Version <% { var sharpDocxAssembly = typeof(DocumentBase).Assembly;  
var fvi = System.Diagnostics.FileVersionInfo.GetVersionInfo(sharpDocxAssembly.Location);  
Write(fvi.FileVersion); } %>  
egonl  
May 2020

SharpDocx

Summary

Generating documents with SharpDocx is a two-step process. First you create a view in Word. A view is a Word document which also contains C# code. Code can be inserted anywhere, e.g. <%= DateTime.Now %> would insert the current date and time.

The next step is to create documents based on this view. This requires two lines of code:

var document = DocumentFactory.Create("view.cs.docx");

document.Generate("output.docx");

Out of the box SharpDocx supports inserting text, tables, images and more. This tutorial shows you how.

If you want, you can specify a view model to be used in your view. Then you could write things like < % foreach (var item in Model.MyList) { % >. See the Model sample.

If you want to do something that's not supported by SharpDocx, you can do so by creating your own document subclass. See the Inheritance example.

SharpDocx is inspired by Web technologies like ASP.NET and JSP. Developers familiar with those technologies should feel right at home.

Generating documents with SharpDocx can be very fast: a slightly modified Model sample produced 25 documents per second on my modest laptop. That’s 1500 documents per minute. Single threaded.

Contents

[The basics 3](#_Toc41517073)

[The Write method 3](#_Toc41517074)

[Conditional content 4](#_Toc41517075)

[Text block limitations 4](#_Toc41517076)

[Loops 5](#_Toc41517077)

[Nested loops 6](#_Toc41517078)

[Loops, tables and the AppendRow method 7](#_Toc41517079)

[Combining loops, text blocks and tables 8](#_Toc41517080)

[Images 9](#_Toc41517081)

[Replacing text 10](#_Toc41517082)

[Referencing assemblies and importing namespaces 11](#_Toc41517083)

[The Map 12](#_Toc41517084)

[The SharpDocx solution 13](#_Toc41517085)

# The basics

At any point in the text you can insert C# statements. Like right <% var i = 1; %>here.

The result looks like this:



## The Write method

If you want to display the value of i, you can use the Write method. Right now, i is <% Write(i); %>.

This will show:



There’s also a shorthand notation for the Write method: i is still <%= i %>.

This results in:



You can insert line breaks by using ‘\n’:

<%= "This paragraph\ncontains two\nline breaks" %>.

<% /\* Which renders like this:



**Note:**  this text is commented out, so it won’t appear in the generated document.\*/ %>

# Conditional content

You can use an if statement to display conditional content.

<% if (true) { Write("This **will** *be* displayed."); } %>

<% /\* The result:

 \*/ %>

In this case, any formatting will be lost because the code parser ignores any formatting.

If you want to conditionally display a paragraph with formatting, use a *text block*: text between two code blocks and placed between curly brackets:

<% if (true) { %>This **will** *also* be displayed, but *with* formatting.<% } %>

<% /\* This will show something like:

 \*/ %>

If you want, you can span multiple elements in a text block. E.g.

<% if (true) { %>

The diverging pronunciation of tomato (though not so much potato) is primarily one of regional dialect.

The pronunciation 'tuh-MAH-toh' is the standard pronunciation in the UK and is accepted in the US regions of New England along with parts of the lower East Coast, while 'tuh-MAY-toh' is found almost everywhere else.

<% } %>

## Text block limitations

1. You can’t use text blocks to conditionally display a part of a paragraph. It’s all or nothing. This makes the text block implementation much simpler. However, it might also give some unexpected results.

This is a paragraph.<% if (false) { %> It will be deleted *entirely*. <% } %>.

2. Text blocks can’t share paragraphs. That means that you can’t write < % } } % > to end two text blocks. Instead, use two paragraphs, each containing < % } % >.

3. Also, text blocks in else statements are at the moment not supported by SharpDocx. Instead, use another if statement.

# Loops

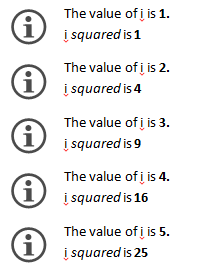
You can add repeating text blocks to a document like this:

<% for (i = 1; i <= 12; ++i) { %>

C:\Users\la_113288\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Infobox_info_icon.emfThe value of i is **<%= i %>.**i *squared* is **<%= i \* i %>**

<% } %>

<% /\* The result:

 \*/ %>

Nested loops  
Loops can also be nested.

<% for (i = 1; i <= 4; ++i) { %>

### Multiples of <%= i %>

<% for (int j = 1; j <= 3; ++j) { %>

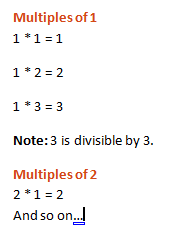
<%= i %> \* <%= j %> = <%= i \* j %>

<% if (i \* j % 3 == 0) { %>**Note:** <%= i \* j %> is divisible by 3.<% } %>

<% } %>

<% } %>

<% /\* This will show something like:

 \*/ %>

# Loops, tables and the AppendRow method

Sometimes you do want a loop, but you don’t want a repeating text block. For example, you just want to append rows to a table, but you don’t want to repeat the table itself. In this case, use {! instead of {.

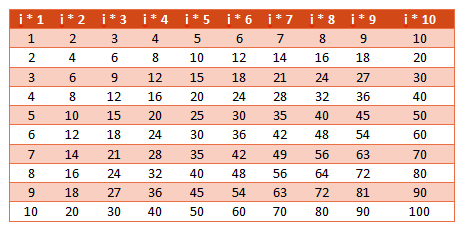
<% for (i = 1; i <= 30; ++i) {! %>

This text and table do NOT repeat, because we used {!. However, a couple of rows do get appended to the table by using the AppendRow method.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i \* 1 | i \* 2 | i \* 3 | i \* 4 | i \* 5 | i \* 6 | i \* 7 | i \* 8 | i \* 9 | i \* 10 |
| <%= i %> | <%= i \* 2 %> | <%= i \* 3 %> | <%= i \* 4 %> | <%= i \* 5 %> | <%= i \*6 %> | <%= i \* 7 %> | <%= i \* 8 %> | <%= i \* 9 %> | <%= i \* 10 %> <% AppendRow(); %> |

<% } %>

<% /\* This renders something like:

 \*/ %>

## Combining loops, text blocks and tables

You can nest tables in text blocks in order to create multiple tables. Note that the inner loop does not create a repeating text block, but does append rows.

<% for (i = 1; i <= 3; ++i) { %>

### Multiples of <%= i %>

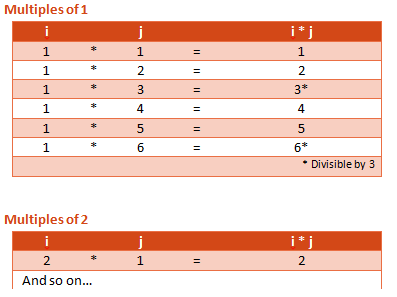
<% for (int j = 1; j <= 6; ++j) {! %>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| i |  | j |  | i \* j |
| <%= i %> | \* | <%= j %> | = | <%= i \* j %><% if (i \* j % 3 == 0) Write("\*"); AppendRow(); %> |
|  |  |  |  | \* Divisible by 3 |

<% } %>

<% } %>

<% /\* This results in something like:

\*/ %>

# Images

Insert images using the Image method.

<% Image("test1.png"); %>

<% /\* The result:

\*/ %>

If only a file name is specified, SharpDocx searches this file in a directory specified by the ImageDirectory property. Right now this property has been set to ‘<%= ImageDirectory %>’.

The Image method accepts a second optional parameter that specifies the relative size of the image. Here’s <% Image("test1.png", 15); %> at 15%.

Images that are too wide to be displayed at 100% are automatically scaled back. Here’s an example:

<% Image("test2.png"); %>

# Replacing text

If you want to replace text, you can use the Replace method.

<% Replace("{text to replace}", "replaced text"); %>

This will replace *all* occurrences of the specified string.[[1]](#footnote-2)

Here’s the **{text to replace}**. And here’s some more {text *to* **replace**}.

<% /\* The result:

 \*/ %>

<%

// Uncomment the next line if you’re adventurous.

// Replace("e", "é");

%>

# Referencing assemblies and importing namespaces

If you want to use specific types in a view, you can use the Assembly and Import directives to get access to them. Directives look like regular code blocks, but they always start with < %@.

Reference an assembly with the Assembly directive.

<%@ Assembly Name="System.Xml" %>  
<%@ Assembly Name="System.Xml.Linq" %>

Import namespaces with the Import directive.

<%@ Import Namespace="System.Xml.Linq" %>

In C# you would write:

using System.Xml.Linq;

Now we can use types in System.Xml.Linq. Let’s read some news.

<% try  
{  
 var atom = XDocument.Load("http://rss.slashdot.org/Slashdot/slashdotMainatom");

foreach (var entry in atom.Descendants("{http://www.w3.org/2005/Atom}entry"))  
 { %>

**<%=** **entry.Element("{http://www.w3.org/2005/Atom}title").Value %>**<%= entry.Element("{http://www.w3.org/2005/Atom}summary").Value.Substring(0,200) %>…

<% }  
}  
catch (Exception ex)  
{   
 Write("Error: " + ex.Message);   
} %>

In a real world scenario you wouldn’t fetch data or have this much code in a view. But hey, this is just an example.

# The Map

The Map maps OpenXmlElements to plain text and vice versa. It’s being used internally by the Replace method and for finding the C# code in views, among other things. At the moment Map.Text looks something like this:

<%= Map.Text.Substring(0,500) %> …

The Map might be handy when you want to search the document for text.

# The SharpDocx solution

### Building the example programs

The Tutorial, Inheritance and Model samples will by default be build for .NET Framework 3.5, 4.5 and .NET Core 2.0. The .NET Core 2.0 builds will use the .NET Standard 2.0 version of SharpDocx.

The Tutorial and Inheritance samples will by default run in .NET Framework 4.5. The Model sample will by default run in .NET Core 2.0. If you want to change this, right click on the solution file in Visual Studio and select **Edit SampleName.csproj**. This will open the csproj-file. The first target named on this line will be used for startup/debugging in Visual Studio:

<TargetFrameworks>net45;net35;netcoreapp2.0</TargetFrameworks

### Unix-like systems

On Unix-like systems, remove the net45 and net35 targets from all projects, because they are only available on Windows. Other than that, the library and samples should compile and run fine.

1. Actually, this will only replace text in the body of the document, and not in headers, footers, end- or footnotes. So this <% Replace("footnotes", "FOOTNOTES"); %>won’t work as expected. But you <%= "c" + "a" + "n" %> use code here. [↑](#footnote-ref-2)