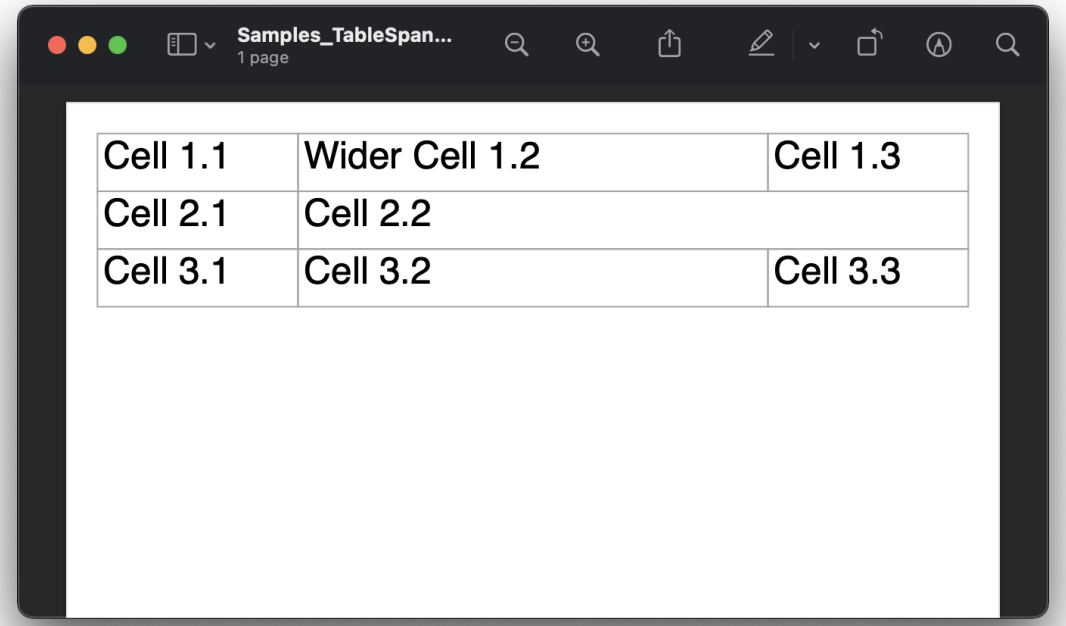
10.33.3 Table width and cell spans

Applying the full-width (`width:100%`) will make the table use all available space in it's container, obeying any fixed column widths. The cells support a column-span attribute to allow multiple column content.




```
<!-- /Templates/Tables/TableSpanned.html -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Tables</title>
</head>
<body style="padding:20pt">
    <table id='FirstTable' style="width:100%">
        <tr>
            <td>Cell 1.1</td>
            <td style="width: 300pt">Wider Cell 1.2</td>
            <td>Cell 1.3</td>
        </tr>
        <tr>
            <td>Cell 2.1</td>
            <td colspan="2">Cell 2.2</td>
        </tr>
        <tr>
            <td>Cell 3.1</td>
            <td>Cell 3.2</td>
            <td>Cell 3.3</td>
        </tr>
    </table>
</body>
</html>
```

```
//Scryber.UnitSamples/TableSamples.cs

public void SpannedTable()
{
    var path = GetTemplatePath("Tables", "TableSpanned.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Tables", "TableSpanned.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

10.33.4 Tables in code

Tables can be created just as easily through code. The `table` has a `Rows` property and each `row` has a `Cells` property. These properties wrap the protected `InnerContent` property from the `PDFContainerComponent` class.

```
//Scryber.UnitSamples/TableSamples.cs

public void CodedTable()
{
```

(continues on next page)

(continued from previous page)

```
    var doc = new Document();

    var pg = new Page();
    doc.Pages.Add(pg);
    pg.Padding = new PDFThickness(20);

    var tbl = new TableGrid();
    pg.Contents.Add(tbl);

    //Full width is equivalent to width:100%
    tbl.FullWidth = true;

    for (int i = 0; i < 3; i++)
    {
        var row = new TableRow();
        tbl.Rows.Add(row);

        for (int j = 0; j < 3; j++)
        {
            if (i == 1 && j == 2)
            {
                //We make the previous cell 2 columns wide rather than add a new one.
                row.Cells[1].CellColumnSpan = 2;
                continue;
            }
            else
            {
                var cell = new TableCell() { BorderColor = PDFColors.Aqua, FontItalic␣
↪= true };
                row.Cells.Add(cell);

                var txt = new TextLiteral("Cell " + (i + 1) + "." + (j + 1));
                cell.Contents.Add(txt);
            }
        }
    }

    using (var stream = DocStreams.GetOutputStream("Samples_TableInCode.pdf"))
    {
        doc.SaveAsPDF(stream);
    }
}
```

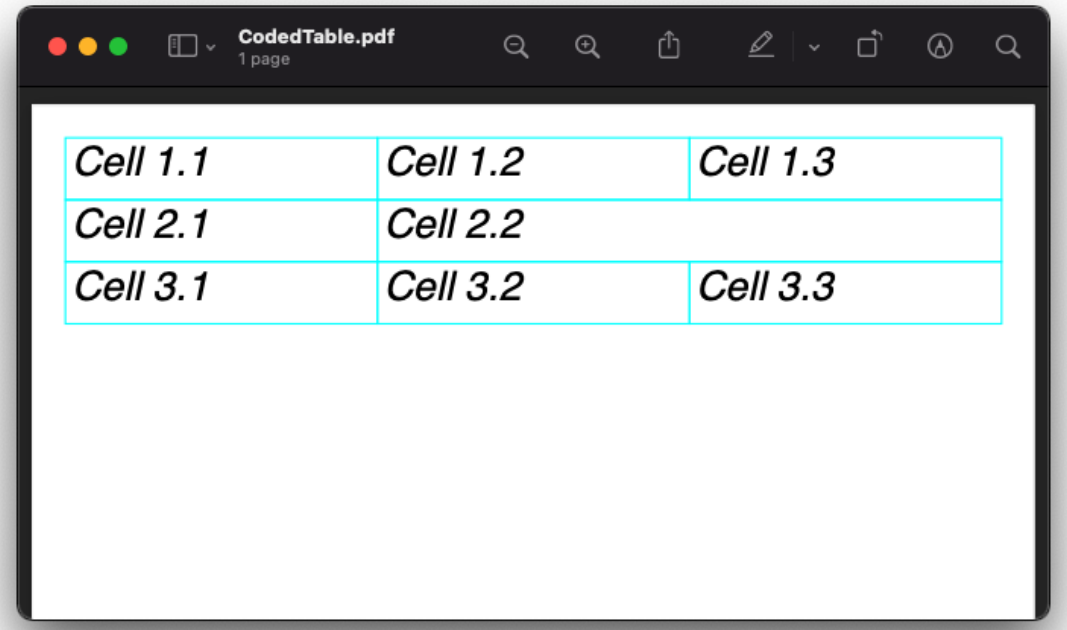
Full size version

Note: The property for the number of columns spanned by a cell is CellColumnSpan. The ColumnCount property will refer to the number of columns to layout inner content with.

It is also possible to access a parsed table to alter the content as needed.

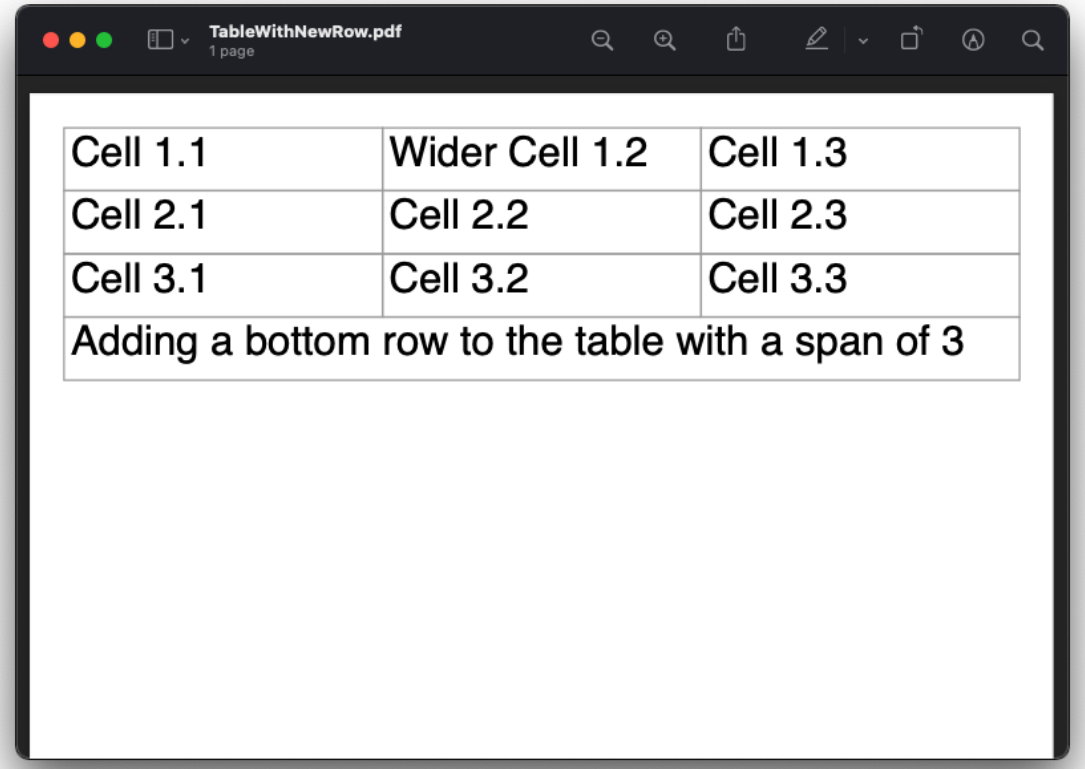
```
//Scryber.UnitSamples/TableSamples.cs

public void ModifyTable()
{
    //Use the simple table sample
```

(continues on next page)



(continued from previous page)

```
    var path = GetTemplatePath("Tables", "TableSimple.html");

    using (var doc = Document.ParseDocument(path))
    {
        //Make full width and add a footer to the table
        if(doc.TryFindAComponentByID("FirstTable", out TableGrid tbl))
        {
            tbl.FullWidth = true;

            var row = new TableRow();
            tbl.Rows.Add(row);

            var span = tbl.Rows[0].Cells.Count;

            var cell = new TableCell();
            cell.Contents.Add(new TextLiteral("Adding a bottom row to the table with␣
↪a span of " + span));
            cell.CellColumnSpan = span;
            row.Cells.Add(cell);
        }

        using (var stream = GetOutputStream("Tables", "TableWithNewRow.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```


TableWithNewRow.pdf
1 page
Cell 1.1
Wider Cell 1.2
Cell 1.3
Cell 2.1
Cell 2.2
Cell 2.3
Cell 3.1
Cell 3.2
Cell 3.3
Adding a bottom row to the table with a span of 3

Full size version

10.33.5 Headers, Footers and overflow

Tables support both headers and footers (single or multiple). The header cells, by default, will repeat across columns and or pages and be in bold, but can be set not to repeat with the `repeat='none'` attribute. (Alternatively, any row can simply be set to repeat with the `repeat='RepeatAtTop'`, and will do so after they have initially been laid out).

Rows support the block styles, except margins, padding and positioning.

Empty cells will still show size and borders, but can be hidden with the `border:none` style.

```html
<!-- /Templates/Tables/TableHeaders.html -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Table Headers and Footers</title>
</head>
<body style="padding:20pt">
    <!-- put the table in a div with 2 columns and a maximum height of 270pt -->
    <div style="max-height: 270pt; font-size:12pt; column-count: 2">

        <table id='TableHead' style="width:100%">
            <thead>
                <tr>
                    <td>Header 1</td>
                    <td>Header 2</td>
                    <td>Header 3</td>
                </tr>
                <!-- This row will not repeat across multiple columns -->
                <tr repeat="none">
                    <td>Header 1</td>
                    <td>Header 2</td>
                    <td>Header 3</td>
                </tr>
            </thead>
            <tbody>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
                <tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>
```

(continues on next page)

(continued from previous page)

```
            </tbody>
            <tfoot style="font-style: italic;">
                <tr>
                    <td colspan="2" style="border:none;"></td>
                    <td>Footer</td>
                </tr>
            </tfoot>
        </table>

    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/TableSamples.cs

public void TableHeaderAndFooter()
{
    var path = GetTemplatePath("Tables", "TableHeaders.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Tables", "TableHeaders.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

The Component classes for Header and Footer rows and cells are `TableHeaderRow`, `TableFooterRow`, `TableHeaderCell` and `TableFooterCell`. They simply inherit from `TableRow` and `TableCell` and can be added to a `TableGrid` and `TableRow` at any point.

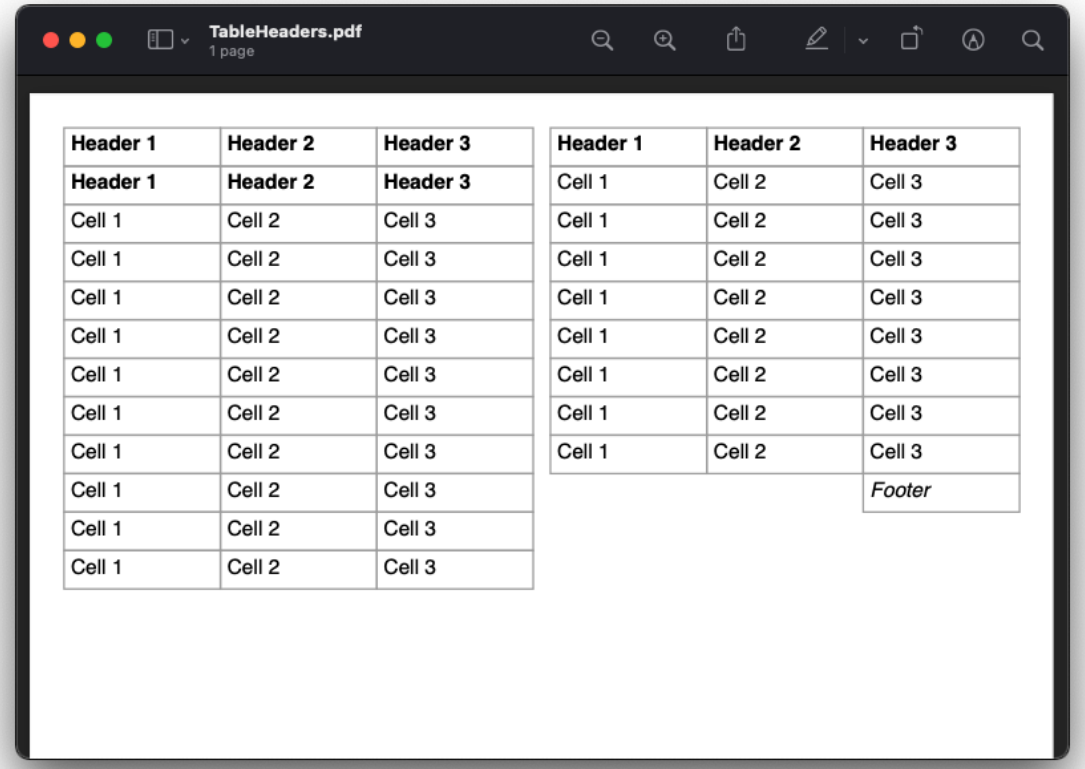
Note: Because of the layout mechanism, repeating cells cannot be accessed or modified between layout itterations (columns or pages). The next table header is from the layout of the original.

10.33.6 Mixed content, styling and nesting

All table cells can contain any content, just like other block components, including other tables, and they also support sizing and alignment of content.

```
<!-- /Templates/Tables/TableNested.html -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Table Headers and Footers</title>
</head>
<body style="padding:20pt">
    <div style="font-size:12pt;">
```

(continues on next page)



Header 1	Header 2	Header 3
Header 1	**Header 2**	**Header 3**
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3

Header 1	Header 2	Header 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
Cell 1	Cell 2	Cell 3
		Footer

(continued from previous page)

```
        <table id='TopTable' style="width:100%">
            <thead>
                <tr>
                    <td colspan="2">Table with mixed content and another nested table
↪</td>
                </tr>
                <tr>
                    <td>Left Side</td>
                    <td>Right Side</td>
                </tr>
            </thead>
            <tbody>
                <tr style="background-color: #AAAAAF;"><td style="min-height:35pt">
↪Cell 1</td><td>Cell 2</td></tr>
                <tr>
                    <td>
                        <img src="../../images/landscape.jpg" style="width:150pt;" />
                        <p style="text-align:center; vertical-align:middle;␣
↪height:60pt; background-color: #AFAFAF">The image above is a beautiful landscape in␣
↪the Cheshire countryside.</p>
                        <table style="width:100%; margin-top: 10pt;">
                            <tr><td>1</td><td>2</td><td style="width:200pt">3</td></
↪tr>
                        </table>
                    </td>
                    <td>
                        <table style="width:100%; margin-top: 10pt">
                            <tr><td>1</td><td>2</td><td>3</td></tr>
                            <tr><td>1</td><td>2</td><td>3</td></tr>
                            <tr><td>1</td><td>2</td><td>3</td></tr>
                            <tr><td>1</td><td>2</td><td>3</td></tr>
                        </table>
                        <p style="text-align:justify">
                            Lorem ipsum dolor sit amet, consectetur adipiscing elit.␣
↪Vivamus pulvinar, ipsum eu molestie elementum,
                            nibh ante ultricies dui, et euismod nulla sapien ac purus.
↪ Morbi suscipit elit tellus, nec elementum lacus dignissim a.
                            Aliquam molestie turpis consectetur rutrum pretium. Orci␣
↪varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.
                            Quisque varius vitae erat sagittis facilisis. Vivamus␣
↪quis tellus quis augue fringilla posuere vitae ac ante. Aliquam ultricies sodales␣
↪cursus.
                            Pellentesque habitant morbi tristique senectus et netus␣
↪et malesuada fames ac turpis egestas.
                            <br />
                            Vestibulum dolor libero, faucibus quis tristique at,␣
↪euismod vitae nunc. Donec vel volutpat urna, eget tristique nunc.
                            Quisque vitae iaculis dolor. Pellentesque habitant morbi␣
↪tristique senectus et netus et malesuada fames ac turpis egestas.
                            Fusce fermentum odio ac feugiat pharetra. Integer sit␣
↪amet elit a urna maximus sollicitudin sit amet sed mauris.
                            Proin finibus nec diam blandit porttitor.
                        </p>
                    </td>
                </tr>
            </tbody>
            <tfoot style="font-style: italic;">
```

(continues on next page)

(continued from previous page)

```
                <tr>
                    <td style="border:none;"></td>
                    <td>Footer</td>
                </tr>
            </tfoot>
        </table>

    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/TableSamples.cs

public void TableMixedNestedContent()
{
    var path = GetTemplatePath("Tables", "TableNested.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Tables", "TableNested.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

10.33.7 Binding to Data

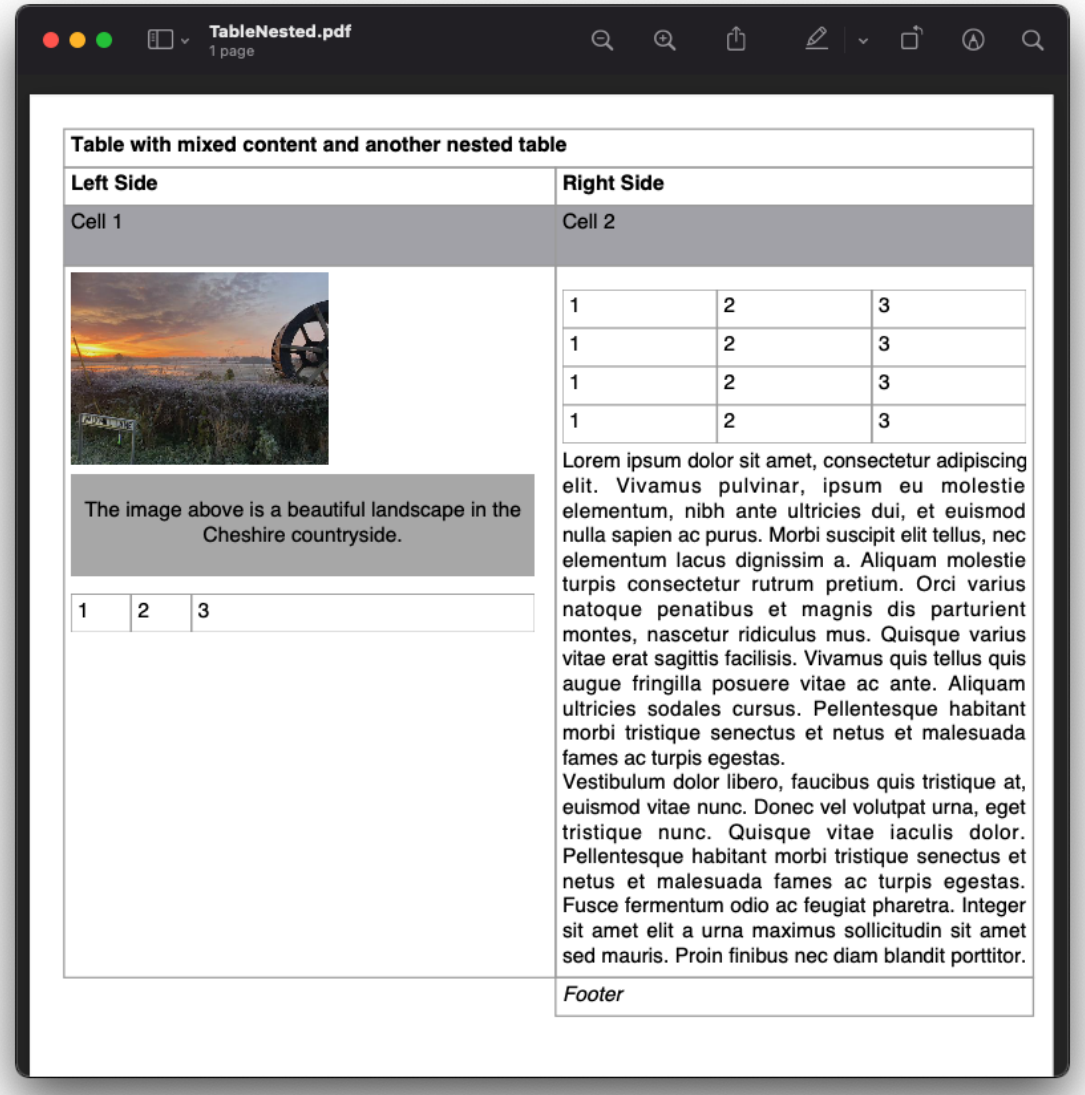
As with all things in scryber. Tables, rows and cells are fully bindable. It is very common to want to layout data in tables so that it can easily be compared.

Tables support the use of the data binding with the `template` tag and `data-bind` attribute.

See binding_databinding for more information on the data binding capabilities of scryber.

```
<!-- /Templates/Tables/TableDatabound.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Table data bound</title>
    <style>
        .grey {
            background-image: linear-gradient(#777, #BBB);
            padding: 20pt
        }

        td.key, td.index {
            color: #333;
            font-size: 10pt;
        }
```

(continues on next page)


TableNested.pdf
1 page
Table with mixed content and another nested table
Left Side
Right Side
Cell 1
Cell 2
1 2 3
1 2 3
1 2 3
1 2 3
The image above is a beautiful landscape in the Cheshire countryside.
1 2 3
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus pulvinar, ipsum eu molestie elementum, nibh ante ultricies dui, et euismod nulla sapien ac purus. Morbi suscipit elit tellus, nec elementum lacus dignissim a. Aliquam molestie turpis consectetur rutrum pretium. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Quisque varius vitae erat sagittis facilisis. Vivamus quis tellus quis augue fringilla posuere vitae ac ante. Aliquam ultricies sodales cursus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.
Vestibulum dolor libero, faucibus quis tristique at, euismod vitae nunc. Donec vel volutpat urna, eget tristique nunc. Quisque vitae iaculis dolor. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Fusce fermentum odio ac feugiat pharetra. Integer sit amet elit a urna maximus sollicitudin sit amet sed mauris. Proin finibus nec diam blandit porttitor.
Footer

(continued from previous page)

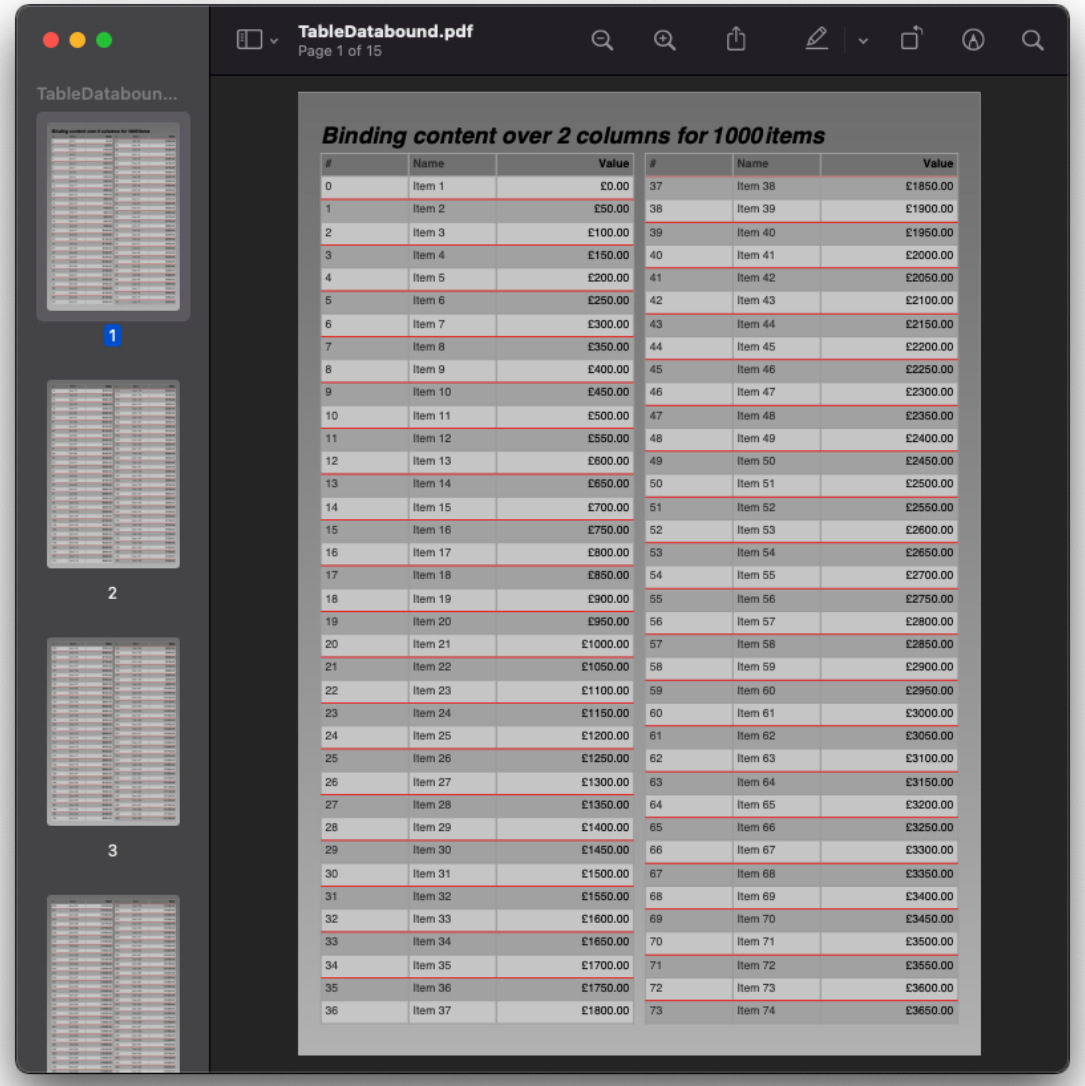
```
        td.value {
            color: black;
            font-size: 10pt;
            text-align:right;
        }

        tr.odd {
            background-color: #AAA;
            border-top: solid 1px red;
        }

        tr.even {
            background-color: #CCC;
        }
    </style>
</head>
<body class="grey">
    <h4>Binding content over 2 columns for {{count(model)}} items</h4>
    <div style="column-count: 2">
        <table id="largeTable" style="width:100%;">
            <thead style="font-weight:bold;">
                <tr>
                    <td class="index">#</td>
                    <td class="key">Name</td>
                    <td class="value" style="width: 120pt">Value</td>
                </tr>
            </thead>
            <template data-bind="{{model}}">
                <tr class="{{if(index() % 2 == 1, 'odd', 'even')}}">
                    <td class="index">{{index()}}</td>
                    <td class="key">{{.Key}}</td>
                    <td class="value">
                        <num value="{{.Value}}" data-format="£##0.00" />
                    </td>
                </tr>
            </template>
        </table>
    </div>
</body>
</html>
```

```
//Scryber.UnitSamples/TableSamples.cs

public void TableBoundContent()
{
    var path = GetTemplatePath("Tables", "TableDatabound.html");

    using (var doc = Document.ParseDocument(path))
    {
        List<dynamic> all = new List<dynamic>();
        for(int i = 0; i < 1000; i++)
        {
            all.Add(new { Key = "Item " + (i + 1).ToString(), Value = i * 50.0 });
        }

        doc.Params["model"] = all;
```

(continues on next page)

(continued from previous page)

```
        using (var stream = GetOutputStream("Tables", "TableDatabound.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



#	Name	Value	#	Name	Value
0	Item 1	£0.00	37	Item 38	£1850.00
1	Item 2	£50.00	38	Item 39	£1900.00
2	Item 3	£100.00	39	Item 40	£1950.00
3	Item 4	£150.00	40	Item 41	£2000.00
4	Item 5	£200.00	41	Item 42	£2050.00
5	Item 6	£250.00	42	Item 43	£2100.00
6	Item 7	£300.00	43	Item 44	£2150.00
7	Item 8	£350.00	44	Item 45	£2200.00
8	Item 9	£400.00	45	Item 46	£2250.00
9	Item 10	£450.00	46	Item 47	£2300.00
10	Item 11	£500.00	47	Item 48	£2350.00
11	Item 12	£550.00	48	Item 49	£2400.00
12	Item 13	£600.00	49	Item 50	£2450.00
13	Item 14	£650.00	50	Item 51	£2500.00
14	Item 15	£700.00	51	Item 52	£2550.00
15	Item 16	£750.00	52	Item 53	£2600.00
16	Item 17	£800.00	53	Item 54	£2650.00
17	Item 18	£850.00	54	Item 55	£2700.00
18	Item 19	£900.00	55	Item 56	£2750.00
19	Item 20	£950.00	56	Item 57	£2800.00
20	Item 21	£1000.00	57	Item 58	£2850.00
21	Item 22	£1050.00	58	Item 59	£2900.00
22	Item 23	£1100.00	59	Item 60	£2950.00
23	Item 24	£1150.00	60	Item 61	£3000.00
24	Item 25	£1200.00	61	Item 62	£3050.00
25	Item 26	£1250.00	62	Item 63	£3100.00
26	Item 27	£1300.00	63	Item 64	£3150.00
27	Item 28	£1350.00	64	Item 65	£3200.00
28	Item 29	£1400.00	65	Item 66	£3250.00
29	Item 30	£1450.00	66	Item 67	£3300.00
30	Item 31	£1500.00	67	Item 68	£3350.00
31	Item 32	£1550.00	68	Item 69	£3400.00
32	Item 33	£1600.00	69	Item 70	£3450.00
33	Item 34	£1650.00	70	Item 71	£3500.00
34	Item 35	£1700.00	71	Item 72	£3550.00
35	Item 36	£1750.00	72	Item 73	£3600.00
36	Item 37	£1800.00	73	Item 74	£3650.00

Full size version

10.33.8 Alternating row styles

The conditional function `if(index(), [true expression], [false expression])` was applied above for an alternating class to the table rows, even when we don't know how many items there are. We could have combined this with the `concat()` function to apply multiple classes.

See binding/binding_functions for more examples and information.

Note: Scryber also includes the data-style-identifier which can improve the speed of output for data bound repeats but can impact the styles within repeating content.

10.33.9 Not (Currently) Supported

There are some things that are not supported on tables.

1. Scryber does not support the `rowspan` property. This is simply a case of complex calculation, and we do expect to implement in the future.
2. Table rows cannot be split across pages. Due to page layout constraints rows should not flow. It has an impact on the column layout, but we may implement in the future.
3. Rows do not support margins, padding, or position. This is a constraint of the layout.
4. Cells do not work well with inner content in multiple columns. It may be that once balanced columns are sorted this automatically resolves itself.

10.34 More Containers - TD

Preamble

10.34.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

See the tables_reference for more details on what is supported in tables.

10.34.2 Binding Simple numbers

10.34.3 Binding Dates and Times

10.34.4 Calculating values

10.34.5 Building in code

10.35 B, I, U, etc - TD

Preamble

10.35.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

See the tables_reference for more details on what is supported in tables.

10.35.2 Binding Simple numbers

10.35.3 Binding Dates and Times

10.35.4 Calculating values

10.35.5 Building in code

10.36 Adding Images to documents - TD

Preamble

10.36.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

See the tables_reference for more details on what is supported in tables.

10.36.2 Binding Simple numbers

10.36.3 Binding Dates and Times

10.36.4 Calculating values

10.36.5 Building in code

10.37 Links in and out of documents

Within a document, it's easy to add a link to another component, another page, another document, or remote web link.

Scryber supports the standard `a` anchor tag with the `href` attribute for linking between items. As a component itself it has no explicit content, but it can support any type of content within it.

The default style for the `a` tag has blue text with an underline.

```
<!-- xmlns='http://www.w3.org/1999/xhtml' -->

<a href='https://www.scyber.co.uk' >Link to scryber</a>

<a href='#componentId' >Link to local component with ID 'componentId'</a>

<a href='LastPage' >Link to the last page in the document</a>
```

The `a` tag has a standard base class of `Scryber.Components.Link`. The base `Link` class does not infer any styles, and inner content should be styled appropriately.


LinksStyledTOC.pdf
Page 1 of 5
LinksStyledTO...
1
2
3
4
Table of Contents
first 2
second 3
third 4
fourth 5
<< < Page 1 of 5 > >>

```
//using Scryber.Components

var href = new Link() { File = "https://scryber.co.uk", Action = LinkAction.Uri  };
href.Contents.Add(new TextLiteral("Link to scryber"));

var complink = new Link() { Destination = "#componentId", Action = LinkAction.
↪Destination };
complink.Contents.Add(new TextLiteral("Link to local component with ID 'componentId'
↪"));

var nav = new Link() { Action = LinkAction.LastPage };
nav.Contents.Add(new TextLiteral("Link to the last page in the document"));
nav.Style.Text.Decoration = Text.TextDecoration.Underline;
```

10.37.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

10.37.2 The Anchor Component

An ‘a‘nchor tag is an invisible component (although styles will be applied to content within it) that allows linking to other components. The href attribute supports 3 types of content.

- **url** - A relative or remote link to a document or webpage.
- **Named Action** - These are the standard links to and from pages (see below).
- **#id** - If set, then the link is an action for a different document or Url. Effectively like the href of an anchor tag in html.

The content within a link can be anything, including images; text; svg components and more. There can also be more than one component within the link.

By default the `a` tag is inline with a style applied for any inner text of underlined blue, but it does support the use of being positioned as a block and all other styling options.

10.37.3 Relative or Remote Links

Using the href attribute a remote link can be made to any url or local document. Links can also contain images or any other content, and can use the target=’_blank’ to open in a new tab.

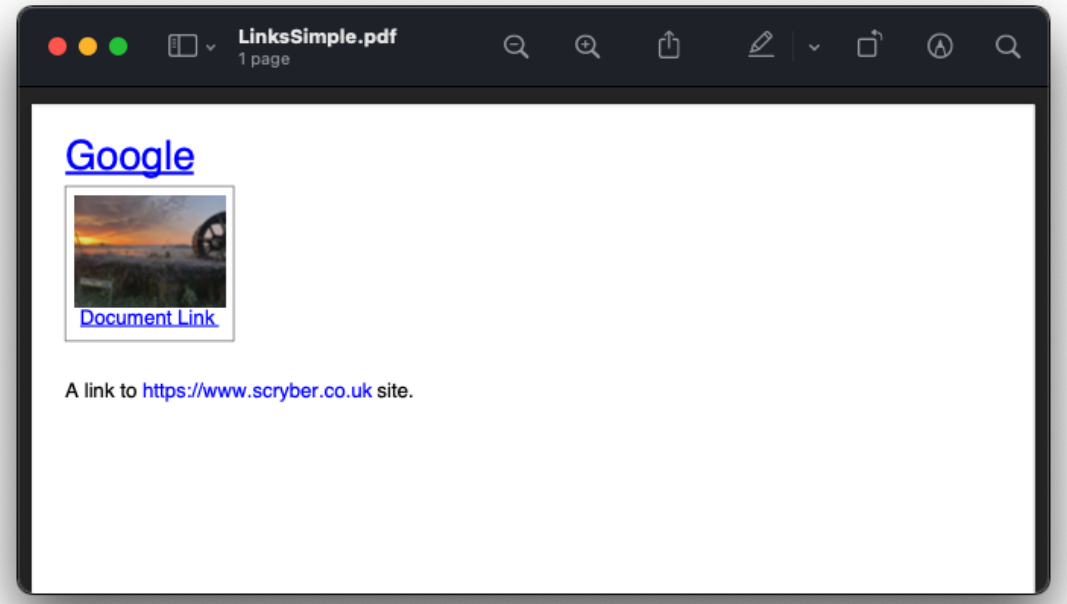
```
<!-- /Templates/Links/LinksSimple.html -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Remote and Relative Links</title>
</head>
<body style="padding:20pt">

    <!-- A web link to the google home page -->
    <a href="https://www.google.com" target="_blank">Google</a><br />

    <!-- a link to a local pdf that will open in a new reader tab or window -->
    <a href="AnotherDocument.pdf" target="_blank">
```

(continues on next page)

(continued from previous page)

```
        <div style="border:solid 1px gray; padding:5pt; text-align:center; font-size:␣
↪12pt; width: 100pt;">
            <img src="../../images/landscape.jpg" />Document Link
        </div>
    </a><br/>
    <br/>
    <!-- binding the url to a document parameter -->
    <span style="font-size:12pt;">A link to <a href="{{siteurl}}" style="text-
↪decoration: none;">{{siteurl}}</a> site.</span>
</body>
</html>
```

```
//Scryber.UnitSamples/LinkSamples.cs

public void SimpleNavigationLinks()
{
    var path = GetTemplatePath("Links", "LinksSimple.html");

    using (var doc = Document.ParseDocument(path))
    {
        using (var stream = GetOutputStream("Links", "LinksSimple.pdf"))
        {
            doc.Params["siteurl"] = "https://www.scryber.co.uk";
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

Note: By the very nature of documents, they can be moved around. Relative links may not be appropriate, but can be bound and converted to absolute links as needed.

10.37.4 Page Named Action

Navigation within a document can be done using a predefined action for the reader to take. The possible actions are (case insensitive) as follows:

- FirstPage
- PrevPage
- NextPage
- LastPage

These are self-evident in their purpose, and no other attributes need defining. It does not matter what page they are put on, they will perform the action if possible.

```
<a href='nextpage' >Next Page Link</a>
```

For example we can create a navigation set of links.

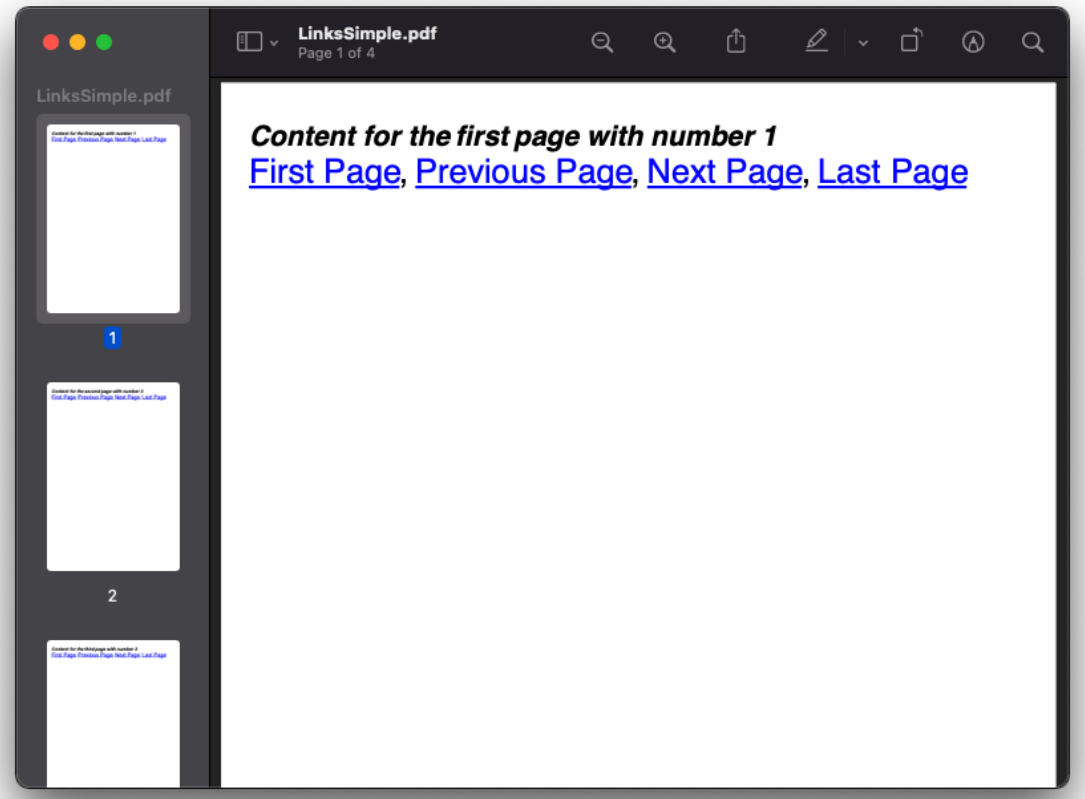
```
<!-- /Templates/Links/LinksNamedActions.html -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Simple Links</title>
    <style>
        .break-before{ page-break-before: always; }
    </style>
</head>
<body style="padding:20pt">
    <template data-bind="{{pages}}">
        <div class="{{if(index() > 0, 'break-before', 'break-none')}}">
            <h4>Content for the {{.Id}} page with number <page /></h4>
            <a href="FirstPage">First Page</a>,
            <a href="PreviousPage">Previous Page</a>,
            <a href="NextPage">Next Page</a>,
            <a href="LastPage">Last Page</a>
        </div>
    </template>
</body>
</html>
```

```
//Scryber.UnitSamples/LinkSamples.cs

public void NamedActionLinks()
{
    var path = GetTemplatePath("Links", "LinksNamedActions.html");

    using (var doc = Document.ParseDocument(path))
    {
        var pages = new[] { new { Id = "first" }, new { Id = "second" }, new { Id =
↪"third" }, new { Id = "fourth" } };
```

(continues on next page)

(continued from previous page)

```
        doc.Params["pages"] = pages;

        using (var stream = GetOutputStream("Links", "LinksNamedActions.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

In this sample we are binding to an array of strings, and then setting the class on an outer div, so that there is a page break before the div on every itteration *except* the first

See ../overview/parameters_and_expressions for more information on binding to data and objects.

10.37.5 Styling Links

Although the default style is inline with blue text and underline. Links can be styled independently.

In this example we use a footer template for the navigation links between pages (see ../overview/pages_and_sections for more on page headers and footers).

We style the footer with a table where the links are set in 50pt wide cells, and the centre cell takes up the rest of the space for a Page N of Total.

```
<!-- /Templates/Links/LinksStyledFooter.html -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Navigation Links</title>
    <style>

        .break-before{ page-break-before: always; }

        h4{ margin: 20pt; padding: 20pt; background-color: #AAA; }

        /* Styled bottom footer of the page */
        footer{ background-image: linear-gradient(#000, #333); padding: 4pt;}

        footer .nav{ width:100%; font-size: 14pt; }

        /* Standard table cell style */
        footer .nav td { border:none; text-align: center; vertical-align:bottom;␣
↪color:white; }

        /* The navigation link cells are 50pt */
        footer .nav-item { width: 50pt; }

        /* The links are white with no underline */
        footer .nav-item > a { font-weight: bold; text-decoration: none; color: white;
↪}

    </style>
</head>
<body>

    <template data-bind="{{pages}}">
        <div id="{{.Id}}" class="{{if(index() > 0, 'break-before', 'break-none')}}">
            <h4>Content for the {{.Id}} page with number <page /></h4>
        </div>
    </template>

    <footer>
        <table class="nav">
            <tr>
                <td class="nav-item">
                    <a href="FirstPage">&lt;&lt;</a>
                </td>
                <td class="nav-item">
                    <a href="PreviousPage">&lt;</a>
                </td>
                <td>
                    Page <page /> of <page property="total" />
                </td>
                <td class="nav-item">
                    <a href="NextPage">&gt;</a>
                </td>
                <td class="nav-item" >
                    <a href="LastPage">&gt;&gt;</a>
```

(continues on next page)

(continued from previous page)

```
                </td>
            </tr>
        </table>
    </footer>
</body>
</html>
```

```
//Scryber.UnitSamples/LinkSamples.cs

public void StyledFooterNavigationLinks()
{
    var path = GetTemplatePath("Links", "LinksStyledFooter.html");

    using (var doc = Document.ParseDocument(path))
    {
        var pages = new[] { new { Id = "first" }, new { Id = "second" }, new { Id =
↪"third" }, new { Id = "fourth" } };
        doc.Params["pages"] = pages;

        using (var stream = GetOutputStream("Links", "LinksStyledFooter.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version

Note: The white color is applied to the 'a'nchor tag as well as the cell, because the default style of blue would override the inherited white color from the cell class.

10.37.6 (H)Over Styles

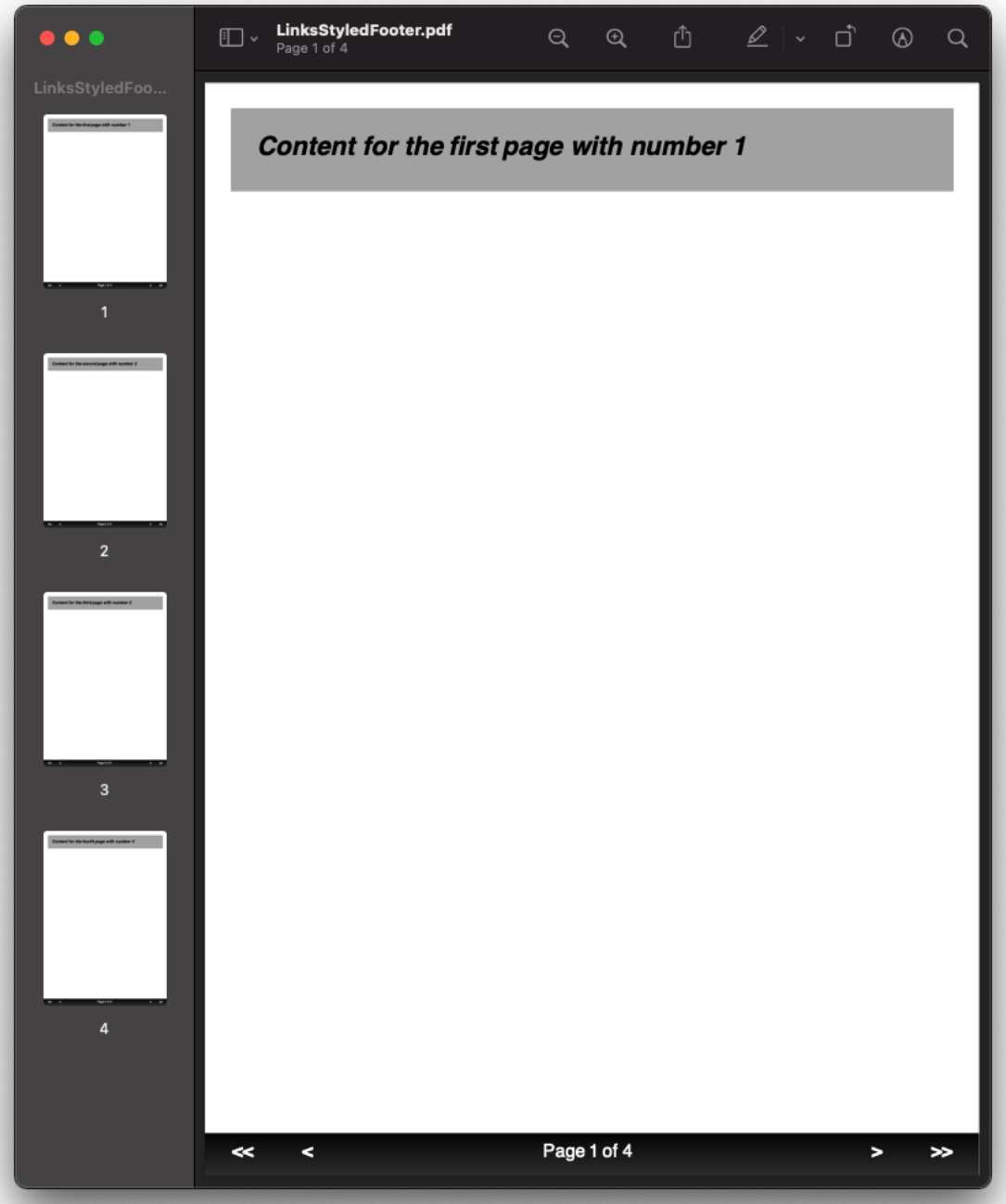
There is no support for hover, down, over, clicked within the scryber pdf support. At the moment the use of the pointer cursor over a link and it's default style is what is available.

10.37.7 Linking within documents

When navigating around the document, scryber supports the direct linking to a specific page or component using the id being referenced attribute. Prefix with a # (hash) to identify it is an element witin the document.

```
<a href='#Id'>Link to item</a>


<div id='Id'>Content to link to</div>
```

This can also be data bound, so with our data we can add a first page for a table of contents linking to each of the headings in the following pages. (Removing the need to check for a first page on the breaks.)


LinksStyledFooter.pdf
Page 1 of 4
LinksStyledFoo...
1
2
3
4
Content for the first page with number 1
Page 1 of 4

10.37.8 A Table of Contents

```
<!-- /Templates/Links/LinksStyledTOC.html -->

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Table of contents Links</title>
    <style>

        .break-before{ page-break-before: always; }

        h4{ margin: 20pt; padding: 20pt; background-color: #AAA; }

        footer{ background-image: linear-gradient(#000, #333); padding: 4pt;}

        footer .nav{ width:100%; font-size: 14pt; }

        footer .nav td { border:none; text-align: center; vertical-align:bottom;
↪color:white; }

        footer .nav-item { width: 50pt; }

        footer .nav-item > a { font-weight: bold; text-decoration: none; color: white;
↪}

        /* Table of contents styles */

        table.toc { margin: 0pt 20pt 0pt 20pt; font-size: 12pt; width:100%; }

        table.toc td {padding: 0; border:none; margin-bottom: 10px; text-decoration:
↪none; }

        table.toc td a { text-decoration:none; }

        table.toc td.name hr.spacer { display:inline; margin-top: 12pt; stroke: #777;
↪stroke-dasharray: 1 2; }
    </style>
</head>
<body>

    <h4>Table of Contents</h4>
    <table class="toc">
        <!-- Loop over the items for the content of the table -->
        <template data-bind="{{pages}}">
            <tr>
                <td class="name">
                    <!-- The link is set to the concatenation of # and Id in the data
                        We also use a hr spacer with a dotted style -->
                    <a href="{{concat('#',.Id)}}">{{.Id}} </a> <hr class="spacer" />
                </td>
                <td class="page-num" style="width:20pt;">
                    <!-- The page for will look up the page number of the item
↪referenced too -->
                    <a href="{{concat('#',.Id)}}"><page for="{{concat('#',.Id)}}" /></
↪a>
```

(continues on next page)

(continued from previous page)

```
                </td>
            </tr>
        </template>
    </table>

    <template data-bind="{{pages}}">
        <!-- Each heading wrapper has the id from the pages data -->
        <div id="{{.Id}}" class="break-before">
            <h4>Content for the {{.Id}} page with number <page /></h4>
        </div>
    </template>

    <footer>
        <table class="nav">
            <tr>
                <td class="nav-item">
                    <a href="FirstPage">&lt;&lt;</a>
                </td>
                <td class="nav-item">
                    <a href="PreviousPage">&lt;</a>
                </td>
                <td>
                    Page <page /> of <page property="total" />
                </td>
                <td class="nav-item">
                    <a href="NextPage">&gt;</a>
                </td>
                <td class="nav-item" >
                    <a href="LastPage">&gt;&gt;</a>
                </td>
            </tr>
        </table>
    </footer>
</body>
</html>
```

```
//Scryber.UnitSamples/LinkSamples.cs

public void StyledFooterWithTOCLinks()
{
    var path = GetTemplatePath("Links", "LinksStyledTOC.html");

    using (var doc = Document.ParseDocument(path))
    {
        var pages = new[] { new { Id = "first" }, new { Id = "second" }, new { Id =
↪"third" }, new { Id = "fourth" } };
        doc.Params["pages"] = pages;

        using (var stream = GetOutputStream("Links", "LinksStyledTOC.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```

Full size version


LinksStyledTOC.pdf
Page 1 of 5
LinksStyledTO...
1
2
3
4
Table of Contents
first 2
second 3
third 4
fourth 5
<< < Page 1 of 5 > >>

10.37.9 Adding links in code

The component class for using links in code is `Scryber.Components.Link` There are 3 primary properties to use for setting what is done when the link is clicked.

To perform one of the Named actions use the `Action` property, setting to the pre-defined `NextPage`, `PrevPage` etc.

```
var link = new Link()
{
    Action = LinkAction.NextPage
};
```

To link to a component within the current document set the `Destination` property value to the id of the component to look for.

```
var link = new Link()
{
    Action = LinkAction.Destination,
    Destination = "#ComponentID",
};
```

To link to a remote page or site set the `File` property to the required url.

```
var link = new Link()
{
    Action = LinkAction.Uri,
    File = "https://www.scryber.co.uk",
};
```

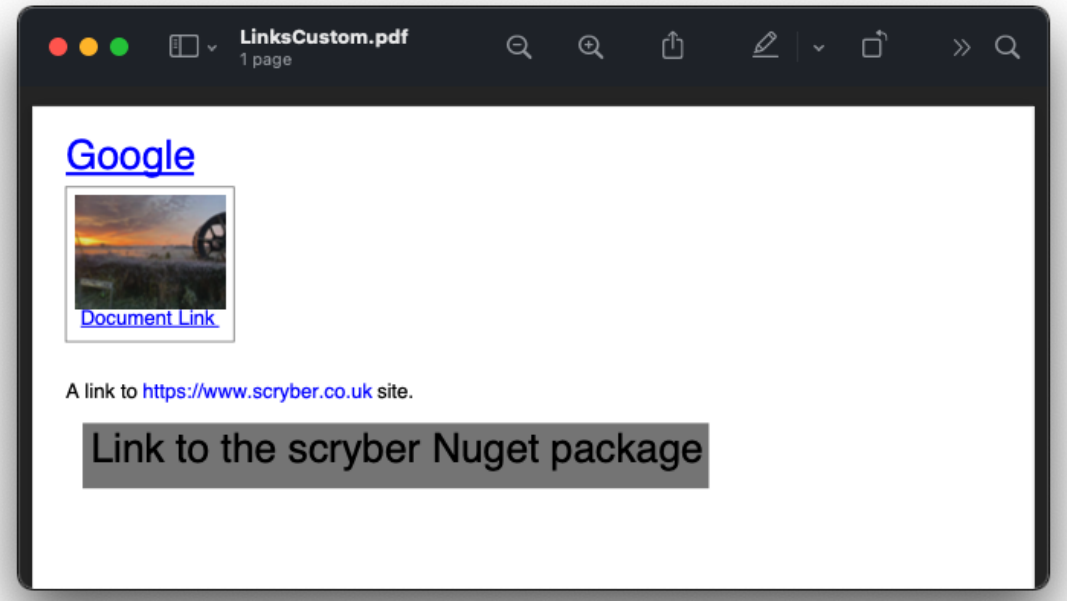
As a container, links can still have any content inside them, and be placed anywhere in the visual content of the document.

```
//Scryber.UnitSamples/LinkSamples.cs

public void SimpleLinksWithCustomAddition()
{
    //template from our first example
    var path = GetTemplatePath("Links", "LinksSimple.html");

    using (var doc = Document.ParseDocument(path))
    {
        doc.Params["siteurl"] = "https://www.scryber.co.uk";

        //create a new link

        var link = new Link()
        {
            Action = LinkAction.Uri,
            File = "https://www.nuget.org/packages/Scryber.Core/",
            Margins = new PDFThickness(10),
            Padding = new PDFThickness(5),
            BackgroundColor = PDFColors.Gray,
            PositionMode = PositionMode.Block
        };

        //add some inner content
```

(continues on next page)

(continued from previous page)

```
        link.Contents.Add(new TextLiteral("Link to the scryber Nuget package"));

        //add it to the page (at the end)

        var pg = doc.Pages[0] as Page;
        pg.Contents.Add(link);

        using (var stream = GetOutputStream("Links", "LinksCustom.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}
```



Full size version

Note: The base link class does not add the blue underlined default style to the output. This can either be added as needed, or styled in any other way.

10.37.10 Fixing Broken Links

By default, when a link is not found for a destination, then it will not be enabled however the style and the output will still be honoured. If links are not working in a document then the, as always, the output trace log can be inspected to see if an error is reported, or change the document parsing mode to strict.

This can all be done either with the `<?scryber append-log='true' ?>` processing instruction or on the

document itself `doc.ConformanceMode = ParserConformanceMode.Strict`.

```
<!DOCTYPE html>
<?scryber append-log='true' ?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Broken Navigation Links</title>
</head>
<body>

    <h4>Links with nowhere to go</h4>
    <div>
        <!-- Loop over the items for the content of the table -->
        <template data-bind="{{pages}}">
        <a href="{{concat('#',.Id)}}">{{.Id}}</a>,
        </template>
    </div>

</body>
</html>
```

Full size version

See ../overview/scryber_output for more about the tracing and logging in document output.

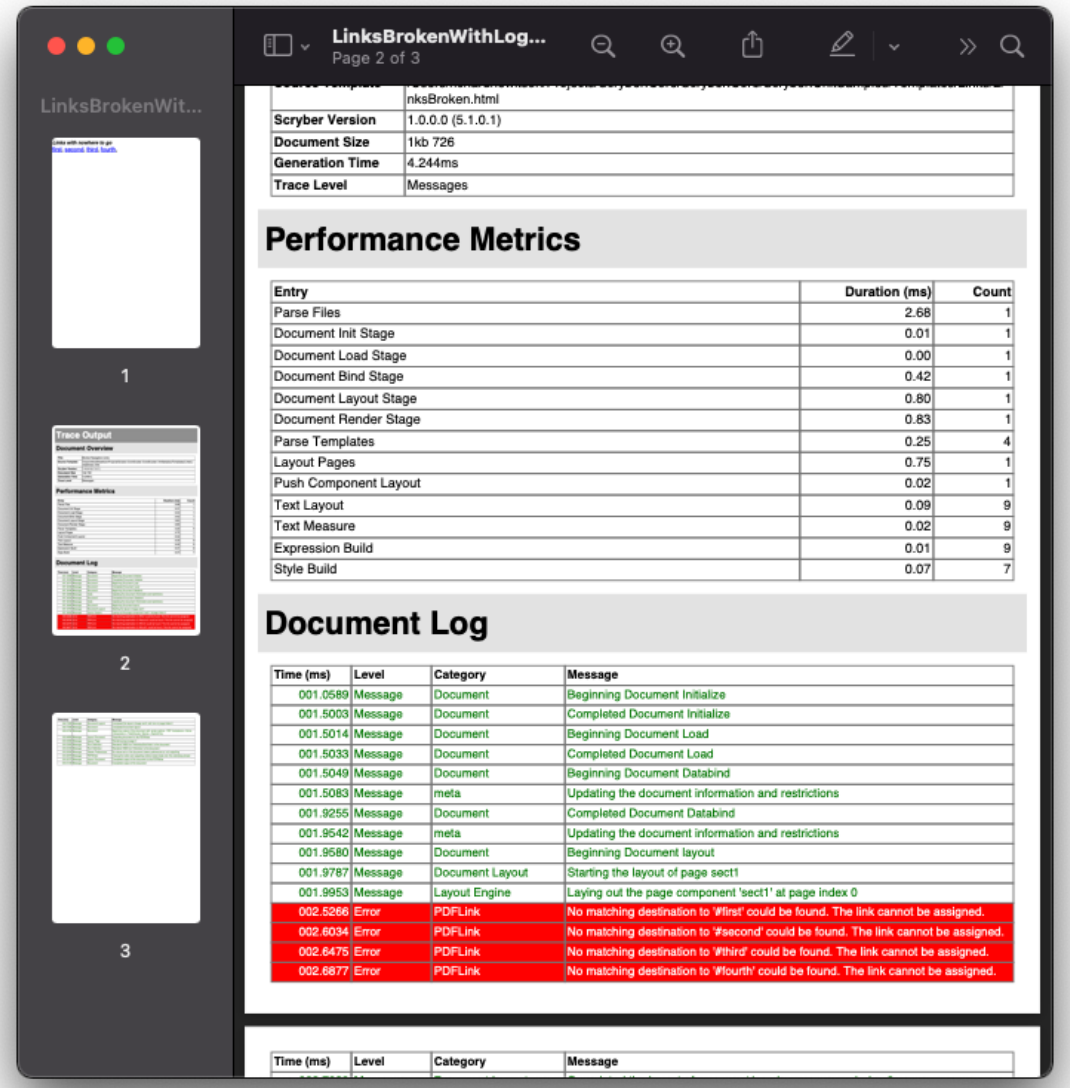
10.38 Numbers, Dates and Times - PD

Within the content of pages the use of numbers, dates and times can often be frought with cultural and positioning issues. Scryber supports the formatting of these types of content with specific elements

```
<!-- xmlns='http://www.w3.org/1999/xhtml' -->

<!-- output as british pounds -->
<num value='100' data-format='£#0.00' />

<!-- output current date time as a long date -->
<time data-format='dd MMMM yyyy' />

<!-- output specific date time as a month -->
<time datetime='2021-09-21 12:00:00' data-format='MMM' />

<!-- the values can also be calculated at run time -->
<num value='{{model.total * 0.2}}' data-format='£#0.00'>
```

They can also be created and used in code

```
//using Scryber.Components;

var num = new Number() { NumberFormat = "£#0.00", Value = 100 };

var dt = new Date() { DateFormat = "D", Value = DateTime.Now };

//look up an existing component
if(parsedDoc.TryFindAComponentById("MyDateTime", out Date found))
    found.Value = DateTime.Now.AddDays(30);
```


LinksBrokenWithLog...
Page 2 of 3
LinksBrokenWit...
1
2
3
nksBroken.html
Scryber Version 1.0.0.0 (5.1.0.1)
Document Size 1kb 726
Generation Time 4.244ms
Trace Level Messages
Performance Metrics
Entry | Duration (ms) | Count
Parse Files 2.68 1
Document Init Stage 0.01 1
Document Load Stage 0.00 1
Document Bind Stage 0.42 1
Document Layout Stage 0.80 1
Document Render Stage 0.83 1
Parse Templates 0.25 4
Layout Pages 0.75 1
Push Component Layout 0.02 1
Text Layout 0.09 9
Text Measure 0.02 9
Expression Build 0.01 9
Style Build 0.07 7
Document Log
Time (ms) | Level | Category | Message
001.0589 Message Document Beginning Document Initialize
001.5003 Message Document Completed Document Initialize
001.5014 Message Document Beginning Document Load
001.5033 Message Document Completed Document Load
001.5049 Message Document Beginning Document Databind
001.5083 Message meta Updating the document information and restrictions
001.9255 Message Document Completed Document Databind
001.9542 Message meta Updating the document information and restrictions
001.9580 Message Document Beginning Document layout
001.9787 Message Document Layout Starting the layout of page sect1
001.9953 Message Layout Engine Laying out the page component 'sect1' at page index 0
002.5266 Error PDFLink No matching destination to '#first' could be found. The link cannot be assigned.
002.6034 Error PDFLink No matching destination to '#second' could be found. The link cannot be assigned.
002.6475 Error PDFLink No matching destination to '#third' could be found. The link cannot be assigned.
002.6877 Error PDFLink No matching destination to '#fourth' could be found. The link cannot be assigned.
Time (ms) | Level | Category | Message

The general use for the `num` and `time` tag is when binding data to a parameters, as other content is treated simply as text.

10.38.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

See the tables_reference for more details on what is supported in tables.

10.38.2 Binding Simple numbers

10.38.3 Binding Dates and Times

10.38.4 Calculating values

10.38.5 Building in code

10.39 Page Headers and Footers.

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

The use of the page-break-before and after is supported on any content to force a new page is set to 'always' on a section, but can, along with page-break-after, be set and supported on any component tag

The body has an optional header and footer that will be used on every page if set.

Scryber also supports the use of the @page rule to be able to change the size and orientation of each of the pages either as a whole, or within a section or tag.

10.39.1 The body and its content

A single page has a structure of optional elements

- header - Optional, but always sited at the top of a page
- Sited between the Header and Footer is any content to be included within the page.
- footer - Optional, but always sited at the bottom of a page

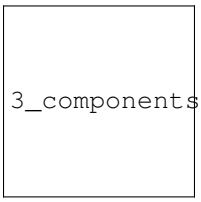
If a page has a header or footer the available space for the content will be reduced.

```
<!-- -->
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style>

    header, footer{
            padding: 10pt;
            background-color: #333;
            color: #EEE;
            border-bottom: 1px solid black;
            border-top: 1px solid black;
```

(continues on next page)

(continued from previous page)

```
    }

    h1{
        padding: 20pt;
    }

    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>
    <h1>This is the content</h1>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```



Note: Any styles set on the body will be applied to the header and footer as well. e.g. padding or margins.

10.39.2 Flowing Pages

If the size of the content is more than can fit on a page it will overflow onto another page. Repeating any header or footer.

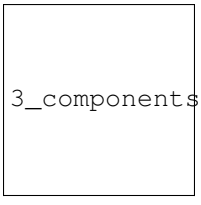
```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style>

    header, footer {
        padding: 10pt;
        background-color: #333;
        color: #EEE;
        border-bottom: 1px solid black;
        border-top: 1px solid black;
    }

    body h1, body div {
        margin: 20pt;
    }

    body div.content {
```

(continues on next page)

(continued from previous page)

```
        font-size:12pt;
        padding: 4pt;
        border: solid 1px silver;
    }

    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>
    <h1>This is the content</h1>
    <div class='content'>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas scelerisque␣
↪porttitor urna.
    Duis pellentesque sem tempus magna faucibus, quis lobortis magna aliquam. Nullam␣
↪eu risus
    facilisis sapien fermentum condimentum. Pellentesque ut placerat diam, sed␣
↪suscipit nibh.
    Integer dictum dolor vel finibus imperdiet. Orci varius natoque penatibus et␣
↪magnis dis
    parturient montes, nascetur ridiculus mus. Integer congue turpis at varius␣
↪porttitor.
    <!-- Truncated for brevity -->
    nec faucibus ipsum bibendum sed. Nunc tristique risus eu quam porttitor blandit.
    In erat mauris, imperdiet a venenatis eu, tempus a nunc.
    <br/>
    Nullam et erat vel nisl suscipit volutpat id vitae massa. Nunc volutpat feugiat␣
↪iaculis.
    Mauris sit amet eleifend augue. Nulla imperdiet eu mauris nec consequat. Donec a␣
↪urna blandit,
    porttitor libero vel, rutrum diam. Fusce scelerisque diam eu rutrum vestibulum.
    Vivamus a quam in nisi euismod laoreet. Morbi mauris augue, lobortis id volutpat␣
↪in,
    venenatis ut ex. Donec euismod risus eros, dapibus tincidunt dolor varius id.
    </div>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```

Here we can see that the content flows naturally onto the next page, including the padding and borders. And the header and footer are shown on the second page.



10.39.3 Page breaks

When using a <section> it will, by default, force a break in the pages using the before the component, so that it flows nicely onto a new page and begins the new content from there. (the default style is page-break-before:always)

This behaviour can can be stopped by applying the css attribute for 'page-break-before:avoid' value, and a page break can also be applied to any element using the style 'page-break-before:always' (or 'page-break-after:always').

Margins, padding, boarder and depth should be preserved during the page break, and the engine will try and layout the content appropriately for breaks inside nested elements.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        header, footer {
        padding: 10pt;
        background-color: #333;
        color: #EEE;
        border-bottom: 1px solid black;
        border-top: 1px solid black;
        }

        body .content {
        margin: 20pt;
        font-size:12pt;
        padding: 4pt;
        border: solid 1px silver;
        }

    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>
    <h1>This is the content</h1>

    <!-- section that does not force a new page (so that it stays on the first page --
↪>
    <section class='content' style="page-break-before:avoid">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas scelerisque␣
↪porttitor urna.
        <!-- Truncated for brevity -->
        Class aptent taciti sociosqu ad litora torquent per conubia nostra, per␣
↪inceptos himenaeos.
        Praesent mollis tempor enim.<br />

    </section>

    <!-- This will be default appear on a new page -->
    <section class='content'>
        Nullam et erat vel nisl suscipit volutpat id vitae massa. Nunc volutpat␣
↪feugiat iaculis.
        Mauris sit amet eleifend augue. Nulla imperdiet eu mauris nec consequat.␣
↪Donec a urna blandit,
        <!-- Truncated for brevity -->
```

(continues on next page)

(continued from previous page)

```
        sagittis dignissim volutpat. Integer efficitur euismod lectus at varius.␣
↪Vestibulum euismod massa mauris.
        Mauris laoreet urna est, et tristique velit lobortis eu.
    </section>

    <!-- Any tag can force a new page within the document flow, and it does not have␣
↪to be at the
        root level. Borders and spacing will be preserved as much as possible -->
    <div class="content">
        The inner content will be on a new page.
        <div class='content' style="page-break-before:always;">
            Phasellus luctus dapibus nisi, et pulvinar neque ultrices vitae.␣
↪Pellentesque quis purus felis.
            <!-- Truncated for brevity -->
            venenatis ut ex. Donec euismod risus eros, dapibus tincidunt dolor varius␣
↪id.
        </div>
        After the content.
    </div>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```

3_components/images/SectionsOverflow.png

10.39.4 Page size and orientation

When outputting a page the default paper size is ISO A4 Portrait (210mm x 29.7mm), however Scryber supports setting the paper size either on the section or via styles to the standard ISO or Imperial page sizes, in landscape or portrait.

- **ISO 216 Standard Paper sizes**
  - A0 to A9
  - B0 to B9
  - C0 to C9
- **Imperial Paper Sizes**
  - Quarto, Foolscap, Executive, GovermentLetter, Letter, Legal, Tabloid, Post, Crown, LargePost, Demy, Medium, Royal, Elephant, DoubleDemy, QuadDemy, Statement,

The body or a section can only be 1 size of paper, but different sections (or page breaks) can be different pages and can have different sizes.

An @page { ... } rule will apply to all pages in the document.

To specify an explicit named page size use the name after the @page rule, and then identify the rule with the page css declaration either on the tag style or in css. The same priories will be applied if multiple page values are matched.

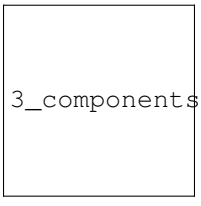
To revert back to the default size use a value of auto or initial.

```
<?xml version="1.0" encoding="utf-8" ?>
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        header, footer {
        padding: 10pt;
        background-color: #333;
        color: #EEE;
        border-bottom: 1px solid black;
        border-top: 1px solid black;
        }

        body .content {
        margin: 20pt;
        font-size:12pt;
        padding: 4pt;
        border: solid 1px silver;
        }

        .small-page{
        page: initial;
        }

        .big-page{
        page: landscape;
        }

        /* This will be the default initial size */
        @page {
        size: A4 landscape;
        }

        /* any new pages with the page:landscape will
        use this size */

        @page landscape {
        size: A3 landscape;
        }

    </style>
</head>
<body>
    <header>
        <h4>This is the header</h4>
    </header>
    <h1>This is the content</h1>

    <!-- section that does not force a new page (so that it stays on the first page --
↪>
    <section class='content' style="page-break-before:avoid">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas scelerisque␣
↪porttitor urna.
```

(continues on next page)

(continued from previous page)

```
        Duis pellentesque sem tempus magna faucibus, quis lobortis magna aliquam.␣
↪Nullam eu risus

        <!-- Truncated for brevity -->

        Praesent mollis tempor enim.
    </section>

    <!-- This will be on the A3 landscape page -->
    <section class='content big-page'>
        Nullam et erat vel nisl suscipit volutpat id vitae massa. Nunc volutpat␣
↪feugiat iaculis.
        Mauris sit amet eleifend augue. Nulla imperdiet eu mauris nec consequat.␣
↪Donec a urna blandit,

        <!-- Truncated for brevity -->

        Mauris laoreet urna est, et tristique velit lobortis eu.
    </section>

    <!-- The inner div of small-page will revert the size back to the default␣
↪(initial) size  -->
    <div class="content">

        The inner content will be on a new page.

        <div class='content small-page' style="page-break-before:always;">
            Phasellus luctus dapibus nisi, et pulvinar neque ultrices vitae.␣
↪Pellentesque quis purus felis.
            Aliquam feugiat efficitur sem quis placerat. Quisque viverra magna vitae␣
↪elit eleifend, a porttitor
            enim vulputate. Quisque elit metus, aliquam eget purus at, blandit␣
↪gravida diam.

            <!-- Truncated for brevity -->

            venenatis ut ex. Donec euismod risus eros, dapibus tincidunt dolor varius␣
↪id.
        </div>
        After the content.
    </div>
    <footer>
        <h4>This is the footer</h4>
    </footer>

</body>

</html>
```



10.39.5 Stopping overflow

If overflowing onto a new page is not required or wanted then the page-break-inside='avoid' will block any overflow or new pages.

A section can be a single page, and never overflow.

10.40 Embeds and iFrames - TD

Preamble

10.40.1 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

See the tables_reference for more details on what is supported in tables.

10.40.2 Binding Simple numbers

10.40.3 Binding Dates and Times

10.40.4 Calculating values

10.40.5 Building in code

10.41 Templates and repeating content - TD

10.41.1 All the operators

10.41.2 Comparison functions

10.41.3 Math Functions

10.41.4 String Functions

10.41.5 Date Functions

10.41.6 Adding your own

10.41.7 Contributing

10.42 Pre, code and text wrapping - TD

10.42.1 All the operators

10.42.2 Comparison functions

10.42.3 Math Functions

10.42.4 String Functions

10.42.5 Date Functions

10.42.6 Adding your own

10.42.7 Contributing

10.43 Styles in your template - PD

In scryber styles are used through out to build the document. Every component has a base style and styles (such as fill colour and font) that flow down to their inner contents.

10.43.1 Styles on elements

```
<div style='margin:20pt;padding:4pt; background-color:#FF0000; color:#FFFFFF; font-
↪family: Arial, sans-serif; font-size:20pt' >
    <span>Hello World, from scryber.</span>
</div>
```

Or if you are dynamically generating some content in the code

```
private static Component StyledComponent()
{
    var div = new Div()
    {
        BackgroundColor = new Scryber.Drawing.PDFColor(Drawing.ColorSpace.RGB, 255, 0,
↪ 0),
        Margins = new Drawing.PDFThickness(20),
        Padding = new Drawing.PDFThickness(4),
        FontFamily = "Arial",
        FontSize = 20,
        FillColor = Scryber.Drawing.PDFColors.White
    };

    div.Contents.Add(new Label()
    {
        Text = "Hello World from scryber"
    });

    return div;

}
```

10.43.2 Style Classes

Along with appling styles directly to the components, Scryber supports the use of styles declaratively and applied to the content dynamically.

```
<html xmlns='http://www.w3.org/1999/xhtml'>
    <head>
        <style>
            div {
                font: Arial 20pt;
            }
            .mystyle {
                background-color:#FF0000;
                color:white;
                padding:20pt;
                margin:20pt;
            }
        </style>
    </head>
    <body>
        <div class="mystyle">
            <span>Hello World, from scryber.</span>
        </div>
```

(continues on next page)

(continued from previous page)

```
    </body>
</html>
```

By using styles, it cleans the code and makes it easier to standardise and change later on. This can either be within the document itself, or in a separate link files (see: referencing_files)

10.43.3 The :root selector

10.43.4 Block Styles

Components such as div's, paragraphs, headings, tables, lists and list items are by default blocks. This means they will begin on a new line. Components such as spans, labels, dates and numbers are inline components. This means they will continue with the flow of content in the current line.

There are certain style attributes that will only be used on block level components. These are:

- Background Styles
- Border Styles
- Margins
- Padding
- Vertical and Horizontal alignment.

Scryber does not (currently) support inline-blocks with their associated styles, but it is in the backlog.

10.43.5 Applying Styles

Just as in css and html, styles can be applied to an element based upon (multiple) combination(s) of 3 attributes of the Style.

id class type

e.g.

```
 <style>

 /* This style will be applied at the document level specifying
 the base level font, size and color for text. Because These
 cascade down, then it will be inherited by components in the document. */

html {
    font-family: "Gill Sans", sans-serif;
    font-size: 14pt;
    color: #333;
}

/* This style will be applied to the body tag for the first (set of) pages. */

body {
    margin: 10px;
}

/* This style will be applied to all top level headings
```

(continues on next page)

(continued from previous page)

```
 specifying the font size and some spacing */

h1 {
    font-weight: bold;
    font-size: 30pt;
    margin-top: 20pt;
    padding: 5pt;
}

/* This style will be applied to all top level headings with a class of 'warning'
 and give a background colour of red on white text.  */

.warning {
    background-color: #FF0000;
    color: #FFFFFF;
}

/* This style will be applied to all components with a class of 'border'
 and give a background colour of red with white text */

.border {
    border-color: #777;
    border-width: 1pt;
    border-style: Solid;
    color: #444;
}

/* This style will be applied to all H1 Headings with a class of 'border'
 and give a border colour of red with white text. It has a higher precedence than␣
↪either h1 or .border */

h1.border {
    border-color: #550000;
    color: white;
}

/* This style will only be applied to a component with ID 'FirstHead'
 and give a font size of 48pt */

#FirstHead {
    font-size: 36pt;
    font-weight: 400;
}

 </style>
```

Note: Currently scryber does not support the concept of pseudo-classes such as :hover or :first as css e.g. div.class:first. Nor does it support !important. It may be supported in the future.

The same styles can also be applied in the code of the document styles

10.43.6 Applying Multiple Styles

Every component supports the 'class' attribute. And the value of this can be one or more class names.

```
<h1 id="FirstHead" class="warning border" style="font-italic:true" >Hello World, from␣
↪scryber</h1>
```

This will apply the h1 style, the 2 classes for the warning and border, and the h1.border applied in that precedence order and increase the size based on the ID of FirstHead. And then the inline italic style will be applied.

10.43.7 Late adding of styles

Even once you have parsed or built a document, the styles can still be modified or added to. Either on a component, or at a document level, as they are evaluated, allowing runtime alteration of the output.

```
//change the style sheet based on a flag check
var sheet = checkflag ? "Sheet1.css" : "Sheet2.css"

using(var doc = PDFDocument.ParseDocument("MyPath.html") as HTMLDocument)
{
    //Load the stylesheet as a referenced component
    var link = new HtmlLink(){ Href = sheet };

    //and add it to the document styles.
    doc.Head.Contents.Add(link);

    //or explicitly define a style on the document
    var defn = new StyleDefn("h1.border");
    defn.Background.Color = (PDFColor)"#FFA";
    defn.Border.Width = 2;
    defn.Border.Color = PDFColors.Red;
    defn.Border.LineStyle = LineType.Solid;

    doc.Styles.Add(defn);
}
```

10.43.8 Data binding Styles

The process of data-binding (see: document_lifecycle, and document_databinding) can apply values to styles and classes on tags.

e.g.

```
<style>

html {
    font-family: "Gill Sans", sans-serif;
    font-size: 14pt;
    color: #333;
}
```

(continues on next page)

(continued from previous page)

```
body {
    margin: 10px;
}

/* this style will be applied as the bound class in the model */

.border {
    border-color: #777;
    border-width: 1pt;
    border-style: Solid;
    color: #444;
}

</style>
<body>
    <!-- apply a theme.headclass and explicit styles -->
    <div class='{@:model.theme.headclass}' style="{@:model.theme.bg}" >

        <!-- dynamic styles for the title and number -->
        <span style="{@:model.theme.title}" >This is the title</span><br/>
        <span style="{@:model.theme.number}" >1</span>
    </div>

</body>
```

Here the theme div and spans will pick up the default theme values. Were the code can provide new style colours and fonts for output.

```
var doc = PDFDocument.ParseDocument(path);
doc.Params["model"] = new {
   theme = new {
       headclass="border",
       bg = "background-color:#FFA;padding:20pt;border:solid 1px red;",
       title = "font-family:\"Times New Roman\", Times, serif;",
       number = "font-style: italic"
    }
};

return this.PDF(doc);
```

4_styles/images/helloworldpage_stylebound.png

10.43.9 Order and Precedence

Scryber tries to apply a priority, just as html to styles as they are loaded. This is based on order, depth and explicit.

div.class has a higher priority than .class

Explicit will be highest priority

And it will always fall back to the default (e.g. blue underline for anchor links).

Note: Scryber does not support !important overrides, nor does it support the use of :first-child, :hover or other pseudo classes.

Scryber has the same precedence order as html - based on the order in the document.

1. The inherited style from the parent is collected.
2. **Any styles in the document are evaluated in the order they appear.**
    1. What is the precedence of the matcher. Tag < Class < ID.
    2. What is the complexity of the match. Tag+Class < Tag+ID < Tag+Class+ID
    3. And parent selectors are evaluated to precedence Child < Parent(s) + Child
3. If a stylesheet reference is encountered, then the styles within it will be evaluated before moving on to the following styles
4. Finally the styles directly applied will be evaluated, giving the full style result.

This will then be flattened as a complete style and used in the layout and rendering of the component.

10.43.10 Supported CSS

The following CSS standard tags are supported...

- **border**
    - border-width
    - border-style
    - border-color
    - **border-top**
        - border-top-width
        - border-top-color
        - border-top-style
    - **border-left**
        - border-left-width
        - border-left-color
        - border-left-style
    - **border-right**
        - border-right-width
        - border-right-color
        - border-right-style
    - **border-bottom**
        - border-bottom-width
        - border-bottom-color

        * border-bottom-style
- color
- **background**
    - background-image
    - background-color
    - background-repeat
    - background-size
    - background-position
- **font**
    - font-style
    - font-weight - Translated to regular and bold (for the moment)
    - font-size
    - font-family
    - line-height
- **margin**
    - margin-left
    - margin-right
    - margin-top
    - margin-bottom
- **padding**
    - padding-left
    - padding-right
    - padding-top
    - padding-bottom
- opacity
- fill-opacity
- column-count
- column-gap
- column-span (for table cells)
- page-break-inside
- page-break-after
- page-break-before
- left
- top
- width
- height

- min-width
- min-height
- max-width
- max-height
- text-align
- vertical-align
- **display**
  - inline
  - block
  - none
- **overflow**
  - visible, auto
  - hidden
- **position**
  - relative
  - absolute
  - static
- text-decoration
- letter-spacing
- word-spacing
- white-space
- **list-style-type (and list-style which is treated as equivalent)**
  - bullet, disc
  - decimal
  - lower-roman
  - lower-alpha
  - upper-roman
  - upper-alpha
  - none
- **stroke**
  - stroke-opacity
  - stroke-width
- **size**
  - A4, A3, Letter, etc.
  - portrait or landscape
- **page**

- explicit name (of an @page style)

10.43.11 at-rules supported

The following at-rules are supported

- @media - including or excluding css based on print.
- @font-face - using explicit font files and names.
- @page - specifying page sizes for sections and breaks.

10.43.12 Styles in code

10.44 Column layout - PD

Scryber supports flowing content layout. No matter the font, content type or structure. (see document_pages) It also supports the same with columns, at the page and container level.

10.44.1 Specifying columns on Pages

All block level components (see component_positioning) support the use of columns.

There is by default on a block a single column, but by specifying a style:column-count (either on the block, or in the style definition) then the layout will split the block into that number of regions within it.

Content within the column will flow down as far as it is able (either the bottom or the page, or the maximum height of the container) and then move to the top of the next column.

Below we specify the section to have 3 columns, they will be of equal size and the content within it will flow onto each column and then ulimately the next page.

```
<html xmlns='http://www.w3.org/1999/xhtml'>
<!-- set the coumn count on the body -->
<body style="column-count:3;">
    <header>
        <h4 style='margin: 5pt; border-bottom-width: 1pt; border-bottom-color:aqua'>
↪This is the header</h4>
    </header>

    <h1 style='margin:5pt;border-width:1pt; border-color:green'>This is the content</
↪h1>

    <div style='margin:5pt; font-size:14pt; border-width: 1pt; border-color: navy'>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer quis orci␣
↪mollis, finibus eros a,
        tincidunt magna. Mauris efficitur nisl lorem, vitae semper nulla convallis id.
↪ Nam dignissim rutrum
        mollis. Fusce imperdiet fringilla augue non venenatis. Mauris dictum velit␣
↪augue, ut iaculis risus
        pulvinar vitae. Aliquam id pretium sem. Pellentesque vel tellus risus. Etiam␣
↪dolor neque, auctor id
        convallis hendrerit, tincidunt at sem. Integer finibus congue turpis eu␣
↪feugiat. Nullam non ultrices enim.<br />
        <!-- Truncated for brevity
```

(continues on next page)

(continued from previous page)

```
        .
        . -->
        Maecenas vitae vehicula mauris. Aenean egestas et neque sit amet pulvinar.
        Phasellus ultrices congue semper. Praesent ultrices orci ipsum. Maecenas␣
↪suscipit tellus elit,
        non ullamcorper nulla blandit sed. Nulla eget gravida turpis, et vestibulum␣
↪nunc. Nulla mollis
        dui eu ipsum dapibus, vel efficitur lectus aliquam. Nullam efficitur, dui a␣
↪maximus ullamcorper,
        quam nisi imperdiet sapien, ac venenatis diam lectus a metus. Fusce in lorem␣
↪viverra, suscipit
        dui et, laoreet metus. Quisque maximus libero sed libero semper porttitor. Ut␣
↪tincidunt venenatis
        ligula at viverra. Phasellus bibendum egestas nibh ac consequat. Phasellus␣
↪quis ante eu leo tempor
        maximus efficitur quis velit. Phasellus et ante eget ex feugiat finibus␣
↪ullamcorper ut nisl. Sed mi
        nunc, blandit ut sem vitae, bibendum hendrerit ipsum.<br />
    </div>
    <h1 style="margin:5pt; border-width:1pt; border-color: green">After the content</
↪h1>

    <footer>
        <h4 style="margin: 5pt; border-width: 1pt; border-color: purple">This is the␣
↪footer</h4>
    </footer>
</body>

</html>
```

Here the page is set to 3 columns across the layout page. The headers are independent of the column setting, but the inner content flows down the first column, and then moves to the next when no more can be fitted, and so on until it reaches the end of the layout page.

As we can overflow, a new layout page is added, and the content latout continues until the end.

Note: The borders and any background will be based around the container and show on each column.

10.44.2 Columns on containers

As we can see above, the headings were also part of the column layout on the page.

Scryber supports the use of columns on containers too. So our layout can be improved if we remove the columns from the page, and set them on the *div* itself.

This will allow the headers to be full width, with the content flowing within the columns of the container.

```
<html xmlns='http://www.w3.org/1999/xhtml'>
<!-- remove the coumn count on the body -->
<body>
    <header>
        <h4 style='margin: 5pt; border-bottom-width: 1pt; border-bottom-color:aqua'>
↪This is the header</h4>
    </header>
    <h1 style='margin:5pt;border-width:1pt; border-color:green'>This is the content</
↪h1>

    <!-- Set the div to have 3 columns
        rather than the page -->

    <div style='column-count:3; margin:5pt; font-size:14pt; border-width: 1pt; border-
↪color: navy'>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer quis orci␣
↪mollis, finibus eros a,
        tincidunt magna. Mauris efficitur nisl lorem, vitae semper nulla convallis id.
↪ Nam dignissim rutrum
        mollis. Fusce imperdiet fringilla augue non venenatis. Mauris dictum velit␣
↪augue, ut iaculis risus
        pulvinar vitae. Aliquam id pretium sem. Pellentesque vel tellus risus. Etiam␣
↪dolor neque, auctor id
        convallis hendrerit, tincidunt at sem. Integer finibus congue turpis eu␣
↪feugiat. Nullam non ultrices enim.<br />
        <!-- Truncated for brevity
        .
        . -->
        Maecenas vitae vehicula mauris. Aenean egestas et neque sit amet pulvinar.
        Phasellus ultrices congue semper. Praesent ultrices orci ipsum. Maecenas␣
↪suscipit tellus elit,
        non ullamcorper nulla blandit sed. Nulla eget gravida turpis, et vestibulum␣
↪nunc. Nulla mollis
        dui eu ipsum dapibus, vel efficitur lectus aliquam. Nullam efficitur, dui a␣
↪maximus ullamcorper,
        quam nisi imperdiet sapien, ac venenatis diam lectus a metus. Fusce in lorem␣
↪viverra, suscipit
        dui et, laoreet metus. Quisque maximus libero sed libero semper porttitor. Ut␣
↪tincidunt venenatis
        ligula at viverra. Phasellus bibendum egestas nibh ac consequat. Phasellus␣
↪quis ante eu leo tempor
        maximus efficitur quis velit. Phasellus et ante eget ex feugiat finibus␣
↪ullamcorper ut nisl. Sed mi
        nunc, blandit ut sem vitae, bibendum hendrerit ipsum.<br />
    </div>
    <!-- after the columns we go full width again -->
    <h1 style="margin:5pt; border-width:1pt; border-color: green">After the content</
↪h1>

    <footer>
        <h4 style="margin: 5pt; border-width: 1pt; border-color: purple">This is the␣
↪footer</h4>
    </footer>
</body>
```

10.44.3 Column breaks and keeping content together

Just as with pages, columns support both breaking before and after along with the flowing layout.

Using the 'break-before: always' style a new column (or page if we are already on the last columns) will be moved to for the layout of content. Using 'break-after: always' will force the following content onto a new column or page.

If there is a block of content that should stay together, the the 'break-inside: avoid' style will attempt to keep all the inner content (blocks, images, text etc), on a single page.

```
<html xmlns='http://www.w3.org/1999/xhtml'>

<body>
    <header>
        <h4 style='margin: 5pt; border-width: 1pt; border-color:aqua'>This is the␣
↪header</h4>
    </header>

    <h1 style='margin:5pt;border-width:1pt; border-color:green'>This is the content</
↪h1>

    <!-- 3 columns with inner block content -->

    <div style='column-count:3; margin:5pt; font-size:14pt; border-width: 1pt; border-
↪color: navy'>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer quis orci␣
↪mollis, finibus eros a,
        <!-- Truncated for brevity -->
        convallis hendrerit, tincidunt at sem. Integer finibus congue turpis eu␣
↪feugiat. Nullam non ultrices enim.
        <br />
        <br />
        <!-- This will always be on a new column -->
        <div style="break-before: always;">
            Sed commodo metus id erat accumsan, quis feugiat augue vestibulum.␣
↪Praesent porta sit amet erat a hendrerit.
            <!-- Truncated for brevity -->
            Quisque tincidunt nec quam sollicitudin consequat.
        </div>
        <div>
            Donec mattis eros non nibh mattis, in volutpat mi pretium. Donec mattis␣
↪iaculis neque a accumsan.
            <!-- Truncated for brevity -->
            vitae sem bibendum laoreet.
        </div>

        <div style="break-inside: avoid" >
            Class aptent taciti sociosqu ad litora torquent per conubia nostra, per␣
↪inceptos himenaeos. Nam viverra laoreet
            <!-- Truncated for brevity -->
```

(continues on next page)

(continued from previous page)

```
            Nulla viverra sagittis leo.
        </div>

        <!-- this content will be kept together and then a column break will be put␣
↪after it. -->
        <div style="break-inside: avoid; break-after: always" >
            Maecenas vitae vehicula mauris. Aenean egestas et neque sit amet pulvinar.
            <!-- Truncated for brevity -->
            nunc, blandit ut sem vitae, bibendum hendrerit ipsum.
        </div>
        <br />
        <br/>
        <div style="break-inside: avoid;" >
        Phasellus congue commodo elit, ac tincidunt ex placerat vitae. Lorem ipsum␣
↪dolor sit amet, consectetur adipiscing
        <!-- Truncated for brevity -->
        Ut sagittis sed mi nec tempus. Pellentesque et consectetur lorem, eget␣
↪sagittis nibh. Cras vehicula ligula est.
        Nulla viverra sagittis leo.
    </div>
    </div>
    <h1 style="margin:5pt; border-width:1pt; border-color: green">After the content</
↪h1>

    <footer>
        <h4 style="margin: 5pt; border-width: 1pt; border-color: purple">This is the␣
↪footer</h4>
    </footer>
</body>

</html>
```

4_styles/images/documentcolumns6.png

10.44.4 Column-width and column-gap

Rather than specifying the number of columns, scryber also supports the standard html column-width option.

This makes the width value the predominant driver, and will layout the maximum number of columns that are at least this width within the available space, so that it is full width.

e.g. if you have a container that is 300pts wide and a column-width of 80pt, then there will be 3 columns of about 92pts wide (assuming the alley / gap is the default 10pt). Increasing the column-width to 120pt, and the number of columns will reduce to 2 of around 145pts.

If our page size or orientation changes then the number of columns fitted changes.

Column-gaps are the margins, or alleys, between each column. The default is 10pt, but it can be specified as a single unit value, e.g. 20pt or 5mm (see drawing_units for more on scryber measurements).

```
<?xml version="1.0" encoding="utf-8" ?>
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
         "http://www.w3.org/TR/html4/strict.dtd">

   <html xmlns='http://www.w3.org/1999/xhtml'>
   <head>
       <style type="text/css" >

           @page {
           size: A4 Landscape;
           }

       </style>
   </head>
   <body>
       <header>
           <h4 style='margin: 5pt; border-bottom-width: 1pt; border-bottom-color:aqua
↪'>This is the header</h4>
       </header>
       <h1 style='margin:5pt;border-width:1pt; border-color:green'>This is the content
↪</h1>
       <!-- Set a section to not break on the first page -->
       <div style='column-width: 150pt; column-gap:60pt; margin:5pt; font-size: 11pt;␣
↪border-width: 1pt; border-color: navy'>
           Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer quis orci␣
↪mollis, finibus eros a,
           tincidunt magna. Mauris efficitur nisl lorem, vitae semper nulla convallis␣
↪id. Nam dignissim rutrum
           ......
           <div style="break-before:always;" >
               In ac diam sapien. Morbi viverra ante non lectus venenatis posuere.␣
↪Curabitur porttitor viverra augue sit amet
               convallis. Duis hendrerit suscipit vestibulum. Fusce fringilla␣
↪convallis eros, in vehicula nibh tempor sed.
               ........
           </div>
           <div style="break-inside:avoid">
               Duis et tincidunt nisi. Etiam sed augue a turpis semper cursus. Proin␣
↪facilisis feugiat risus, in malesuada
               lectus posuere eget. Nullam ultricies velit purus, vel lobortis felis␣
↪commodo nec. Nam bibendum eleifend blandit.
               Vestibulum et turpis a metus euismod euismod nec sed nulla. Aliquam␣
↪iaculis, magna in posuere finibus, turpis
               .......
           </div>
           <br/>
           In ac diam sapien. Morbi viverra ante non lectus venenatis posuere.␣
↪Curabitur porttitor viverra augue sit amet
           convallis. Duis hendrerit suscipit vestibulum. Fusce fringilla convallis␣
↪eros, in vehicula nibh tempor sed.
           Fusce gravida, orci eget venenatis hendrerit, augue erat euismod magna,␣
↪nec interdum eros dolor sed ipsum.
           .....
           maximus efficitur quis velit. Phasellus et ante eget ex feugiat finibus␣
↪ullamcorper ut nisl. Sed mi
           nunc, blandit ut sem vitae, bibendum hendrerit ipsum.
       </div>
```

(continues on next page)

(continued from previous page)

```
        <h1 style="margin:5pt; border-width:1pt; border-color: green">After the content
↪</h1>

        <footer>
            <h4 style="margin: 5pt; border-width: 1pt; border-color: purple">This is␣
↪the footer</h4>
        </footer>
    </body>

    </html>
```

Here we can see that we have changed the paper orientation to landscape, set the column width to 150pt with gap of 60pt. The layout engine adjusts all content automatically within the column widths.

The second div will always be on a new column, and the 3rd div moves to a new column as it cannot fit. And the rest of the layout continues on the 3rd column until it reaches the end, and will flow onto another page.

Note: As can be seen in the above image, scryber does not balance columns across the page (matching height). We may look to support this, but the min-height, max-height and breaks can be used to maintain the structure.

10.44.5 Images and Shapes in columns

As with component_sizing, images and shapes that do not have an explicit size, take their natural width up to the size of the container.

This also applies to columns. If an image is too wide for the column it will be proportionally resized to fit within the column. Any content can be placed in a column.

10.44.6 Nested containers and columns

Scryber fully supports nested columns whether that be at the page or multiple container level. Again mixed content can be used within the columns, and the content will flow as normal.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css" >

        @page {
        size: A4 Landscape;
        }
```

(continues on next page)

(continued from previous page)

```
    </style>
</head>
<body>
    <header>
        <h4 style='margin: 5pt; border-bottom-width: 1pt; border-bottom-color:aqua'>
↪This is the header</h4>
    </header>
    <h1 style='margin:5pt;border-width:1pt; border-color:green'>This is the content</
↪h1>
    <!-- Set a section to not break on the first page -->
    <div style="column-count: 3; column-gap:20pt; font-size:12px; padding:10pt;">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer quis orci␣
↪mollis, finibus eros a,
        tincidunt magna. Mauris efficitur nisl lorem, vitae semper nulla convallis id.
↪ Nam dignissim rutrum
        .....
        <div style="column-count:2; border:solid 1px red; padding:10pt; margin:5pt␣
↪0pt;">
            <img src="./images/group.png" style="break-after:always;" />
            In ac diam sapien. Morbi viverra ante non lectus venenatis posuere.␣
↪Curabitur porttitor viverra augue sit amet
            convallis. Duis hendrerit suscipit vestibulum. Fusce fringilla convallis␣
↪eros, in vehicula nibh tempor sed.
            Fusce gravida, orci eget venenatis hendrerit, augue erat euismod magna,␣
↪nec interdum eros dolor sed ipsum.

        </div>
        Duis et tincidunt nisi. Etiam sed augue a turpis semper cursus. Proin␣
↪facilisis feugiat risus, in malesuada
        lectus posuere eget. Nullam ultricies velit purus, vel lobortis felis commodo␣
↪nec. Nam bibendum eleifend blandit.
        ......
        elit maximus. Suspendisse non ultricies mi. Integer efficitur sapien lectus,␣
↪non laoreet tellus dictum vel.
        Maecenas vitae vehicula mauris. Aenean egestas et neque sit amet pulvinar.<br␣
↪/>
        <img src="./images/group.png" style="margin:5pt;" />
        Maecenas vitae vehicula mauris. Aenean egestas et neque sit amet pulvinar.
        .......
        nunc, blandit ut sem vitae, bibendum hendrerit ipsum.
    </div>
    <h1 style="margin:5pt; border-width:1pt; border-color: green">After the content</
↪h1>

    <footer>
        <h4 style="margin: 5pt; border-width: 1pt; border-color: purple">This is the␣
↪footer</h4>
    </footer>
</body>

</html>
```

10.45 Textual Layout - PD

FIGURE

10.45.1 Overview

10.45.2 Generation methods

All methods and files in these samples use the standard testing set up as outlined in ../overview/samples_reference

10.45.3 Text Horizontal Alignment

Scryber supports the alignment of text at a block (rather than line level)

- Left
- Right
- Center
- Justified

The value is inherited so that child components will be aligned in the same way, unless explicitly set differently.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Document Text H Align</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font {
            font-size: 14pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
            font-family: sans-serif;
        }

    </style>
</head>
<body style="padding: 20pt">
    <div class="std-font" style="text-align:left;">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc pellentesque␣
↪turpis ac pellentesque scelerisque.
```

(continues on next page)

(continued from previous page)

```
        Etiam at nibh mattis, pulvinar velit eget, consequat ligula.
        Aenean sit amet nibh urna. Praesent odio magna, pharetra a posuere non,␣
↪dignissim non lectus.
        Maecenas cursus porttitor sem vitae posuere. Phasellus quis lorem sapien.␣
↪Aenean dictum pretium rutrum.
    </div>

    <div class="std-font" style="text-align:right">
        Proin id blandit ante, at pellentesque nulla. Fusce viverra, nibh eu␣
↪sollicitudin euismod.
        Praesent aliquam gravida scelerisque. Pellentesque ac ante eu augue lacinia␣
↪blandit nec vitae tellus.
        <div>Inner content</div>
        Nullam lacus dolor, mollis et orci vitae, ornare euismod turpis. Mauris␣
↪tempus at orci id bibendum.
        Integer et aliquet velit. Proin eget ullamcorper libero. Sed bibendum mattis␣
↪sem. Mauris in purus leo.
    </div>

    <div class="std-font" style="text-align:center">
        Phasellus dignissim risus vel nisi pellentesque dapibus. Vivamus at eros␣
↪finibus, cursus mi eget, viverra elit.
        Integer non felis eget justo mollis aliquam. Donec sed pharetra odio.
        <div style="text-align:left">Left inner content</div>
        Fusce pulvinar elit leo, sit amet egestas neque porttitor nec.
        Nunc ac varius augue, ac rhoncus orci. Integer sit amet porta erat, vel␣
↪scelerisque augue.
    </div>

    <div class="std-font" style="text-align:justify">
        Sed quis nibh libero. Vivamus euismod metus vel purus tristique, vitae␣
↪gravida massa pretium.
        Proin facilisis arcu fringilla diam malesuada dictum.
        Praesent vel viverra nibh. Donec rhoncus nisl fermentum ante auctor␣
↪consectetur.
        Proin posuere orci sed justo placerat elementum. Praesent cursus ullamcorper␣
↪leo.
        Etiam ut massa lectus. Nunc dapibus tempus velit id tincidunt. Phasellus␣
↪cursus finibus commodo.
    </div>
</body>
</html>
```

10.45.4 Text Vertical Alignment

The vertical alignment in text is also based on the container, and supports the following values.

- top

- middle
- bottom

Note: The size of a block is normally shrunk to the size of the content, so vertical alignment has no visible effect. So a height must be set on the container.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Document Text H Align</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font {
            font-size: 14pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
            font-family: 'Hiragino Mincho', sans-serif;
        }

    </style>
</head>
<body style="padding: 20pt">

    <div class="std-font" style="vertical-align:top; height: 150pt">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc pellentesque␣
↪turpis ac pellentesque scelerisque.
        Etiam at nibh mattis, pulvinar velit eget, consequat ligula.
        Aenean sit amet nibh urna. Praesent odio magna, pharetra a posuere non,␣
↪dignissim non lectus.
        Maecenas cursus porttitor sem vitae posuere. Phasellus quis lorem sapien.␣
↪Aenean dictum pretium rutrum.
    </div>

    <div class="std-font" style="vertical-align:middle; height: 150pt">
        Proin id blandit ante, at pellentesque nulla. Fusce viverra, nibh eu␣
↪sollicitudin euismod.
        Praesent aliquam gravida scelerisque. Pellentesque ac ante eu augue lacinia␣
↪blandit nec vitae tellus.
        Nullam lacus dolor, mollis et orci vitae, ornare euismod turpis. Mauris␣
↪tempus at orci id bibendum.
        Integer et aliquet velit. Proin eget ullamcorper libero. Sed bibendum mattis␣
↪sem. Mauris in purus leo.
    </div>

    <div class="std-font" style="vertical-align: bottom; height: 150pt">
        Phasellus dignissim risus vel nisi pellentesque dapibus. Vivamus at eros␣
↪finibus, cursus mi eget, viverra elit.
        Integer non felis eget justo mollis aliquam. Donec sed pharetra odio.
        Fusce pulvinar elit leo, sit amet egestas neque porttitor nec.
        Nunc ac varius augue, ac rhoncus orci. Integer sit amet porta erat, vel␣
↪scelerisque augue.
```

(continues on next page)

(continued from previous page)

```
    </div>


</body>
</html>
```

```
4_styles/images/documentTextValign.png
```

10.45.5 Differences to HTML rendering

As mentioned the vertical and horizontal alignment are declared at the container level and apply to all content within.

This is in contrast to HTML that will genrally align on the element level and flow down. It is usually easy to replicate behaviour and visual style on both.

10.45.6 Alignment in code

10.45.7 Next Steps

10.46 Aligning your content - PD

Alignment of components in pages, containers and tables is fully supported, as is text alignment.

Unlike html, the alignment is set on the container, rather than the element. So if you set left align on a div, all the content within the div will be aligned to the left.

Also unlike html, vertical alignment is fully supported, without hacks or fixes. This is because we have a known height of a page or container.

10.46.1 Horizontal Allignment

The alignment of content within a page or container can either be set as the standard values:

- Left (default)
- Right
- Center
- Justified

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
              xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Styles.xsd"
              xmlns:data="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Data.
↪xsd" >
```

(continues on next page)

(continued from previous page)

```
<Styles>
    <styles:Style applied-type="doc:H1" >
        <styles:Position h-align="Center"/>
        <styles:Border color="green" width="1pt"/>
        <styles:Padding all="5pt"/>
    </styles:Style>

    <styles:Style applied-type="doc:Div" >
        <styles:Border color="red" width="1pt" />
        <styles:Padding all="5pt"/>
        <styles:Margins top="10pt"/>
    </styles:Style>
</Styles>
<Pages>
    <doc:Section >
        <Content>
            <doc:H1 >This is the Centered Heading</doc:H1>
            <doc:Div>
                Default alignment.<doc:Br/>
                With mixed content.
                <doc:Image src="../../content/images/group.png" styles:width="50pt" />
            </doc:Div>
            <doc:Div styles:h-align="Right">
                Right alignment.<doc:Br/>
                With mixed content.
                <doc:Image src="../../content/images/group.png" styles:width="50pt" />
            </doc:Div>
            <doc:Div styles:h-align="Center">
                Center alignment.<doc:Br/>
                With mixed content.
                <doc:Image src="../../content/images/group.png" styles:width="50pt" />
            </doc:Div>
            <doc:Div styles:h-align="Justified">
                This is justified alignment across multiple lines in the container␣
↪that
                will stretch to the width with word and character spacing, with mixed␣
↪content.
                <doc:Image src="../../content/images/group.png" styles:width="50pt" />
            </doc:Div>
            <doc:H1>After the content</doc:H1>
        </Content>
    </doc:Section>
</Pages>

</doc:Document>
```

10.46.2 Vertical alignment

Along with horizontal alignment vertical alignment can be applied to the page or containers.

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
              xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Styles.xsd"
              xmlns:data="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Data.
↪xsd" >
<Styles>
    <styles:Style applied-type="doc:H1" >
        <styles:Position h-align="Center"/>
        <styles:Border color="green" width="1pt"/>
        <styles:Padding all="5pt"/>
    </styles:Style>

    <styles:Style applied-type="doc:Div" >
        <styles:Border color="red" width="1pt" />
        <styles:Padding all="5pt"/>
        <styles:Margins top="10pt"/>
    </styles:Style>
</Styles>
<Pages>

    <doc:Section styles:v-align="Bottom" >
    <Content>
        <doc:H1 >Page has a bottom vertical alignment</doc:H1>
        <doc:Div styles:height="200pt">
            Default top left alignment.
            <doc:Image src="../../content/images/group.png" styles:width="50pt" />
        </doc:Div>
        <doc:Div styles:h-align="Right" styles:v-align="Bottom" styles:height="200pt">
            Right Bottom alignment with mixed content.
            <doc:Image src="../../content/images/group.png" styles:width="50pt" />
        </doc:Div>
        <doc:Div styles:h-align="Center" styles:v-align="Middle" styles:width="350pt"␣
↪styles:height="200pt">
            Center Middle alignment with fixed width and mixed content.
            <doc:Image src="../../content/images/group.png" styles:width="50pt" />
        </doc:Div>
    </Content>
    </doc:Section>

</Pages>

</doc:Document>
```

Note: With containers unless they have a specific height there gerenally will not be any vertical change, because they

shrink to their respective heights. However actual heights will support the alligment.

10.46.3 Nested Alignment in containers

Alignment applies to the containers individually. Nesting various alignment attributes on content within a page can be done, and also applied to positioned components.

So complex layouts can be achieved with minimal .

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
              xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Styles.xsd"
              xmlns:data="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Data.
↪xsd" >
<Styles>
    <styles:Style applied-type="doc:H1" >
        <styles:Position h-align="Center"/>
        <styles:Background color="#6666FF"/>
        <styles:Fill color="white"/>
        <styles:Font size="24pt" family="Segoe UI" bold="false" />
        <styles:Padding all="5pt"/>
    </styles:Style>

    <styles:Style applied-type="doc:Section" >
        <styles:Font family="Segoe UI Light"/>
    </styles:Style>

    <styles:Style applied-class="floating" >
        <styles:Position mode="Absolute" x="50pt" y="100pt" h-align="Center" v-align=
↪"Bottom"/>
        <styles:Size width="200pt" height="200pt"/>
        <styles:Background color="#6666FF" img-src="../../content/images/landscape.jpg
↪" repeat="Fill"/>
        <styles:Fill color="white"/>
    </styles:Style>

</Styles>
<Pages>
    <doc:Section styles:v-align="Bottom" styles:bg-color="#DDDDFF" >
    <Header>
        <doc:H1 styles:column-count="3" >
        Nested
        <doc:ColumnBreak/>
        Content
        <doc:ColumnBreak/>
        Alignment
        </doc:H1>
    </Header>
    <Content>
        <doc:Div styles:height="400pt" styles:h-align="Right" styles:bg-color="#AAAADD
↪">
        <doc:H1 styles:width="400pt" styles:h-align="Left">
            The alignment of a page content.
        </doc:H1>
```

(continues on next page)

(continued from previous page)

```
        </doc:Div>

        <doc:Div styles:class="floating">
        <doc:Div styles:bg-color="#6666FF" styles:font-size="16pt" styles:h-align=
↪"Center" styles:padding="5pt">
            Pushing to absolute.
        </doc:Div>
        </doc:Div>
    </Content>
    <Footer>
        <doc:H1 styles:column-count="3" styles:font-size="10pt" styles:height="40pt"␣
↪styles:v-align="Middle" >
            Accounts
            <doc:ColumnBreak/>
            For Columns
            <doc:ColumnBreak/>
            And Footers
        </doc:H1>
    </Footer>
    </doc:Section>
</Pages>

</doc:Document>
```

4_styles/images/documentnestedalign.png

10.47 Sizing your content - PD

Scryber has an intelligent layout engine. By default eveything will be laid out as per the flowing layout of the document Pages and columns. Each component, be it block level or inline will have a position next to its siblings and move and following content along in the document. If the content comes to the end of the page and cannot be fitted, then if allowed, it will be moved to the next page.

However it is very easy to size and position (see component_positioning) the content. All measusements are using the scryber unit and thickness (see drawing_units for more on the use of measurements and dimensions).

10.47.1 Width and Height

All block components support an explicit width and / or height value. If it's width is set, then any full-width style will be ignored.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
```

(continues on next page)

(continued from previous page)

```
    <style type="text/css" >

        .bordered {
            border: solid 1px #777;
            background-color: #EEE;
            padding:4pt;
        }

    </style>
</head>
<body>
    <div class="bordered">
        The content of this div is all as a block (by default)
    </div>

    <div class="bordered" style="width:300pt">
        The content of this div is set to 300pt <u>wide</u>, so the content will flow␣
↪within this width,
        and grow the height as needed.
    </div>

    <div class="bordered" style="height:150pt">
        The content of this div is set to 150pt <u>high</u>, so the content will flow␣
↪within this
        as full width, but the height will still be 150pt.
    </div>

    <div class="bordered" style="width:300pt; height:150pt">
        The content of this div is set to 300pt <u>wide</u> and 150pt <u>high</u>, so␣
↪the content will flow within this
        as full width, but the height will still be 150pt.
    </div>
</body>

</html>
```

10.47.2 Images with width and height

Scryber handles the sizing of images based on the natural size of the image. If no explicit size or positioning is provided, then it will be rendered at the native size for the image.

If the available space within the container is not sufficuent to hold the image at it's natural size, then the image render size will be reduced proportionally to fit the space available.

If either a width **or** height is assigned, then this will be used to proportionally resize the image to that height or width.

If both a width **and** height are assigned, then they will both be used to fit the image to that space. No matter what the originals' proportions are.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css" >

        body{ padding: 20pt;}

        .bordered { font-size:14pt; }

        .columns{ column-count: 4; }

        .columns div.bordered{ break-after:always; }

    </style>
</head>
<body>
    <div class="bordered" style="margin:30pt;">
        An image will natually size to it's dimensions without space restriction.
        <img src="./images/landscape.jpg" />
    </div>
    <div class="columns" style="column-count: 4">
        <div class="bordered">
            <b>First Column</b><br />
            An image will fit to it's container if no explicit size is set.
            <img src="./images/landscape.jpg" />
        </div>
        <div class="bordered">
            <b>Second Column</b><br />
            If a width is set, then the sizing will be proportional.
            <img src="./images/landscape.jpg" style="width:100pt;" />
        </div>

        <div class="bordered" >
            <b>Third Column</b><br />
            If a height is set, then the sizing will be proportional.
            <img src="./images/landscape.jpg" style="height:50pt;" />
        </div>

        <div class="bordered" >
            <b>Fourth Column</b><br />
            If a width and height are set these will be used explicitly.
            <img src="./images/landscape.jpg" style="width:100pt; height:50pt;" />
        </div>
    </div>

    <!-- Photo by Bailey Zindel on Unsplash -->
</body>

</html>
```

10.47.3 Margins and Padding

All block level elements support padding and margins. Unlike html, scryber does not count the width of the border as part of the box dimensions (on purpose).

Dimensions can be set either directly on the component, or on a style applied to the components (see: document_styles).

The *Margin* and *Padding* style have the 4 individual properties that can also be set.

- Top
- Right
- Bottom
- and Left

If an individual side property is set, then this will override any value set on all.

The margins or padding attributes on tags can alsp be set with 1, 2 or 4 values. If only one is provided it will be applied to each. If 4 are provided, they will be applied to each individual value in the *top*, *right*, *bottom*, *left* (as per html padding). If 2 are provided the first will be applied to the top and bottom, the second to the left and right.

Note: If any margins or padding attribute is set on the component, it will override ALL values set in any style.

If not set then the values will be zero.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        body {
            margin: 20pt;
            font-size:12pt;
        }

        .bordered {
            border-style: solid;
            border-width: 1pt;
            border-color: #777;
            background-color: #EEE;
        }

        .red {
            border-color: #F00;
```

(continues on next page)

(continued from previous page)

```
        }

        .spaced {
            margin: 20pt;
            margin-left: 10pt;
            margin-right: 10pt;
            padding: 5pt;
        }
    </style>
</head>
<body class="bordered">

    <b>First Example</b>
    <div class="bordered red">
        The content of this div has a red border with no padding or margins.
    </div>

    <b>Second Example</b>
    <div class="bordered red spaced">
        The content of this div has a red border with both margins and padding set␣
↪from the style.
    </div>

    <b>Third Example</b>
    <div class="bordered red spaced" style='padding:20pt;'>
        The content of this div has a red border with margins set from the style and␣
↪padding overridden explicitly on the component.
    </div>

    <b>Borders are supported on images and other blocks too, and will respect the␣
↪width and or height properties.</b>
    <img src="./Images/landscape.jpg" class="bordered spaced" style="width:100pt" />
    <h1 class="bordered spaced">Heading with spacing.</h1>

</body>
</html>
```

10.47.4 Minimum and Maximum size

Along with the use of width and height, scryber also supports the use of minimum height/width and maximum height/width.

As you might expect, the minimum will ensure that a container is at least as big as the specified value, and that the maximum will ensure the content, never grows beyond that specified value.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
```

(continues on next page)

(continued from previous page)

```
<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        body {
            margin: 20pt;
            font-size:12pt;
        }

        .bordered {
            border-style: solid;
            border-width: 1pt;
            border-color: #777;
            background-color: #EEE;
        }

        .red {
            border-color: #F00;
        }

        .spaced {
            margin: 20pt;
            margin-left: 10pt;
            margin-right: 10pt;
            padding: 5pt;
        }

        .sized{
            max-height:60pt;
            max-width:350pt;
        }
    </style>
</head>
<body class="bordered">
    <br />
    <b>Minimum Size, not reached</b>
    <div class="bordered red" style="min-height:60pt; min-width:350pt">
        This div has a red border with min size.
    </div>
    <br />
    <b>Minimum Size, width reached</b>
    <div class="bordered red" style="min-height:60pt; min-width:350pt">
        This div has a red border with min size, but the content will push this out␣
↪beyond the minimum width.
    </div>
    <br />
    <b>Minimum Size, width reached</b>
    <div class="bordered red" style="min-height:60pt; min-width: 350pt">
        This div has a red border with min size, but the content will push this out␣
↪beyond the minimum width to the
        space in the container, and then flow as normal.
    </div>
    <br />
    <b>Maximum Size, not reached</b>
    <div class="bordered red sized">
        This div has a red border with max size.
```

(continues on next page)

(continued from previous page)

```
    </div>
    <br />
    <b>Maximum Size, width reached</b>
    <div class="bordered red sized">
        This div has a red border with max size, and the content will flow as the max-
↪width is reached with the text.
    </div>

</body>
</html>
```

4_styles/images/documentsizingminmax.png

10.47.5 Full width blocks

The div component automatically fills the available width of the region. Even if the inner content does not need it. It's effectivly set as 100% width.

If an explicit width, or max-width or min-width, is applied, the block will honour these rather than stretch to full width. This applies to the page, or a column containing the block.

By default div's and paragraphs are set to full width. blockQuotes, tables and lists are not. If it is needed to set the width of one of these to expand to the full avaialble space, then the 100% width is supported.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<body style="margin:20pt; font-size:20pt">
    <div style="border:solid 1pt black; padding: 5pt">
        This div is full width<br />
        And will extend beyond the content.<br />
        To the width of its container.
    </div>
    <br />
    <div style="border:solid 1pt black; padding: 5pt; max-width:300pt;">
        This div is NOT full width<br />
        And will only size to the content.
    </div>
    <br />
    <div style="border:solid 1pt black; padding: 5pt; min-width:300pt;">
        This div is NOT full width,
        but will  size to the content available in the container,
        and then flow to the next line.
    </div>
    <br />
    <table>
        <tr>
            <td>First</td>
```

(continues on next page)

(continued from previous page)

```
            <td>Second</td>
            <td>Third</td>
        </tr>
        <tr>
            <td>Fourth</td>
            <td>Fifth</td>
            <td>Sixth</td>
        </tr>
    </table>
    <br />
    <!-- We set the table to 100% width so that it will expand out -->
    <table style="width:100%">
        <tr>
            <td>First</td>
            <td>Second</td>
            <td>Third</td>
        </tr>
        <tr>
            <td>Fourth</td>
            <td>Fifth</td>
            <td>Sixth</td>
        </tr>
    </table>
</body>
</html>
```

Note: Only 100% is currently supported as a relative value. 50% or 5em etc. are not supported - as pages are fixed size, the required dimensions are known and scryber uses the available space.

10.48 Positioning your content - PD

Scryber has an intelligent layout engine. By default eveything will be laid out as per the flowing layout of the document body, sections and columns. Each component, be it block level or inline will have a position next to its siblings and move along in the document. If the content comes to the end of the page and cannot be fitted then, if allowed, it will be moved to the next page.

10.48.1 Inline Positioning

Inline components such as text and spans will continue on the current line, and if they do not fit all the contnet, then they will flow onto the next line (or column or page). If the content moves, so the inline content will move with the container.

Carriage returns within the content of the file are ignored by default, as per html (see document_textlayout if you don't want them to be.).

Examples of inline components are spans, labels, text literals, page numbers,

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<body style="margin:20pt; font-size:20pt">
    This is the content of the page,
    <span style="color:maroon">
        and this will continue on the current line until it reaches the end
        and then flow onto the next line.
    </span>
    This with then flow after the line.<br />
    A line break forces a new line in the content but flow in the page (#<page />)␣
↪will continue.
    <span style="color:maroon; font-size:30pt;">It also supports the use of multiple␣
↪font sizes</span> in multiple lines,
    adjusting the line height as needed.
</body>
</html>
```

For more information on laying out textual content see document_textlayout

10.48.2 Block Positioning

A block starts on a new line in the content of the page. Children will be laid out within the block (unless absolutely positioned), and content after the block will also begin a new line.

Examples of blocks are Div's, Paragraphs, Tables, BlockQuotes, Headings, Images, and Shapes.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<body style="margin:20pt; font-size:20pt">
    This is the content of the page,

    <div style="color:maroon">
        This will always be on a new on the line, and it's content will then continue␣
↪inline until it reaches the end
        and then flow onto the next line.
    </div>

    After a block, this with then continue with the previous flow on the next line.
↪<br />
    A line break forces a new line in the content but flow in the page (#<page />)␣
↪will continue.
```

(continues on next page)

(continued from previous page)

```
    <div style="color:#666600">
        Blocks also support the use of inline and block content within them
        <span style="color:#006666;font-size:30pt">in multiple lines, adjusting the␣
↪line height as needed.</span>
        <div>As a separate block within the container</div>
    </div>
</body>
</html>
```

Blocks also support the use of backgrounds, borders, margins and padding. They also support document_columns

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<body style="margin:20pt; font-size:20pt">
    This is the content of the page,

    <div style="color:maroon; margin: 20pt 10pt 10pt 10pt">
        This will always
        be on a new on the line, and it's content will then continue inline
        until it reaches the end and then flow onto the next line.
    </div>

    After a block, this with then continue with the previous flow on the next line.
↪<br />
    A line break forces a new line in the content but flow in the page (#<page />)␣
↪will continue.

    <div style="color:#666600; background-color:#BBBB00; padding:10pt;
                margin: 10pt; column-count: 2">
        Blocks also supports the use of inline and block content within them

        <span style="color:#006666; font-size:30pt;">
            in multiple lines,
            adjusting the line height as needed.
        </span>

        <div style="color:black; background-color:white; break-before:always;">
            As a separate block within the container
        </div>
        And coming after the child block.
    </div>
</body>
</html>
```

10.48.3 Changing the display mode

Scryber (currently) supports the following values for the display style mode:

- block
- inline
- none

It is posible to change the default display mode for many components on the page. A span can be a block and a div can be inline. Images and shapes (see document_images and drawing_paths) also support the use of the the display mode.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<body style="margin:20pt; font-size:20pt">
    <div style="color: black; border-width: 1pt">
        The content of this div is all as a block (by default)

        <div style="color: maroon">This div is positioned as a block.</div>

        <!-- Images are by detault displayed as blocks -->
        <img style="width:60pt" src="./Images/group.png" />

        After the content.
    </div>

    <div style="color: black; border-width: 1pt">
        The content of this div is all as a block (by default)

        <div style="color: maroon; display: inline">This div is positioned as a block.
↪</div>

        <!-- Images can be inline and will adjust the line height as needed -->
        <img style="width:60pt; display:inline" src="./Images/group.png" />

        After the content.
    </div>

    <!-- The display:none is also supported, and will not display the content. -->
    <div style="color: black; border-width: 1pt; display: none;">
        The content of this div is all as a block (by default)

        <div style="color: maroon; display: inline">This div is positioned as a block.
↪</div>
```

(continues on next page)

(continued from previous page)

```
        <!-- Images are by detault displayed as blocks -->
        <img style="width:60pt; display:inline" src="./Images/group.png" />

        After the content.
    </div>
</body>
</html>
```

4_styles/images/documentpositioningblocks3.png

10.48.4 Relative Positioning

When you set the position-mode to Relative, it declares the position of that component relative to the block parent. The component will no longer be in the flow of any inline content, nor alter the layout of the following components.

Warning: In HTML relative has a different meaning, scryber uses the container block offsets for relative positions and the page for absolute.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
    <head>
        <style type="text/css">
            .bordered{
                border: solid 1pt black;
                padding:5pt;
                background-color: #AAA;
            }
        </style>
    </head>
    <body style="margin:20pt; font-size:20pt">
        This is the content of the page,

        <div class="bordered">This is the content above the block.</div>

        <div class="bordered">
            This is the flowing content within the block that will span over multiple␣
↪lines
            <span style="position:relative; background-color:aqua">This is relative</
↪span>
            with the content within it.
        </div>

        <div class="bordered">
            After a block, this will then continue with the previous flow of content.
```

(continues on next page)

(continued from previous page)

```
        </div>
    </body>
</html>
```

By default the position will be 0,0, but using the top and left values it can be altered. As soon as a left or top value are specified, the position:relative becomes inferred and is not needed.

Any parent blocks will grow to accomodate the content including any of it's relatively positioned content. And push any content after the block down.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">
        .bordered{
            border: solid 1pt black;
            padding:5pt;
            background-color: #AAA;
        }
    </style>
</head>
<body style="margin:20pt; font-size:20pt">
    This is the content of the page,

    <div class="bordered">This is the content above the block.</div>

    <div class="bordered">
        This is the flowing content within the block that will span over multiple␣
↪lines
        <span style="position:relative; top:300pt; left:60pt; background-color:aqua">
↪This is relative</span>
        with the content within it.
    </div>

    <div class="bordered">
        After a block, this will then continue with the previous flow of content.
    </div>
</body>
</html>
```

Note: By applying a position of relative the span (which is normally inline has automatically become a block and supports the background colours etc.

10.48.5 Absolute Positioning

Changing the positioning mode to Absolute makes the positioning relative to the current page being rendered. The component will no longer be in the flow of any content, nor alter the layout of following components.

The parent block will NOT grow to accomodate the content. The content within the absolutely positioned component will be flowed within the available width and height of the page, but if a size is specified, then this will be honoured over and above the page size.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">
        .bordered{
            border: solid 1pt black;
            padding:5pt;
            margin:5pt;
            background-color: #AAAAAA;
        }
    </style>
</head>
<body style="margin:20pt; font-size:20pt">
    This is the content of the page,

    <div class="bordered">This is the content above the block.</div>

    <div class="bordered">
        This is the flowing content within the block that will span over multiple␣
↪lines
        <span style="left:300pt; top:60pt; position:absolute; background-color:aqua">
            This is absolute
        </span>
        with the content within it.
    </div>

    <div class="bordered">
        After a block, this will then continue with the previous flow of content.
    </div>

    <img src="./images/group.png" style="position:absolute; top:150pt; left:500pt;␣
↪height:150pt; opacity:0.7;" />
</body>
</html>
```

10.48.6 Numeric Positioning

All content positioning is from the top left corner of the page or parent. This is a natural positioning mechanism for most cultures and developers. (unlike PDF, which is bottom left to top right).

Units of position can either be specified in

- points (1/72 of an inch) e.g *36pt*,
- inches e.g. *0.5in* or
- millimeters e.g. *12.7mm*
- pixels (1/96 of an inch) e.g. *48px*

If no units are specified then the default is points. See drawing_units for more information.

Note: 100% is also supported for widths to allow for the full-width capability. More support for percentage widths may be added in future.

10.48.7 Rendering Order

All relative or absolutely positioned content will be rendered to the output in the order it appears in the document. If a block is relatively positioned, it will overlay any content that preceded it, but anything coming after will be over the top.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">
        .bordered{
            border: solid 1pt black;
            padding:5pt;
            margin:5pt;
            background-color: #EEEEEE;
        }
    </style>
</head>
<body style="margin:20pt; font-size:20pt">
    This is the content of the page,

    <div class="bordered">This is the content above the block.</div>

    <div class="bordered">
        This is the flowing content within the block that will span over multiple␣
↪lines
```

(continues on next page)

(continued from previous page)

```
        <span style="left:25pt; top:20pt; background-color:aqua; padding:4pt;">
            This is relatively positioned
        </span>
        with the content within it.
    </div>

    <div class="bordered" style="padding:10pt 10pt 10pt 60pt">
        <img src="./images/group.png"
            style="position:relative; top:-10pt; left:-40pt; width:100pt; opacity:0.5;
↪" />
        This is the content that will flow over the top with the 60 point left␣
↪padding and the
        image set at -40, -10 relative to the container with a width of 100pt
        and a 50% opacity.
    </div>

</body>
</html>
```

By using this rule interesting effects can be designed.

10.48.8 Position z-index

It's not currently supported, within scryber to specify a z-index on components. It may be supported in future.

10.48.9 Drawing Canvas

For complete control of drawing content, scryber supports svg. This can be used as drawing support for shapes and paths etc. See drawing_paths for more details.

10.49 Borders on blocks - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.50 Strokes on components - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.51 Backgrounds on blocks - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.52 Fills on components - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.53 Images as backgrounds and fills - PD

Images are also supported on the backgrounds of block level components (see component_positioning), and of fills for shapes, text, etc.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        div.bg {
            background-image: url("./images/landscape.jpg");
            min-height: 260px;
            text-align:center;
            color:white;
            font-family: sans-serif;
            font-size:larger;
            font-weight:bold;
            padding-top:10pt;
        }

    </style>
</head>
<body style="padding:20pt;">
    <div class="bg" style="">
        <span>Background image with the default settings</span>
    </div>

</body>
</html>
```

The background has been drawn with the image repeating from the top left corner at its natural size (or default 96ppi), clipped to the boundary of the container.

Along with specifying the image background, there are various other options for how the pattern is laid out that will change the defaults of how the image repeats.

10.53.1 Background Size

The background size option can either be a specific size, or 'cover' which will cover the entire container as a single image.

(Scryber does not currently support 'contain' but it's on our roadmap).

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        div.bg {
            background-image: url("./images/landscape.jpg");
            min-height: 260px;
            text-align:center;
            color:#333;
            font-family: sans-serif;
            font-size:larger;
            font-weight:bold;
            padding-top:10pt;
            border:solid 1px #333;
        }

    </style>
</head>
<body style="padding:20pt;">

    <div class="bg" style="background-size: 40pt 40pt; color:white;">
        <span>Background image with explicit size</span>
    </div>
    <br/>
    <div class="bg" style="background-size:cover">
        <span>Background image with cover</span>
    </div>

</body>
</html>
```

4_styles/images/drawingImagesBackgroundSize.png

10.53.2 Background Repeat

The options for the background repeating are:

- repeat - The default value, where the image repeats both X and Y directions.
- repeat-x - The background will only repeat in the X (horizontal) direction.
- repeat-y - The background will only repeat in the Y (vertical) direction.
- none - The background will only be shown once.

These can be applied with a size, but will not affect anything if the size is cover.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        div.bg {
            background-image: url("./images/landscape.jpg");
            min-height: 260px;
            text-align:center;
            font-family: sans-serif;
            font-size:larger;
            font-weight:bold;
            padding-top:10pt;
            border:solid 1px #333;
            /* consistent size across all */
            background-size: 60pt 60pt;
        }

    </style>
</head>
<body style="padding:20pt;">


    <div style="column-count:2; margin-bottom: 10pt; color:white;">
        <div class="bg" style="background-repeat:repeat; break-after:always;">
            <span>Background image with the default repeat</span>
        </div>
        <div class="bg" style="background-repeat:repeat-x">
            <span>Background image with repeat horizontal</span>
        </div>
    </div>

    <div style="column-count:2; color:#333;">
        <div class="bg" style="background-repeat:repeat-y; break-after:always;">
            <span>Background image with repeat vertical</span>
        </div>

        <div class="bg" style="background-repeat:no-repeat">
            <span>Background image with no repeating</span>
        </div>

    </div>
</body>
</html>
```

10.53.3 Background Position

- **The starting position of the pattern.**
  - x-pos - Determines the horizontal offset of the rendered background image in units.
  - y-pos - Determines the vertical offset of the rendered background image in units.
- **The pattern repeat step.**
  - x-step - Sets the horizontal offset between repeating patterns, which can be more or less than the size of the rendered image.
  - y-step - Sets the vertical offset between repeating patterns, which can be more or less than the size of the rendered image.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <style type="text/css">

        div.bg {
            background-image: url("./images/landscape.jpg");
            min-height: 260px;
            text-align: center;
            font-family: sans-serif;
            font-size: larger;
            font-weight: bold;
            padding-top: 10pt;
            border: solid 1px #333;
            /* consistent size across all */
            background-size: 60pt 60pt;
        }
    </style>
</head>
<body style="padding:20pt;">


    <div style="column-count:2; margin-bottom: 10pt; color:white;">
        <!-- Position value for x and y -->
        <div class="bg"
            style="background-repeat:repeat;
                    background-position: 20pt 20pt;
                    break-after:always;">
            <span>Background image with the default repeat at 20,20</span>
        </div>
        <!-- Single value should be applied to both x and y -->
        <div class="bg"
```

(continues on next page)

(continued from previous page)

```
            style="background-repeat:repeat-x;
                    background-position: 20pt">
            <span>Background image with repeat horizontal at 20,20</span>
        </div>
    </div>

    <div style="column-count:2; color:#333;">
        <!-- x and y as individual properties -->
        <div class="bg"
            style="background-repeat: repeat-y;
                    background-position-x: 20pt;
                    background-position-y: 40pt;
                    break-after: always;">
            <span>Background image with repeat vertical at 20,20</span>
        </div>
        <!-- Single repeat with a bakground color -->
        <div class="bg"
            style="background-repeat: no-repeat;
                    background-position: 150pt 100pt;
                    background-color: aquamarine">
            <span>Background image with no repeating at 150,100 and background color</
↪span>
        </div>

    </div>
</body>
</html>
```

10.53.4 Images as fills

Scryber also supports images as fills. See the SVG documentation for this.

10.54 Linear and Radial Gradients - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.55 Textual wrapping and spacing - PD

10.55.1 Line leading

TODO

10.55.2 Word wrapping

By default, scryber will wrap text around the available space and flow evenly across the page, no matter the content in the source.

If this is not the desired behaviour, then the css attributes for white-space are supported.

- nowrap - will ignore white space, AND not wrap the content when the outer edge is reached.
- pre - will take all white space into account and render content as seen.

The layout also supports the use of overflow-x and overflow-y to clip the visibility to the bounds of the container. (Scroll is not supported).

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Document Text H Align</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font {
            font-size: 14pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
            font-family: sans-serif;
        }
    </style>
</head>
<body style="padding: 20pt">
    <div class="std-font" style="white-space: nowrap">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.Fusce␣
↪pulvinar elit leo, sit amet egestas neque porttitor nec.
        Nunc pellentesque turpis ac pellentesque scelerisque.

        Etiam at nibh mattis, pulvinar velit eget, consequat ligula.
    </div>

    <div class="std-font" style="white-space: nowrap">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.Fusce␣
↪pulvinar elit leo, sit amet egestas neque porttitor nec.<br />
        Nunc pellentesque turpis ac pellentesque scelerisque.<br />

        Etiam at nibh mattis, pulvinar velit eget, consequat ligula.<br />
    </div>

    <div class="std-font" style="white-space:pre; overflow-x:hidden;">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.Fusce␣
↪pulvinar elit leo, sit amet egestas neque porttitor nec.
    Nunc pellentesque turpis ac pellentesque scelerisque.

    Etiam at nibh mattis, pulvinar velit eget, consequat ligula.
    </div>

</body>
```

(continues on next page)

(continued from previous page)

```
</html>
```

4_styles/images/documentTextPre.png

10.55.3 Character and Word Spacing

With scryber the character and word spacing is supported at the style definition level (not on the component attributes). They are less frequently used, but can help in adjusting fonts that are too narrow at a particular size, or for graphical effect.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>Document Character spacing</title>
    <meta name="author" content="Scryber Team" />
    <style type="text/css">

        .std-font {
            font-size: 14pt;
            background-color: #AAA;
            padding: 4pt;
            margin-bottom: 10pt;
            font-family: 'Segoe UI', sans-serif;
        }

        .narrow{ letter-spacing:-0.5pt;}

        .wide{ letter-spacing:1.5pt; line-height:15pt; }

        .wide-word{ letter-spacing: 0; word-spacing: 10pt; }

    </style>
</head>
<body style="padding: 20pt">
    <div style="column-count:3;font-size:10pt">
        <div class="std-font narrow" style="break-after:always">
            Segoe UI in 10pt font size with the default
            leading used on each line of the paragraph. But the character spacing is␣
↪reduced by 0.5 points.
        </div>
        <div class="std-font wide" style="break-after:always">
            Segoe UI in 10pt font size with the leading increased to 15pt
            on each line of the paragraph. The character spacing is also
            set to an extra 1.5 points.
        </div>
        <div class="std-font wide-word">
            Segoe UI in 10pt font size with the leading and character space normal,␣
↪but the word
```

(continues on next page)

(continued from previous page)

```
            spacing increased by 5 points. It should continue to flow nicely onto␣
↪multiple lines.
        </div>
    </div>

    <div class="std-font wide" style="line-height:30pt;" >
        Even using various
        <span style="font-size:30pt; font-family:Optima, serif;">font sizes and␣
↪families</span>
        will maintain the character and
        word spacing that <b>has been applied.</b>
    </div>

</body>
</html>
```

Note: There is a known issue with the baseline adjust on multiple font sizes that has crept in, and will hopefilly be resolved in the next release.

10.55.4 Wrapping and spacing in code

10.55.5 Next Steps

10.56 Transformations - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.57 Floats and positions - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.58 @page support - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.59 Building styles in code

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.60 Building styles in code

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.61 Drawing with SVG - PD

Scryber includes the drawing capability with a subset of SVG capabilities.

- Lines
- Recangles
- Elipses
- Polygons
- Bezier Curves
- Groups
- Text
- Use and definitions
- ViewPorts

The drawing components should all be within a namespace qualified svg element, or prefixed svg at the document root.

There are may resources for SVG available, below is a description of the capabilities of scryber's implementation of SVG. And it is perfectly possible to draw an entire page in SVG within a body or section.

10.61.1 Drawing SVG content

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML >

<html xmlns='http://www.w3.org/1999/xhtml' >
<head>
    <style type="text/css">

        svg{
            font-family:sans-serif;
            font-size:12pt;
        }

        .colored {
            fill: blue;
        }
```

(continues on next page)

(continued from previous page)

```
    </style>
</head>
<body style="padding:20pt;">
    <p>The svg content is below</p>
    <!-- Adding an svg rect and some text to the page -->
    <svg xmlns="http://www.w3.org/2000/svg" >
        <rect class="colored" width="100pt" height="100pt" >
        </rect>
        <text x="20" y="50" fill="white" >I'm SVG</text>
    </svg>
    <p>And after the svg content</p>
</body>
</html>
```

5_shapes/images/drawingPathsSVG.png

As can be seen, the svg content is as a block and renders within the flow of the content. The rect(angle) picks up the styles from the css and the font flows down from the svg container. The use of svg a an inline, or inline-block may be supported in the future.

With scryber it is possible to use svg elements directly in the document by delcaring a prefixed namespace at the top. And the example below will render the same. (See namespaces_and_assemblies for more information on how namespaces are used)

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML >

<!-- Declare the naespace here -->
<html xmlns='http://www.w3.org/1999/xhtml'
    xmlns:svg="http://www.w3.org/2000/svg" >
<head>
    <style type="text/css">

        .canvas {
            font-family: sans-serif;
            font-size: 12pt;
        }
        .colored {
            fill: blue;
        }

    </style>
</head>
<body style="padding:20pt;">
    <p>The svg content is below</p>
    <div class="canvas">
        <!-- And prefix our elements here (in a div) -->
        <svg:rect class="colored" width="100pt" height="100pt" >
        </svg:rect>
        <svg:text x="20" y="50" fill="#EEF" >I'm SVG</svg:text>
```

(continues on next page)

(continued from previous page)

```
    </div>
    <p>And after the svg content</p>
</body>
</html>
```

Note: depending on the purpose, this might be advantageous. But not make any html parsers happy unless wrapped in an svg:svg element.

All examples below will follow the standard <svg xmlns=" > convention.

10.61.2 Supported shapes

Scryber supports the standard shapes for rectangles, elipses, circles and lines. Generally, as closed shapes they will have a black fill and no stroke.

A group group (g) can contain multiple shapes and paths, and alter the style of inner content, e.g. applying a constitent stroke.

Without a width or height the svg element in scryber with size to the inner content, but it is good practice to specify values.

Scryber also supports the use of styles on the svg element itself.

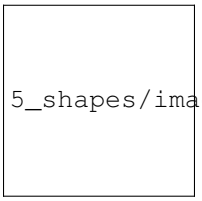
```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML >

<html xmlns='http://www.w3.org/1999/xhtml' >
<head>
</head>
<body style="padding:20pt;">
    <p>The svg content is below</p>

    <svg xmlns="http://www.w3.org/2000/svg" style="border:solid 1px black" >
        <rect x="0pt" y="0pt" width="100pt" height="80pt" fill="lime" ></rect>
        <g id="eye" stroke="black" stroke-width="2pt" >
            <ellipse cx="50pt" cy="40pt" rx="40pt" ry="20pt" fill="white"></ellipse>
            <circle cx="50pt" cy="40pt" r="20pt" fill="#66F"></circle>
            <circle cx="50pt" cy="40pt" r="10pt" fill="black"></circle>
            <line x1="10" x2="90" y1="40" y2="40" />
            <line x1="50" x2="50" y1="20" y2="60" />
        </g>
    </svg>

    <p>And after the svg content</p>
</body>
</html>
```

5_shapes/images/drawingPathsSVGShapes.png

10.61.3 Polylines, gons and paths

Scryber supports the standard paths, polylines and polygons

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML >

<html xmlns='http://www.w3.org/1999/xhtml' >
<head>
    <style type="text/css">

        .transparentish {
            fill: aqua;
            stroke:blue;
            stroke-width:2pt;
            fill-opacity: 0.5;
            stroke-opacity: 0.5;
        }

        .img-bg{
            fill: url(./images/landscape.jpg);
        }

    </style>
</head>
<body style="padding:20pt;">
    <p>The svg content is below</p>
    <div style="text-align:center;">

        <svg xmlns="http://www.w3.org/2000/svg" style="border:solid 1px black" width=
↪"310" height="110">
            <path class="transparentish img-bg" d="M 10,30
        A 20,20 0,0,1 50,30
        A 20,20 0,0,1 90,30
        Q 90,60 50,90
        Q 10,60 10,30 z" ></path>
            <polyline class="transparentish" points="150,5 121,95 198,40 102,40 179,95
↪" stroke="blue"
                    stroke-width="2pt"></polyline>
            <polygon class="transparentish" points="250,5 221,95 298,40 202,40 279,95
↪" stroke="blue"
                    stroke-width="2pt"></polygon>
        </svg>
    </div>
    <p>And after the svg content</p>
</body>
</html>
```



A path has the operations explicitly defined within the 'd' attribute, see below.

A polyline is rendered using secific x,y points from the top left of the container.

A polygon automatically closes the path.

Scryber does not currently support the use of patterns or gradients as fills e.g. fill='url(#mypattern)', but does support images as fills, and backgrounds e.g. fill='url(./path/toimage.png)'. We will look at this for future releases.

10.61.4 Drawing paths

Scryber supports the use of bezier paths for the creation of the complex curves and shapes. The format of the drawing data (d) is exacly the same as the **svg** drawing operations.

- M = moveto
- L = lineto
- H = horizontal lineto
- V = vertical lineto
- C = curveto
- S = smooth curveto
- Q = quadratic Bézier curve
- T = smooth quadratic Bézier curveto
- A = elliptical Arc
- Z = closepath

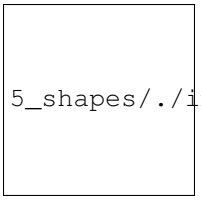
```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML >

<html xmlns='http://www.w3.org/1999/xhtml' >
<head>
    <style type="text/css">

        .transparentish {
            fill: aqua;
            stroke:blue;
            stroke-width:2pt;
            fill-opacity: 0.5;
            stroke-opacity: 0.5;
        }

    </style>
</head>
<body style="padding:20pt;">
    <p>The svg content is below</p>
    <div style="text-align:center;">
        <svg id="ClockIcon" top="0" left="0" width="100" height="100" viewBox="0 0 20␣
↪20" xmlns="http://www.w3.org/2000/svg">
            <path fill="blue" d="M11.088,2.542c0.063-0.146,0.103-0.306,0.103-0.476c0-
↪0.657-0.534-1.19-1.19-1.19c-0.657,0-1.19,
            0.533-1.19,1.19c0,0.17,0.038,0.33,0.102,0.476c-4.085,0.535-7.243,
            4.021-7.243,8.252c0,4.601,3.73,8.332,8.332,8.332c4.601,0,
            8.331-3.73,8.331-8.332C18.331,6.562,15.173,3.076,11.088,2.542z M10,
            1.669c0.219,0,0.396,0.177,0.396,0.396S10.219,2.462,10,2.462c-0.22,
            0-0.397-0.177-0.397-0.396S9.78,1.669,10,1.669z M10,18.332c-4.163,
            0-7.538-3.375-7.538-7.539c0-4.163,3.375-7.538,7.538-7.538c4.162,0,
```

(continues on next page)

(continued from previous page)

```
            7.538,3.375,7.538,7.538C17.538,14.957,14.162,18.332,10,18.332z M10.386,
            9.26c0.002-0.018,0.011-0.034,0.011-0.053V5.24c0-0.219-0.177-0.396-0.396-0.
↪396c-0.22,
            0-0.397,0.177-0.397,0.396v3.967c0,0.019,0.008,0.035,0.011,0.053c-0.689,0.
↪173-1.201,0.792-1.201,1.534c0,0.324,0.098,0.625,0.264,0.875c-0.079,0.014-0.155,0.
↪043-0.216,0.104l-2.244,2.244c-0.155,0.154-0.155,0.406,0,0.561s0.406,0.154,0.561,0l2.
↪244-2.242c0.061-0.062,0.091-0.139,0.104-0.217c0.251,0.166,0.551,0.264,0.875,0.264c0.
↪876,0,1.587-0.711,1.587-1.587C11.587,10.052,11.075,9.433,10.386,9.26z M10,11.586c-0.
↪438,0-0.793-0.354-0.793-0.792c0-0.438,0.355-0.792,0.793-0.792c0.438,0,0.793,0.355,0.
↪793,0.792C10.793,11.232,10.438,11.586,10,11.586z"></path>
        </svg>
        <!-- Icon from dribble -->
    </div>
    <p>And after the svg content</p>
</body>
</html>
```

The viewbox defines the area visible and will scale the content of the svg appropriately to the required width and height.



10.61.5 Line options

The stroke style also supports the standed ending and join options for paths, that will alter the way lines and vertices are rendered.

```
<path id="smiley" fill="yellow" stroke="black" stroke-width="8pt" stroke-linecap=
↪"round" stroke-linejoin="round"
            d="M50,10 A40,40,1,1,1,50,90 A40,40,1,1,1,50,10 M30,40 Q36,35,42,40 M58,
↪40 Q64,35,70,40 M30,60 Q50,75,70,60 Q50,75,30,60" />
```

10.61.6 Definitions and use

Scryber supports the definition of shapes and reuse within the content. This can either be directly, or within another viewbox for scaling and position.

The preserveAspectRatio is the standard svg enumeration that allows the content position in the viewbox to be defined on the outer container.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML>

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>

</head>
<body style="padding:20pt;">
```

(continues on next page)

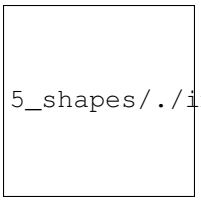
(continued from previous page)

```
    <p>The svg content is below</p>
    <div style="text-align:center;">
        <svg xmlns="http://www.w3.org/2000/svg">

            <!-- define our smiley approx 100 x 100 units -->
            <defs>
                <path id="smiley" fill="yellow" stroke="black" stroke-width="8pt"␣
↪stroke-linecap="round" stroke-linejoin="round"
                    d="M50,10 A40,40,1,1,1,50,90 A40,40,1,1,1,50,10 M30,40 Q36,35,42,
↪40 M58,40 Q64,35,70,40 M30,60 Q50,75,70,60 Q50,75,30,60" />
            </defs>

            <!-- use it in the middle so it is scaled to be fully visible -->
            <svg id="smileyWrapper" x="0" width="50pt" height="25pt" viewBox="0 0 100␣
↪100" style="background-color:#5555FF;"
                preserveAspectRatio="xMidYMid">
                <use href="#smiley" />
            </svg>

            <!-- on the left side fully visible -->
            <svg id="smileyWrapper" x="55" width="50pt" height="25pt" viewBox="0 0␣
↪100 100" style="background-color:#55FF55;"
                preserveAspectRatio="xMinYMid">
                <use href="#smiley" />
            </svg>

            <!-- on the right side fully visible -->
            <svg id="smileyWrapper" x="110" width="50pt" height="25pt" viewBox="0 0␣
↪100 100" style="background-color:#FF5555;"
                preserveAspectRatio="xMaxYMid">
                <use href="#smiley" />
            </svg>

            <!-- slice will make the contents fill the box rather than fit -->

            <!-- top middle -->
            <svg id="smileyWrapper" x="0" y="30" width="50pt" height="25pt" viewBox=
↪"0 0 100 100" style="background-color:#5555FF;"
                preserveAspectRatio="xMidYMin slice">
                <use href="#smiley" />
            </svg>

            <!-- middle middle -->
            <svg id="smileyWrapper" x="55" y="30" width="50pt" height="25pt" viewBox=
↪"0 0 100 100" style="background-color:#55FF55;"
                preserveAspectRatio="xMidYMid slice">
                <use href="#smiley" />
            </svg>

            <!-- bottom middle -->
            <svg id="smileyWrapper" x="110" y="30" width="50pt" height="25pt" viewBox=
↪"0 0 100 100" style="background-color:#FF5555;"
                preserveAspectRatio="xMidYMax slice">
                <use href="#smiley" />
            </svg>

            <!-- meet vertical align -->
```

(continues on next page)

(continued from previous page)

```
            <svg id="smileyWrapper" x="165" width="25pt" height="55pt" viewBox="0 0␣
↪100 100" style="background-color:#5555FF;"
                preserveAspectRatio="xMidYMin">
                <use href="#smiley" />
            </svg>

            <svg x="195" width="25pt" height="55pt" style="background-color:#55FF55;"␣
↪viewBox="0 0 100 100"
                preserveAspectRatio="xMidYMid meet">
                <use href="#smiley" />
            </svg>

            <svg x="225" y="0" width="25pt" height="55pt" style="background-color:
↪#FF5555;" viewBox="0 0 100 100"
                preserveAspectRatio="xMidYMax meet">
                <use href="#smiley" />
            </svg>

            <!-- scale vertical align -->

            <svg id="smileyWrapper" x="255" width="25pt" height="55pt" viewBox="0 0␣
↪100 100" style="background-color:#5555FF;"
                preserveAspectRatio="xMinYMax slice">
                <use href="#smiley" />
            </svg>

            <svg x="285" width="25pt" height="55pt" style="background-color:#55FF55;"␣
↪viewBox="0 0 100 100"
                preserveAspectRatio="xMidYMax slice">
                <use href="#smiley" />
            </svg>

            <svg x="315" y="0" width="25pt" height="55pt" style="background-color:
↪#FF5555;" viewBox="0 0 100 100"
                preserveAspectRatio="xMaxYMax slice">
                <use href="#smiley" />
            </svg>

            <!-- Finally just fill the box -->

            <svg x="0" y="60" width="340pt" height="155pt" style="background-color:
↪#555555;" viewBox="0 0 100 100"
                preserveAspectRatio="none">
                <use href="#smiley" />
            </svg>

        </svg>
    </div>
    <p>And after the svg content</p>
</body>
</html>
```

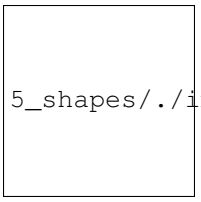


10.61.7 SVG Text

Scryber supports the use of the SVG Text and text spans for rendering characters within the drawing.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML>

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <link type="text/css" rel="stylesheet" href="https://fonts.googleapis.com/css2?
↪family=Roboto:ital,wght@0,100;0,700;1,100&amp;display=swap" />
    <style>

        body {
            font: 12pt 'Roboto';
            padding: 20pt;
        }

        .small {
            font: italic 13px 'Roboto';
        }

        .heavy {
            font: bold 30px 'Roboto';
        }

        .red {
            font: italic 40px 'Roboto';
            fill: red;
        }

    </style>
</head>
<body style="padding:20pt;">
    <p>The svg content is below</p>
    <div style="text-align:center;">
        <svg xmlns="http://www.w3.org/2000/svg">
            <text x="20" y="35" class="small">My</text>
            <text x="35" y="35" class="heavy">cat</text>
            <text x="55" y="60" class="small">is</text>
            <text x="60" y="60" class="red">Grumpy!</text>
        </svg>
    </div>
    <p>And after the svg content</p>
</body>
</html>
```

Here we are linking to and using the Roboto font from the google api's.



10.61.8 Referencing drawings

It is also possible to load an svg file directly into the document with an embed option.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML>

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <link type="text/css" rel="stylesheet" href="https://fonts.googleapis.com/css2?
↪family=Roboto:ital,wght@0,100;0,700;1,100&amp;display=swap" />
    <style>

        body {
            font: 12pt 'Roboto';
            padding: 20pt;
        }

        .small {
            font: italic 13px 'Roboto';
        }

        .heavy {
            font: bold 30px 'Roboto';
        }

        .red {
            font: italic 40px 'Roboto';
            fill: red;
        }

    </style>
</head>
<body style="padding:20pt;">
    <p>The svg content is below</p>
    <div style="text-align:center;">
        <embed src="./Fragments/MyDrawing.svg" />
    </div>
    <p>And after the svg content</p>
</body>
</html>
```

And the referenced SVG file is ./Fragments/MyDrawing.svg

```
<svg xmlns="http://www.w3.org/2000/svg" width="500" height="400">

    <path id="lineAB" d="M 100 350 l 150 -300" stroke="red" stroke-width="3" fill=
↪"none" />
    <path id="lineBC" d="M 250 50 l 150 300" stroke="red" stroke-width="3" fill="none
↪" />
```

(continues on next page)

(continued from previous page)

```
    <path d="M 175 200 l 150 0" stroke="green" stroke-width="3" fill="none" />
    <path d="M 100 350 q 150 -300 300 0" stroke="blue" stroke-width="5" fill="none" />
    <!-- Mark relevant points -->
    <g stroke="black" stroke-width="3" fill="black">
        <circle id="pointA" cx="100" cy="350" r="3" />
        <circle id="pointB" cx="250" cy="50" r="3" />
        <circle id="pointC" cx="400" cy="350" r="3" />
    </g>
    <!-- Label the points -->
    <g font-size="30" font-family="sans-serif" fill="black" stroke="none" >
        <text x="70" y="350" >A</text>
        <text x="220" y="60" >B</text>
        <text x="410" y="350" >C</text>
    </g>
</svg>
```

5_shapes/./images/drawingPathsSVGPathReference.png

10.61.9 Attributes Supported

Only a few of the full capabilities and attributes of SVG are supported. More are supported using the style=" css attribute settings, and we will be adding more in future.

10.62 Styles in SVG - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.63 HR and Lines - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.64 Squares and Rectangles - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.65 Circles and Elipses - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.66 Paths and shapes - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.67 SVG Text and runs - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.68 The drawing viewport - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.69 Groups, definitions and refs - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.70 Unsupported SVG content - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.71 Dynamic content in your template - PD

A document template is just that, a template. You can add any source of information to be included.

- As a discreet value
- As an object with properties
- An array or dictionary
- As a template to use.
- To make decisions on layout

And they can be used in your document in many locations

- Within the document content
- On styles and classes
- In templates and loops
- Referenced content

The notation for an binding on an attribute or content is based on the { and } with a method (@ for the built model), a colon ':', and then finally the selector.

e.g. *attribute='{@:paramName}'* for values or *attribute='{@:paramName.property[index].value}'* for objects or even in textual content.

10.71.1 Document parameters

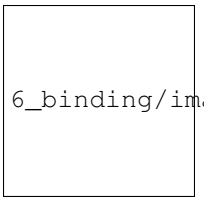
Every Document can have parameters associated with it, and these can simply be bound to the content. This includes style and value properties as well as text, and can be set after parsing the content.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>{@:DocTitle}</title>
    <meta name="author" content="{@:DocAuthor}" />
</head>
<body style="margin:20pt; font-size:20pt">
    <header>
        <span style="{@:ThemeHeader}">{@:DocAuthor}</span>
    </header>
    <div>
        <h1 class="title" >{@:DocTitle}</h1>
        <span>{@:DocContent}</span>
    </div>

</body>
</html>
```

And the value can be set or changed at runtime

```
using (var doc = Document.ParseDocument(path))
{
    //pass paramters as needed, supporting simple values, arrays or complex classes.

    using (var stream = DocStreams.GetOutputStream("documentation.pdf"))
    {
        doc.Params["DocTitle"] = "Binding Title";
        doc.Params["DocAuthor"] = "Binding Name";
        doc.Params["ThemeHeader"] = "background-color:#EEE;padding:5pt";
        doc.Params["DocContent"] = "This is the content of the document";

        doc.SaveAsPDF(stream);
    }

}
```

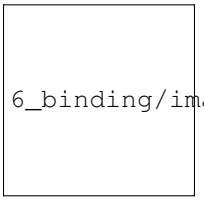
And this will be used in the output.



10.71.2 Using objects

Simple values work well, but with complex entries it will start to get extremely complex. Scryber supports the standard object notations for properties arrays and dictionaries to help divide up the binding.

As a use case, we may need some purchase details.




```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.
↪dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>
<head>
    <title>{@:Content.Title}</title>
    <meta name="author" content="{@:DocAuthor}" />
    <style>
        .header-details{ column-count:3; font-size:10pt; vertical-align:middle;}
        .header-column { break-after:always; text-align: center;}
        .header-column.logo { text-align: left; height: 40pt; max-width: 120pt;}

        .item{ border:solid 0.5px gray; }

        .list{ width:100%; font-size:12pt;}

        .item.vat, .item.price, .item.qty, .item.value{ width:60pt; text-align:right;␣
↪}
        .total, .list thead { font-weight:bold;}

        .total.empty{ border:none; }

    </style>
</head>
<body style="font-size:20pt">
    <!-- Page header with theme logo and content -->
    <header>
        <div class="header-details" style="{@:Theme.Header}">
            <img class="header-column logo" src="{@:Theme.Logo}" />
            <div class="header-column title" >{@:Content.Title}</div>
            <div class="header-column author" >{@:Content.Author}</div>
        </div>
    </header>
    <div style="margin:20px">
```

(continues on next page)

(continued from previous page)

```
        <h1 class="title" >{@:Content.Title}</h1>
        <!-- A table of contents using the same theme for static headers -->
        <table class="list" >
            <thead>
                <tr style="{@:Theme.Header}">
                    <td>Item</td>
                    <td class="item price">Price</td>
                    <td class="item qty">Qty</td>
                    <td class="item value">Total</td>
                </tr>
            </thead>
            <tbody>
                <!-- and a template for the table rows looping over each of the items␣
↪-->
                <template data-bind="{@:Model.Items}">
                    <tr>
                        <!-- each one is bound with a . prefix for the current item --
↪>
                        <td class="item name">
                            <span>{@:.Item}</span>
                        </td>
                        <td class="item price" >
                            <span>{@:.Price}</span>
                        </td>
                        <td class="item qty" >
                            <span>{@:.Quantity}</span>
                        </td>
                        <td class="item value" >
                            <span>{@:.Value}</span>
                        </td>
                    </tr>
                </template>
            </tbody>
            <tfoot>
                <!-- Footer rows for the titles -->
                <tr>
                    <td class="total empty" style="border:none;"></td>
                    <td><span>Tax:</span></td>
                    <td class="total vat" style="width:60pt; text-align:right;">
                        <span>{@:Model.Tax.Rate}</span>
                    </td>
                    <td class="total vat" style="width:60pt; text-align:right;">
                        <span>{@:Model.Tax.Value}</span>
                    </td>
                </tr>
                <tr>
                    <td colspan="3" class="total empty" style="border:none;" ></td>
                    <td class="total grand" style="width:60pt; text-align:right;">
                        <span>{@:Model.Total.Value}</span>
                    </td>
                </tr>
            </tfoot>
        </table>
    </div>
    <div id='footnote' style="padding-left:40pt; font-size: 14pt;">
        <span>Kind regards</span><br/>
        <i>{@:Content.Author}</i>
```

(continues on next page)

(continued from previous page)

```
    </div>
</body>
</html>
```

And with that we can bind the source into the document

```
using (var doc = Document.ParseDocument(path))
{
    //pass paramters as needed, supporting simple values, arrays or complex classes.

    using (var stream = DocStreams.GetOutputStream("documentation.pdf"))
    {
        doc.Params["Theme"] = new {
                        Header = "background-color:#666; color: white;padding:5pt",
                        Logo = "./images/ScyberLogo2_alpha_small.png"
        };

        doc.Params["Content"] = new {
            Title = "Purchase List",
            Author = "The Scryber Team"
        };

        doc.Params["Model"] = new
        {
            Items = new[] {
                        new { Item = "First Item", Quantity = "4", Price = "€50.00",␣
↪Value = "€200.00" },
                        new { Item = "Second Item", Quantity = "2", Price = "€25.00",␣
↪Value = "€50.00" },
                        new { Item = "Third Item", Quantity = "3", Price = "€100.00",␣
↪Value = "€300.00" }
                    },
            Tax = new { Rate = "20%", Value = "€110.00" },
            Total = new { Value = "€660.00" }
        };


        doc.SaveAsPDF(stream);
    }

}
```

10.71.3 Injecting content

If it is needed to inject some dynamic content within the document then it is easy to look up elements and then add the content either as html or as code.

Let's say the ask was to add an optional foot note to our Purchase list for the high demand items, and also a custom footer to the pages. We can do this in our code, without changing the template.

```
if (IsHighDemandItem())
{
    //Add the content to the footnote

    var div = doc.FindAComponentById("footnote") as Div;
```

(continues on next page)

(continued from previous page)

```
    //Lets do this via conversion of dynamic xhtml into a component
    //Still needs to be valid XHTML
    var footnoteContent = "<div xmlns='http://www.w3.org/1999/xhtml'><span>Warmest␣
↪regards from all the scryber team</span><br/>" +
        "<i>" + System.Environment.UserName + "</i><br/><br/>" +
        "<b>Your order is for a high demand item. Please allow 6 weeks for delivery</
↪b></div>";

    var content = doc.ParseTemplate(doc, new System.IO.StringReader(footnoteContent))␣
↪as Component;

    //Remove the old content, as we want to
    div.Contents.Clear();
    div.Contents.Add(content);
}
```

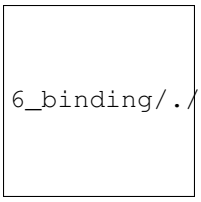
The string content is parsed, so needs to be xhtml, but then simply added to an existing div with a matching ID.

And for the footer, we use the IPDFTemplate that is used for all dynamic content building - Headers, Footers, HTML-Templates, etc.

```
//Add the custom footer
doc.Pages[0].Footer = new CustomFooter();


/// <summary>
/// Implements the IPDFTemplate for a custom footer.
/// </summary>
public class CustomFooter : IPDFTemplate
{
    /// <summary>
    /// Returns the object content (may be called multiple times).
    /// </summary>
    public IEnumerable<IPDFComponent> Instantiate(int index, IPDFComponent owner)
    {
        //Wrap it all in a div so we can set the style

        Div div = new Div() { StyleClass = "footer", FontSize = 10,
                              Padding = new PDFThickness(10),
                              HorizontalAlignment = HorizontalAlignment.Center };

        div.Contents.AddRange(new Component[]
        {
            new TextLiteral("Page Number "),
            new PageNumberLabel() { DisplayFormat = "{0} of {1}"}
        });

        return new IPDFComponent[] { div };
    }
}
```

As you can see, pretty much anything can be data bound and the output can be altered in any way using the combination of styles, declarative html content, data objects and code.

As expected this will flow with the layout of the document and maybe even onto another page.



10.72 The ‘handlebars’ syntax - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.73 Bindiable properties - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.74 CSS var() and calc support - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.75 Visual decisions with binding - TD

All the visual content in a document sits in pages. Scryber supports the use of both a single body with content within it, and also explicit flowing pages in a section.

10.76 Available expression functions - TD

10.76.1 All the operators

10.76.2 Comparison functions

10.76.3 Math Functions

10.76.4 String Functions

10.76.5 Date Functions

10.76.6 Adding your own

10.76.7 Contributing

10.77 Operators reference - TD

10.77.1 All the operators

10.77.2 Comparison functions

10.77.3 Math Functions

10.77.4 String Functions

10.77.5 Date Functions

10.77.6 Adding your own

10.77.7 Contributing

10.78 The Document Params dictionary - TD

10.78.1 All the operators

10.78.2 Comparison functions

10.78.3 Math Functions

10.78.4 String Functions

10.78.5 Date Functions

10.78.6 Adding your own

10.78.7 Contributing

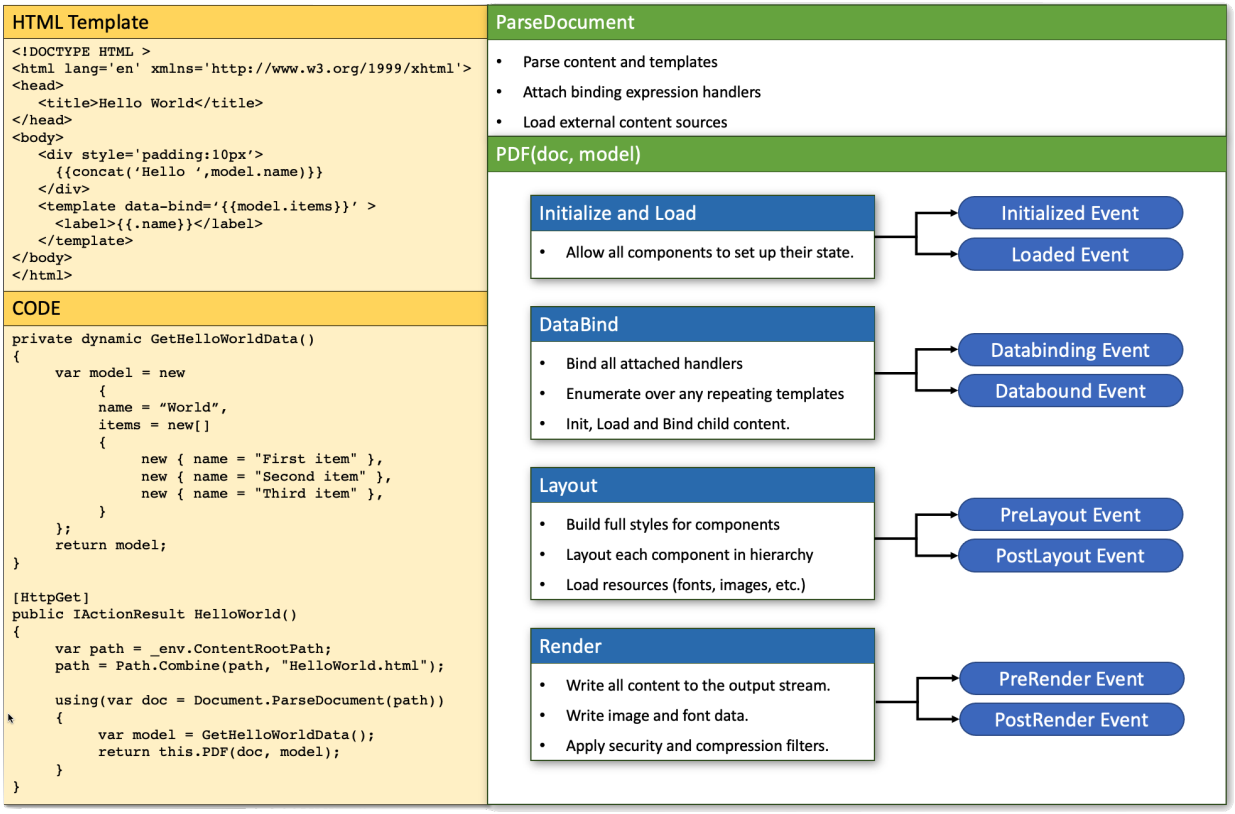
10.79 Binding to events - TD

10.79.1 6.14.1. Document Processing lifecycle

- Parsing the template creates the document object model.
- Initialize and load on each component to ensure the correct state.
- Databind to any data models (which can create further components)
- Layout converts the high level components to lower level entities
- Render allows the layout entities to render themselves to a PDFWriter

Between parsing and initializing is a neat point to add your own content, or add any model(s) needed to the document, along with setting up any events or custom code.

Each of the stages raises events that can be captured to perform any custom processing required



Full size version

10.80 Controllers for your templates - PD

Sometimes it's just not quite enough to give the data and render the output. More control is needed over altering the content.

Scryber supports this through the use of code controllers, which can be attached to files through the scryber processing instruction. And have properties set or methods called during the document lifecycle.

10.80.1 Template with controller

```
<?scryber controller='ControllerNamespace.MyController, MyAssembly' ?>
<html xmlns='http://www.w3.org/1999/xhtml' id='MyDocument' >
<head>
    <title>HTML Document</title>
    <style>
        .grey{ background-color: grey; }
    </style>
</head>

<body class="grey" title="Page 1">
    <p id='DocPara' title="Inner">Hello World, from scryber.</p>
    <img on-databound='UpdateImagePath' src='{@:HeaderImageName}' />
    <div style='padding:20pt'>
        <img on-databound='UpdateImagePath' src='{@:ContentImageName}' />
    </div>
</body>
</html>
```

10.80.2 And the controller code

```
using System;
using Scryber;
using Scryber.Components;
using Scryber.Html.Components;

namespace ControllerNamespace
{
    // A new instance of the class will be created for each document generation.
    // But the classes must have a parameterless constructor.

    public class MyController
    {
        // This will be set to the instance with id 'MyDocument'
        // As it is required, then an error will be raised if there
        // is no HTMLDocument with the specified id - so you can be sure it is set.

        [PDFOutlet("MyDocument", Required=true)]
        public HTMLDocument Document { get; set; }

        // This property will be set to a paragraph with id 'DocPara'
        // as there is no explicit name on the outlet attribute.
        // If a paragraph is not found with this id, it will be null
        // but not an error.

        [PDFOutlet]
        public HTMLPara DocPara { get; set; }


        // constructors need to be parameterless
        // scrybere does not currently support dependency injection,
        // but can use the document params collection to store anything needed

        public MyController()
```

(continues on next page)

(continued from previous page)

```
        {
        }


        // This action will be called after the images have been databound
        // So we know the src in the template will have been set.

        [PDFAction]
        public void UpdateImagePath(object sender, PDFDataBindEventArgs args)
        {
            //Do whatever is needed here to update the component.
            var img = sender as HTMLImage;

            //We can access the document from the property
            var path = this.Document.Params["basePath"] as string;

            //Update the image source, which has already been data bound.
            img.Source = "/" + path +"/" + img.Source + ".png";
        }
    }
}
```

10.80.3 Outlets and Actions

Scryber uses an opt-in approach to controllers. This allows the re-use of other classes and makes sure the content being served is wanted.

A PDFOutlet is on property that will be set just after parsing of a template and controller instantiation. It can be strongly typed, but as long as the referencing element in the template can be assigned it will be.

By default it the parser finds a componet with the same ID as an outlet property it will try to assign the coponent to that propoerty.

There are 2 attribute customizers that can be used to alter behaviour

1. **Name** - If set to a string value, then that will be the ID of the component to use (rather than defaulting to the name of the actual Property itself.
2. **Required** - This is false by default, but if set to true, then if the outlet is not assigned during parsing of the template an error will be raised.

Once set then the instance can be used an manipulated at run time however is seen fit.

A PDFAction is a method that is called during the processing of a component from initialize to load to binding to render that can used to change the content or output of the document. New content can be added, or specific content removed. What ever is needed.

All actions have their own specific signature, but follow the standard .net event handling mechanism. (more below)

There are again 2 attribute customizers available to alter behaviour.

1. **Name** - If set the name that will be looked for on a template event attribute in preference to the actual method name.
2. **IsAction** - This is by default true, so can be used. If it's set to false (for example by an overriding class method, then it will be ignored.

Actions can be called more than once, and can also be called inside repeating templates.

10.80.4 Event calling pipeline

Scryber has a full event piplene that can be used at any stage in the document lifecycle. All template handler attributes start with on-xxxx and are available on all element tags

1. **on-init** - Will be called at the very start, with the sender as the registered receiver and a PDFInitEventArgs instance. The full document structure many not be in place by this point.
2. **on-load** - Will be called once all the document has be parsed and has the heirarchy in place, but not databound.
3. **on-databind** - Will be called on each component in turn before any databinding statements are executed. e.g. {@:MyValue} will still be unset.
4. **on-itemdatabound** - Will be called by a template each time a new item is databound in the content, passing the item that has been created as well as the context.
5. **on-databound** - Will be called on each component in turn after any databinding statements have been executed and their values set.
6. **on-prelayout** - Will be the last chance to inject any content into the document graph before it is converted to an explicit page layout.
7. **on-postlayout** - Will be called with the actual content measured and laid out into eplicit pages, blocks, regions, lines and runs.
8. **on-prerender** - Will be called before the layout is output to a stream with the right structure.
9. **on-postrender** - Will be called after everything is done and rendered.

Some of the most opportune times to capture events are

on-init or **on-load** for the document, to prepare anything you may need as a controller.

on-itemdatabound for a template, so other content can be added or set up based on the context.

on-databound or **on-prelayout** for a component with dynamic content, that can be adjusted before laying out.

on-postrender for the document, so any resources can be cleaned up and/or disposed.

10.80.5 Event Method Signatures

As mentioned all the events follow the standard .net event method signature, each with specific arguments based on the type. All the arguments contain at least a reference to the current context which can be specific to the pipeline but will as a minimum contain the following

- Document - As an IPDFDocument which has been parsed.
- Items - The **currrent** collection of Items that been assigned as parameters on the document or explicitly set.
- TraceLog - For addign messages and statements to the log output
- PerformanceMonitor - For capturing specific performance metrics
- OutputFormat - Can only bbe PDF
- ConformanceMode - Should errors be raised as exceptions, or logged.

Initialize has the PDFInitEventArgs with the PDFInitContext.

```
[PDFAction("init-para")]
public void ParagraphInit(object sender, PDFInitEventArgs args)
{
    (sender as HTMLParagraph).Contents.Add(new Label() { Text = "We are initialized",␣
↪StyleClass = "block"});
```

(continues on next page)

(continued from previous page)

```
    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Initialized the␣
↪paragraph");
}
```

Load has the PDFLoadEventArgs with the PDFLoadContext.

```
[PDFAction("load-para")]
public void ParagraphLoad(object sender, PDFLoadEventArgs args)
{
    (sender as HTMLParagraph).Contents.Add(new Label() { Text = "We have loaded",␣
↪StyleClass = "block"});
    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Loaded the paragraph
↪");
}
```

DataBinding and DataBound have the PDFDataEventArgs with the PDFDataContext, which is far more interesting. The PDFDataContext also has the current DataStack, CurrentIndex and a namespace resolver for inner parsing if needed.

The DataStack is a stack of the objects and IDataSource implementors that is consistent across all binding calls. It is possible to push data onto the stack in the DataBind method, and pop it off after on the databound method. This gives complete control over what children will use for binding at runtime, even without setting an explict document model parameters.

```
[PDFAction("bind-para")]
public void ParagraphBinding(object sender, PDFDataBindEventArgs args)
{
    (sender as HTMLParagraph).Contents.Add(new Label() { Text = "We are binding",␣
↪StyleClass = "block"});
    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Binding the␣
↪paragraph");
}

[PDFAction("bound-para")]
public void ParagraphBound(object sender, PDFDataBindEventArgs args)
{
    (sender as HTMLParagraph).Contents.Add(new Label() { Text = "We have bound",␣
↪StyleClass = "block"});
    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Bound the paragraph
↪");
}
```

The Layout and render event handlers are more useful for component developers as they can really affect the quality of output, but are documented here for completeness. They Layout contains a reference to the current document layout along with the PDFGraphics and the PDFStyleStack. And the PDFRenderContext contains the current rendering pages, offsets and sizes along with the PDFGraphics and PDFStyleStack.

```
[PDFAction("pre-layout-para")]
public void ParagraphPreLayout(object sender, PDFLayoutEventArgs args)
{
    (sender as HTMLParagraph).Contents.Add(new Label() { Text = "We are laying out",␣
↪StyleClass = "block"});
    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Laying-out the␣
↪paragraph");
}
```

(continues on next page)

(continued from previous page)

```
[PDFAction("post-layout-para")]
public void ParagraphPostLayout(object sender, PDFLayoutEventArgs args)
{
    //This label will not appear as we have finished using the components
    (sender as HTMLParagraph).Contents.Add(new Label() { Text = "We have been laid out
↪", StyleClass = "block"});

    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Laid-out the␣
↪paragraph");
}


[PDFAction("pre-render-para")]
public void ParagraphPreRender(object sender, PDFRenderEventArgs args)
{
    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Rendering the␣
↪paragraph");
}

[PDFAction("post-render-para")]
public void ParagraphBinding(object sender, PDFRenderEventArgs args)
{
    args.Context.TraceLog.Add(TraceLevel.Message, "Custom Code", "Rendered the␣
↪paragraph");
}
```

10.80.6 Adding to a template

If we apply the methods above to a template with our controller specified

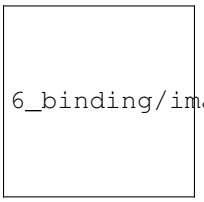
```
<?scryber append-log='true' controller='Scryber.Core.UnitTests.Mocks.
↪GenericControllerClass, Scryber.UnitTests' ?>
<!DOCTYPE HTML>
<html xmlns='http://www.w3.org/1999/xhtml' id="MyDocument" on-init="Initialized">
<head>
    <title></title>
    <style type="text/css">

        body {
            font-size: 14pt;
        }

        .block{
            display:block;
            border:solid 1px blue;
            padding:5pt;
            margin-bottom: 5pt;
            width:100%;
        }

    </style>
</head>
<body style="padding:20pt">
    <p on-init="init-para" on-loaded="load-para"
    on-databinding="bind-para" on-databound="bound-para"
```

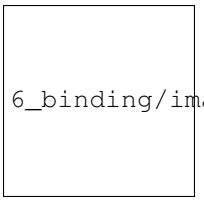
(continues on next page)

(continued from previous page)

```
    on-prelayout="pre-layout-para" on-post-layout="post-layout-para"
    on-prerender="pre-render-para" on-postrender="post-render-para"></p>
</body>
</html>
```

We can see the output in the page up to the point of layout and the messages in the log.





10.80.7 Dependency Injection

The controller must have a parameterless constructor, but if access to other instances and services is needed, they can be passed to the document and then used on the controller.

```
//document parsing

var doc = Document.ParseTemplate("path.html");
doc.Params["DataService"] = GetDataService();

doc.SaveAsPDF("Path.pdf");
```

```
[PDFAction("load-doc")]
public void DocumentLoaded(object sender, PDFLoadEventArgs args)
{
    PDFDocument doc = (PDFDocument)args.Document;
    this.DataService = (MyDataService)doc.Params["DataService"];

    //Do what ever else is needed.
}
```

10.80.8 Events inside a <template>

The **on-item-databound** event will be called each and every time a template creates and binds the inner content. Any events registered within the template, on components, will be raised for each and every component.

```
<?scryber controller='ControllerNamespace.MyController, MyAssembly' ?>
<html xmlns='http://www.w3.org/1999/xhtml' id='MyDocument' >
<head>
    <title>HTML Document</title>
    <style>
```

(continues on next page)

(continued from previous page)

```
        .grey{ background-color: grey; }
    </style>
</head>

<body class="grey" title="Page 1">

    <!-- The template item binding event will be bound for each of the items -->
    <template data-bind='{@:AllItems}' on-item-databound='template-item-bound" >

        <!-- The image will be bound for each of the items -->
        <img on-databound='image-item-bound' src='{@:ContentImageName}' />
    </div>
</body>
</html>
```

10.81 Binding Performance and style caching - PD

Scryber does a number of activities underneath to improve performance and reuse cached data where possible. Downloaded font-files are kept locally Images can be cached and re-used, and only one reference is used in the files.

It also caches the full stype of individual elements. Including those in templates. Appending the log file can also give great insight into the

If for example there is a need to output a 10,000 row listing to PDF then we can create a template to do that and inject the Model.

```
<?scryber append-log='true' log-level='messages' ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml' title="Root">
<head>
    <title>Large File Test Document</title>
    <style>
        .grey{ background-color: grey; padding: 20pt}
        td.key{ color: #333; font-size:10pt;}
        td.value{  color: #333; font-size:10pt; }
        tr.odd{ background-color: #AAA;}
        tr.even{ background-color: #CCC; }
    </style>
</head>
<body class="grey" title="Outer">
    <header>
        <p style="text-align:center; font-size:12pt; padding:5pt; border-bottom:␣
↪solid 1pt red; margin-bottom: 5pt;">Binding large data</p>
    </header>
    <p title="Inner">This is a test of binding the content to a table.</p>
    <div style="column-count: 2">
        <table style="width:100%;">
            <thead style="font-weight:bold;">
                <tr>
                    <td class="key">Name</td>
                    <td class="value" style="width: 120pt">Value</td>
                </tr>
```

(continues on next page)

(continued from previous page)

```
            </thead>
            <!-- our table is built with 2 columns -->
            <template data-bind="{@:model.Items}" >
                <tr class="{@:.Row}">
                    <td class="key">{@:.Key}</td>
                    <td class="value">{@:.Value}</td>
                </tr>
            </template>
        </table>
    </div>
    <footer>
        <p style="text-align:center; font-size: 12pt; padding:5pt; border-top: solid␣
↪1pt red; margin-bottom: 5pt;">
            Page <page /> of <page property="total" />.
        </p>
    </footer>
</body>
</html>
```

```
public void LargeFileTest()
{

    var path = System.Environment.CurrentDirectory;
    path = System.IO.Path.Combine(path, "../../../Content/HTML/LargeFile.html");

    data = new
    {
        Items = GetListItems(10000)
    };
    using (var doc = Document.ParseDocument(path))
    {
        doc.Params["model"] = data;
        using (var stream = DocStreams.GetOutputStream("LargeFile.pdf"))
        {
            doc.SaveAsPDF(stream);
        }

    }
}

private class ListItem
{
    public string Key { get; set; }
    public int Value { get; set; }
    public string Row { get; set; }
}

private static ListItem[] GetListItems(int count)
{
    var mocks = new ListItem[count];

    for (int i = 0; i < count; i++)
    {
        ListItem m = new ListItem() { Key = "Item " + i.ToString(), Value = i, Row =␣
↪(i % 2 == 1) ? "odd" : "even" };
        mocks[i] = m;
```

(continues on next page)

(continued from previous page)

```
    }

    return mocks;
}
```

And the output from this will be 157 pages of lovely tables of content.

6_binding/images/documentStyleCaching.png

As can be seen with the scryber processing instruction in the template, we are appending the trace log tables to this file.

6_binding/images/documentStyleCachingTrace.png

The contents of this show the breakdown of time including the template parsing - 680ms for 10,314 items (table rows, page header and footers).

10.82 Extending Scryber - TD

Content coming soon

10.83 When to declare it

10.84 IFunction interface

10.85 Implementing a function

10.86 When to declare it

10.87 IFunction interface

10.88 Implementing a function

10.89 Scryber Configuration Opions - TD

Content coming soon

10.90 Extending images and image factories - TD

Content coming soon

10.91 Security and encryption options - TD

Content coming soon

10.92 When to declare it

10.93 IFunction interface

10.94 Implementing a function

10.95 Scryber library namespaces - TD

Content to complete

10.96 Namespaces and their Assemblies - PD

Scryber relies on the xml namespaces (xmlns) to identify the classes it should use when parsing an XML or XHTML file. This is based on a mapping of xmlns value to a fully qualified assembly name and namespace.

10.96.1 Declared Namespaces

The 3 base namespaces that are automatically added are:

- **http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Components.xsd**
  - The base level components used in scryber. e.g. Document, TableGrid, List, Span etc.
  - It refers to the Scryber.Components namespace in the Scryber.Components assembly (Version=1.0.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe)
- **http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Data.xsd**
  - The data components used in scryber. e.g. ForEach, Choose, If.
  - It refers to the Scryber.Data namespace in the Scryber.Components assembly (Version=1.0.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe)
- **http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Styles.xsd**
  - The style components and attributes used in scryber. e.g. StyleDefn, StyleGroup, StylesDocument.
  - It refers to the Scryber.Styles namespace in the Scryber.Styles assembly (Version=1.0.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe)

The html and svg namespaces are also automatically added.

- **http://www.w3.org/1999/xhtml**
  - The html components used in scryber. e.g. div, span, section etc.
  - It refers to the Scryber.Html.Components namespace in the Scryber.Components assembly (Version=1.0.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe)
- **http://www.w3.org/2000/svg**
  - The svg drawing components used in scryber. e.g. ellipse, circle, rect etc.
  - It refers to the Scryber.Svg.Components namespace in the Scryber.Components assembly (Version=1.0.0.0, Culture=neutral, PublicKeyToken=872cbeb81db952fe)

Note: If a file or stream of content does not have a namespace, then the classes cannot be found and therefore parsed.

10.96.2 Class attributes

Within the assembly namespaces, referred to above, are actual classes decorated with attributes for each of their properties and contents.

```
namespace Scryber.Html.Components
{
    [PDFParsableComponent("body")]
    public class HTMLBody : Scryber.Components.Section
    {

        [PDFAttribute("class")]
        public override string StyleClass
        {
            get => base.StyleClass;
            set => base.StyleClass = value;
        }

        [PDFAttribute("style")]
        public override Style Style
        {
            get => base.Style;
            set => base.Style = value;
        }

        [PDFElement("")]
        [PDFArray(typeof(Component))]
        public override ComponentList Contents
        {
            get { return base.Contents; }
        }

        [PDFElement("header")]
        [PDFTemplate(IsBlock= true)]
        public override IPDFTemplate Header
        {
            get => base.Header;
            set => base.Header = value;
        }
```

(continues on next page)

(continued from previous page)

```
        [PDFElement("footer")]
        [PDFTemplate(IsBlock = true)]
        public override IPDFTemplate Footer
        {
            get => base.Footer;
            set => base.Footer = value;
        }

        [PDFAttribute("hidden")]
        public string Hidden
        {
            get { return (this.Visible) ? string.Empty : "hidden" }
            set { this.Visible = (string.IsNullOrEmpty(value) || value != "hidden") ?␣
↪true : false; }
        }

        [PDFAttribute("title")]
        public override string OutlineTitle
        {
            get => base.OutlineTitle;
            set => base.OutlineTitle = value;
        }

        public HTMLBody()
            : base()
        {
        }
    }
}
```

Here we can see the html body class decorated with the PDFParsableComponent attribute, so the parser know when it gets to a <body> tag in the content stream in namespace Scryber.Html.Components it create an instance of the HTMLBody class.

The class inherits from the Scryber.Components.Section (an overflowing page), and overrides some of the base functionality to support the standard html attributes. For example the [PDFAttribute("class")] maps the @class attribute in the content stream to the StyleClass string property in the instance.

The explicitly named [PDFElement("head")] if found will be assigned to the Header property, in this case as a template so it can be used multiple times (See binding_model)

Finally the empty PDFElement attribute with the PDFArray attribute tells the parser it should expect inner child components (that are not nested within another element) of type Component, and they should be added to this collection.

Note: Scryber has an explicit parser, rather than implicit. So if classes or properties are not decorated, then they will not be used.

10.96.3 Parsing the content

Considerring the below content we can see the namespace mapping to the classes

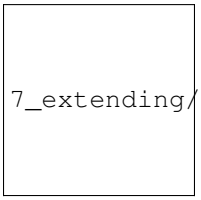
```
<html xmlns='http://www.w3.org/1999/xhtml'>
    <body style='padding:20pt;' title='Top Level' >
        <head><p>This is the header</p><head>
```

(continues on next page)

(continued from previous page)

```
        <p class='main'>This is the content</p>
    </body>
</html>
```

When parsed this will give us an object graph of the below. The content in the header is kept as a string and will be parsed when used each time.



See extending_scryber to understand how to add your own classes and namespaces.

10.97 Paramter types and the model - PD

The document parameters are values that can be set within the template or in code. They are able to hold values used through out the generation process (see document_lifecycle) and are passed via the Context to any event handlers.

10.97.1 Declaring and Using in Documents

Document parameters are declared within the *<Params>* element of Declaring a parameter in a document is not required for it to be used later on, but is best practice. It is more readable and supports default values if they do not change.

When referring to a parameter within your document template use the @: binding syntax (or item: syntax) e.g. *{@:MyParamName}* Then the parser finds a binding reference with this syntax it will create a binding expression that will be evaulated in the document_lifecycle Databinding phase and the value applied to the property it was set on.

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
              xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Styles.xsd" >
    <Params>
        <!-- Declare the parameters -->
        <doc:Bool-Param id="ShowTitle" value="true" />
        <doc:String-Param id="MyTitle" value="Document Title" />
        <doc:Color-Param id="TitleBg" value="#AAAAAA" />
    </Params>

    <Pages>
        <!-- Use the 'MyTitle' parameter for the outline. -->
        <doc:Page outline-title="{@:MyTitle}" styles:margins="20pt" styles:font-size=
↪"12pt">
        <Content>
            <!-- And use it as the text on the heading -->
            <doc:H1 visible="{@:ShowTitle}" styles:bg-color="{@:TitleBg}"  text="
↪{@:MyTitle}" > </doc:H1>
            <doc:Para >This is the content of the document</doc:Para>
        </Content>
```

(continues on next page)

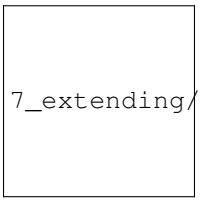
(continued from previous page)

```
        </doc:Page>
    </Pages>

</doc:Document>
```

Generating this document the parameter values are applied to the final output and rendered.

```
[HttpGet]
public IActionResult DocumentParameters()
{
    var path = _rootPath;
    path = System.IO.Path.Combine(path,"Views", "PDF", "DocumentParameters.pdfx");
    var doc = PDFDocument.ParseDocument(path);

    return this.PDF(doc);
}
```

And this then output as follows

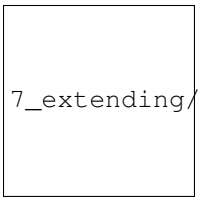


10.97.2 Changing the values

Parameters are evaluated during the data binding stage of a document creation, so once bound, any changes to the parameters will not be evaluated. The best time to change the values of a parameters are once it has been parsed, or on the loaded event.

```
[HttpGet]
public IActionResult DocumentParameters()
{
    var path = _rootPath;
    path = System.IO.Path.Combine(path,"Views", "PDF", "DocumentParameters.pdfx");
    var doc = PDFDocument.ParseDocument(path);

    doc.Params["MyTitle"] = "New Document Title";
    doc.Params["TitleBg"] = new PDFColor(255,0,0);

    return this.PDF(doc);
}
```

Generating the file will then apply the vaules at binding time to the content and rendering will produce the following output.



It is perfectly acceptable to assign a parameter in the document that is not declared, nor does it have to be typed.

e.g.

```
[HttpGet]
public IActionResult DocumentParameters()
{
    var path = _rootPath;
    path = System.IO.Path.Combine(path,"Views", "PDF", "DocumentParameters.pdfx");
    var doc = PDFDocument.ParseDocument(path);

    doc.Params["MyTitle"] = "New Document Title";
    doc.Params["TitleBg"] = new PDFColor(255,0,0);
    //Undeclared parameter
    doc.Params["Size"] = (PDFUnit)30;
    return this.PDF(doc);
}
```

And the used in your template

```
<doc:H1 visible="{@:ShowTitle}" styles:font-size="{@:Size}" styles:bg-color="
↪{@:TitleBg}"  text="{@:MyTitle}" > </doc:H1>
```

But it will not be co-erced into the correct type, nor will it have a clear initial value.

10.97.3 Simple Parameter Types

Scryber is strongly typed. The xml templates are defined as classes in namespaces and assemblies, and so are the **parameter** declarations.

There are a range of types available, and options for using complex types (see below).

- String-Param: Any string value, the default if not set is null.
- Int-Param: Single integer value, the defult if not set is 0.
- Guid-Param: A GUID value, the default is an empty guid.
- Double-Param: Holds double values, the default is 0.0
- Bool-Param: Boolean (True, False) values, the default is false.
- Date-Param: Date and time values, the default is minimum date time and values are culture sensitive.
- Unit-Param: Holds a reference/pdf_unit value, see component_positioning for more info. The default is empty (zero) unit.
- Color-Param: Holds a reference/pdf_color value, the default is transparent.
- Thickness-Param: Holds a reference/pdf_thickness value (used in padding, margins, clipping etc.). The default is empty (zero) thickness.
- Enum-Param: Has a specific *type* attribute that specifies the type of enum that should be stored. The default is null.

There are 3 other parameter types available XML, Template and Object which are discussed later on in this document.

10.97.4 Complex Object Parameters

Whilst Scryber Parameters can be simple types, it also supports complex objects that can be traversed.

Our previous example could have been written with a single parameter rather than the 3 individual ones, an the values retrieved from the properties on that object.

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
            xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Styles.xsd" >
    <Params>
        <!-- Declare the parameters -->
        <doc:Object-Param id="Heading" />
    </Params>

    <Pages>
        <!-- Use the 'MyTitle' parameter for the outline. -->
        <doc:Page outline-title="{@:Heading.Title}" styles:margins="20pt" styles:font-
↪size="12pt">
        <Content>
            <!-- And use it as the text on the heading -->
            <doc:H1 visible="{@:Heading.Visible}" styles:bg-color="{@:Heading.
↪Background}"  text="{@:Heading.Title}" > </doc:H1>
            <doc:Para >This is the content of the document</doc:Para>
        </Content>
        </doc:Page>
    </Pages>

</doc:Document>
```

The dot notation is evaluated at runtime to bind the appropriate value.

```
[HttpGet]
public IActionResult DocumentParameters()
{
    var path = _rootPath;
    path = System.IO.Path.Combine(path,"Views", "PDF", "DocumentParameters.pdfx");
    var doc = PDFDocument.ParseDocument(path);

    //Set the heading param to a new dynamic type.
    doc.Params["Heading"] = new
    {
        Title = "Model Document Title",
        Visible = true,
        Background = "#FF0000"
    };

    return this.PDF(doc);
}
```

It is also possible to stronly type the object parameter by specifying the expected **full** type name, so you can be sure the content coming into the template matches. Inherited types will be acceptable as will interfaces.

```
<Params>
    <!-- Declare the parameters -->
    <doc:Object-Param id="Heading" type="MyNamespace.MyType, MyAssembly" />
</Params>
```

```
doc.Params["Heading"] = new MyNamespace.MyType("Title",true, "#FF0000");
```

Note: If you provide a class that is not assignable to the parameter type a PDFDataException will be raised directly on assignment, so easily troubleshooted.

10.97.5 The MVC model

In the scryber.core.mvc project, there is a special extension method on the controller that accepts not just the document, but also an object as the model. Within this extension method, the *Model* parameter value will directly be assigned, even if it does not exist.

```
[HttpGet]
public IActionResult DocumentParameters()
{
    var path = _rootPath;
    path = System.IO.Path.Combine(path,"Views", "PDF", "DocumentParameters.pdfx");
    var doc = PDFDocument.ParseDocument(path);

    //Set the heading param to a new dynamic type.
    var model = new MyNamespace.MyType("Title",true, "#FF0000");

    return this.PDF(doc, model);
}
```

And in your template, you can specify the model type you are expecting.

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
              xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Styles.xsd" >
    <Params>
        <!-- Declare the parameters -->
        <doc:Object-Param id="Model" type="MyNamespace.MyType, MyAssembly" />
    </Params>

    <Pages>
        <!-- Use the 'MyTitle' parameter for the outline. -->
        <doc:Page outline-title="{@:Model.Title}" styles:margins="20pt" styles:font-
↪size="12pt">
        <Content>
            <!-- And use it as the text on the heading -->
            <doc:H1 visible="{@:Model.Visible}" styles:bg-color="{@:Model.Background}
↪"  text="{@:Model.Title}" > </doc:H1>
            <doc:Para >This is the content of the document</doc:Para>
        </Content>
        </doc:Page>
    </Pages>

</doc:Document>
```

10.97.6 Combining selector paths

The object selectors support complex notation for retrieving values.

- **{@:*dotnotation*} for binding to a paramter passed to the document. This supports complex paths**

  - {@:ParamName} for the direct value.
  - {@:ParamName.Property} for getting a property value.
  - {@:ParamName[n]} for getting the n'th value from an array
  - {@:ParamName['key']} for geting a dictionary value based on key.
- **The statements can be chained together as long as needed.**
  - {@:Model.Property[0].Property['key'].Value}
  - If one of the properties evaluates to null, then the chain will no longer be evaluated, and no value will be set.

10.97.7 Binding to Collections

With complex objects it is possible to bind to object arrays or any other type of collection. The object that is extracted from the collection at that time will become the current *context*.

To refer to the properties in the current context simply precede the property with a dot (.)

```
<?xml version='1.0' encoding='utf-8' ?>
    <doc:Document xmlns:doc = 'http://www.scryber.co.uk/schemas/core/release/v1/
↪Scryber.Components.xsd'
                  xmlns:styles = 'http://www.scryber.co.uk/schemas/core/release/v1/
↪Scryber.Styles.xsd'
                  xmlns:data = 'http://www.scryber.co.uk/schemas/core/release/v1/
↪Scryber.Data.xsd'
                      >
    <Params>
        <doc:Object-Param id='Model' ></doc:Object-Param>
    </Params>

    <Pages>

    <doc:Section>
        <Content>

            <data:ForEach value='{@:Model.List}' >
                <Template>
                    <!-- Here we can refer to the current object and set values from␣
↪properties. -->
                    <doc:Label id='{@:.Id}' text='{@:.Name}' ></doc:Label>
                    <doc:Br/>
                </Template>
            </data:ForEach>

        </Content>
    </doc:Section>

    </Pages>
</doc:Document>
```

And when we are providing the value we can add an array or list.

```
doc.Params["Model"] = new
{
    Color = Scryber.Drawing.PDFColors.Aqua,
```

(continues on next page)

(continued from previous page)

```
    List = new[] {
        new { Name = "First", Id = "FirstID"},
        new { Name = "Second", Id = "SecondID" }
    }
};
```

For more on looping through content and the available data components see document_databinding

10.97.8 Binding Styles to Parameters

As styles are full qualified members of the document object, they also support databinding to values.

```
<?xml version='1.0' encoding='utf-8' ?>
<doc:Document xmlns:doc = 'http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd'
              xmlns:styles = 'http://www.scryber.co.uk/schemas/core/release/v1/
↪Scryber.Styles.xsd'
              xmlns:data = 'http://www.scryber.co.uk/schemas/core/release/v1/
↪Scryber.Data.xsd' >
<Params>
    <doc:Object-Param id='Model' ></doc:Object-Param>
</Params>

<Styles>
    <!-- Bind the head and body styles to the Theme -->
    <styles:Style applied-class='head'>
    <styles:Padding all='20pt'/>
    <styles:Background color='{@:Model.Theme.TitleBg}' />
    <styles:Fill color='{@:Model.Theme.TitleColor}'/>
    <styles:Font family='{@:Model.Theme.TitleFont}' bold='false' italic='false' />
    </styles:Style>

    <styles:Style applied-class='body'>
    <styles:Font family='{@:Model.Theme.BodyFont}' size='{@:Model.Theme.BodySize}' />
    <styles:Fill color='#333300'/>
    <styles:Padding all='20pt'/>
    </styles:Style>

</Styles>

<Pages>

    <doc:Section>
    <Content>
        <!-- Specify the class names on the components to use the styles -->
        <doc:H1 styles:class='head' text='{@:Model.Title}' ></doc:H1>
        <doc:Div styles:class='body' >
        <!-- and then loop through -->
        <data:ForEach value='{@:Model.List}' >
            <Template>
            <doc:Label id='{@:.Id}' text='{@:.Name}' ></doc:Label>
            <doc:Br/>
            </Template>
        </data:ForEach>
        </doc:Div>
```

(continues on next page)

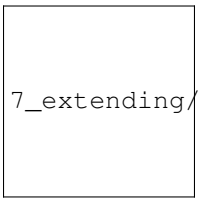
(continued from previous page)

```
    </Content>
    </doc:Section>

</Pages>
</doc:Document>
```

And we can generate this content by providing the Theme as well as the List.

```
[HttpGet]
public IActionResult DocumentStyleParameters()
{
    var path = _rootPath;
    path = System.IO.Path.Combine(path, "Views", "PDF", "DocumentStyleParameters.pdfx
↪");
    var doc = PDFDocument.ParseDocument(path);

    var model = new
    {
        Title = "This is the document title",
        List = new[] {
            new { Name = "First", Id = "FirstID" },
            new { Name = "Second", Id = "SecondID" }
        },
        Theme = new
        {
            TitleBg = new PDFColor(1,0,0),
            TitleColor = new PDFColor(1,1,1),
            TitleFont = "Segoe UI Light",
            BodyFont = "Segoe UI",
            BodySize = (PDFUnit)12
        }
    };

    return this.PDF(doc, model);
}
```

These styles should then be used in the creation of the document



7_extending/images/documentstyleparameters.png

Very quickly our document complexity can grow and then it becomes more important to split the data from the content, and we can do that using the document_datasources and document_controllers

10.97.9 XML parameters

Along with the object parameters, scryber supports the use of XML as a parameter. These are just as poweful as objects.

The xml data parameter, similar to the object parameter supports full xpath deep binding, and functions such as substring and concat. For more details on the xpath syntax see the document_datasources

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
              xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/
↪Scryber.Styles.xsd"
          xmlns:data="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Data.
↪xsd">
<Params>
    <!-- Declare the parameters -->
    <doc:String-Param id="MyTitle" value="Document Title" />

    <!-- This is the xml content that will be used by default -->
    <doc:Xml-Param id="MyData" >
    <Root>
        <Entry id="First">First Name</Entry>
        <Entry id="Second">Second Name</Entry>
        <Entry id="Third">Third Name</Entry>
    </Root>
    </doc:Xml-Param>

</Params>

<Pages>
    <!-- Use the 'MyTitle' parameter for the outline. -->
    <doc:Page outline-title="{@:MyTitle}" styles:margins="20pt" styles:font-size="12pt
↪">
    <Content>
        <!-- And use it as the text on the heading with a visble flag and background -
↪->
        <doc:H1  text="{@:MyTitle}" > </doc:H1>
        <doc:Para >This is the content of the xml document</doc:Para>

        <doc:Ul>
            <!-- Now bind the content of the MyData parameter into a foreach, with␣
↪the selector of //Root/Entry
                to loop through each one in turn -->
            <data:ForEach value="{@:MyData}" select="//Root/Entry" >
            <Template>
                <doc:Li >
                <doc:Text value="{xpath:text()}" />
                </doc:Li>
            </Template>
            </data:ForEach>
        </doc:Ul>

    </Content>
    </doc:Page>
</Pages>

</doc:Document>
```

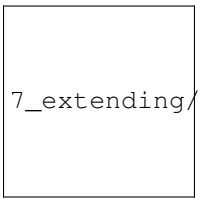
If we generate this content as is the xml will be bound to the unordered list and created.

```
[HttpGet]
public IActionResult DocumentXmlParameters()
{
    var path = _rootPath;
```

(continues on next page)

(continued from previous page)

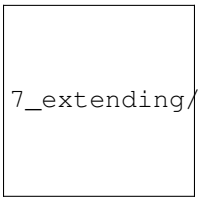
```
    path = System.IO.Path.Combine(path, "Views", "PDF", "DocumentXmlParameters.pdfx");
    var doc = PDFDocument.ParseDocument(path);

    doc.Params["MyTitle"] = "New Document Title";

    return this.PDF(doc);
}
```



By using the xml data as a template we can generate this dynamically too, or load it from a file, or pull from a service. The xml parameter will accept XmlNode values, XPathNavigators, and Linq XElements for values, along with strings.

```
[HttpGet]
public IActionResult DocumentXmlParameters()
{
    var path = _rootPath;
    path = System.IO.Path.Combine(path, "Views", "PDF", "DocumentXmlParameters.pdfx");
    var doc = PDFDocument.ParseDocument(path);

    doc.Params["MyTitle"] = "Xml Document Title";

    //Replace the xml content in the MyData parameter
    var ele = new XElement("Root",
        new XElement("Entry", new XAttribute("id", "Fourth"), new XText("Fourth Name
↪")),
        new XElement("Entry", new XAttribute("id", "Fifth"), new XText("Fifth Name")),
        new XElement("Entry", new XAttribute("id", "Sixth"), new XText("Sixth Name"))
        );
    doc.Params["MyData"] = ele;

    return this.PDF(doc);
}
```

Generating this file again will render the content with the new xml data.



10.97.10 Template Parameters

Along with the XML parameter, scryber supports the Template parameter, which is xml content of scryber components. So you can provide both dynamic data, and dynamic structure to your document at generation time.

```
<?xml version="1.0" encoding="utf-8" ?>
<doc:Document xmlns:doc="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.
↪Components.xsd"
                xmlns:styles="http://www.scryber.co.uk/schemas/core/release/v1/
↪Scryber.Styles.xsd"
            xmlns:data="http://www.scryber.co.uk/schemas/core/release/v1/Scryber.Data.
↪xsd">
<Params>
    <!-- Declare the parameters -->
    <doc:String-Param id="MyTitle" value="Document Title" />

    <!-- This is the xml content that will be used by default -->
    <doc:Xml-Param id="MyData" >
    <Root>
        <Entry id="First">First Name</Entry>
        <Entry id="Second">Second Name</Entry>
        <Entry id="Third">Third Name</Entry>
    </Root>
    </doc:Xml-Param>

    <!-- this is the template content. -->
    <doc:Template-Param id="MyContent" >
        <doc:Li><doc:Text value="{xpath:text()}" /></doc:Li>
    </doc:Template-Param>

</Params>

<Pages>
    <!-- Use the 'MyTitle' parameter for the outline. -->
    <doc:Page outline-title="{@:MyTitle}" styles:margins="20pt" styles:font-size="12pt
↪">
    <Content>
        <!-- And use it as the text on the heading with a visble flag and background -
↪->
        <doc:H1  text="{@:MyTitle}" > </doc:H1>
        <doc:Para >This is the content of the xml document</doc:Para>

        <doc:Ul>
            <!-- Now we specify the template content from the parameter -->
            <data:ForEach value="{@:MyData}" select="//Root/Entry" template="
↪{@:MyContent}" ></data:ForEach>
        </doc:Ul>


    </Content>
    </doc:Page>
</Pages>

</doc:Document>
```

Creating this document at runtime pulls the template data from the parameter *MyContent*

We can then change the value in code to use a different template as well as the xml (including any binding statements).

```
 [HttpGet]
public IActionResult DocumentTemplateParameters()
{
    var path = _rootPath;
```

(continues on next page)

(continued from previous page)

```
    path = System.IO.Path.Combine(path, "Views", "PDF", "DocumentTemplateParameters.
↪pdfx");
    var doc = PDFDocument.ParseDocument(path);

    doc.Params["MyTitle"] = "Xml Document Title";
    var ele = new XElement("Root",
        new XElement("Entry", new XAttribute("id", "Fourth"), new XText("Fourth Name
↪")),
        new XElement("Entry", new XAttribute("id", "Fifth"), new XText("Fifth Name")),
        new XElement("Entry", new XAttribute("id", "Sixth"), new XText("Sixth Name"))
        );
    doc.Params["MyData"] = ele;

    //Just a simple example to change the template.
    doc.Params["MyTemplate"] = "<doc:Li><doc:H1 text='{xpath:text()}' /></doc:Li>";

    return this.PDF(doc);
}
```

The document will then be generated with headings as the content of the list items, rather than just text values.

7_extending/images/documenttemplateparameters.png

The following components support the use of the template attrobute to pull the value from a parameter.

- ForEach (see reference/data_foreach)
- PlaceHolder (see reference/pdf_placeholder)
- DataTemplateColumn (see reference/data_templatecolumn)
- Choose When (see reference/data_choose)
- Choose Otherwise (see reference/data_choose)
- If (see reference/data_if)

10.97.11 Passing parameters to References

The final capability for discussion is the use of the parameters when loading referenced files.

This is discussed in detail in the *Overriding and passing data* section of referencing_files and any type of data can be passed including templates and objects.

It starts to get really fun what you can do!