

User Guides

In this section we cover the concepts that you need to know in order to use FlexCel effectively.

The documents here are not a full reference for every type and method in FlexCel: If you are looking for the reference documentation in the methods and classes, you can find it in the [API Documentation](#) tab.

In this section:

Installation Guide

Because before using FlexCel you need to install it.

Getting started

An introduction to start using FlexCel quickly.

API Developer Guide

How to use the FlexCel API to read and write Excel files.

Reports Developer Guide

How to create Excel files by replacing tags in a template.

Reports Designer Guide

How to design a template to be used with FlexCelReport.

Reports Tag Reference

A list with all the tags available in a report.

Performance Guide

How to get the most of FlexCel performance wise.

PDF Exporting Guide

How to use FlexCel to export Excel files to PDF.

HTML Exporting Guide

How to use FlexCel to export Excel files to HTML.

iOS Guide

Special considerations when working in iOS.

Android Guide

Special considerations when working in Android.

Windows Phone, Store and Universal apps Guide

How to use FlexCel inside Universal apps, and the limitations on it.

Mono Guide

How to use FlexCel with Mono.

.NET Core

Using FlexCel with .NET Core.

FlexCel Installation Guide

Choosing how to install FlexCel

You can install FlexCel in 2 different ways:

1. If installing in Windows, you can just run the setup.exe downloaded from our website. It will install all the examples, the dlls, the NuGet packages and the docs.
2. If installing in macOS, we don't have an automatic setup, but you can download the NuGet packages from our website, register them, and use them. You should still install the full FlexCel with setup.exe in a Windows machine to be able to see and run all the examples.

Registering the FlexCel NuGet packages

If you installed FlexCel via setup.exe, then you don't need to do anything else: Setup should have registered the FlexCel NuGet packages for you. If you downloaded the NuGet packages directly, then you need to put them in a **private NuGet repository**.

IMPORTANT

Because FlexCel is not open source, the FlexCel NuGet packages must be put inside a **private** repository that only people with a license can access it. Please don't put them in a public repository like nuget.org.

There are many ways to put FlexCel in a private repository, and depending on your needs you might find some better than others. You can read about it here: <https://docs.microsoft.com/en-us/nuget/hosting-packages/overview>

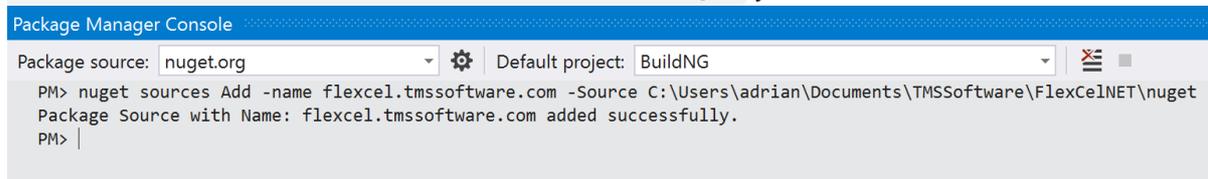
However, the simplest way is just to add the folder where the FlexCel NuGet packages are to the NuGet feeds. Even when <https://docs.microsoft.com/en-us/nuget/hosting-packages/overview> mentions putting them in a network share, **there is no need to share the folder**. You can use a simple not shared folder, and this is what the FlexCel Setup does. To do it manually when you are not using Setup.exe follow the steps below:

1. If you want to install from inside Visual Studio, go to Menu->Tools->NuGet Package Manager->Package Manager Console. If you don't have Visual Studio, you can install NuGet following the instructions in <https://docs.microsoft.com/en-us/nuget/guides/install-nuget>
2. Either from the Package Manager console, a command line prompt or a terminal, type:

```
nuget sources Add -name flexcel.tmssoftware.com -Source <path-to-where-the-flexcel-package-is>
```

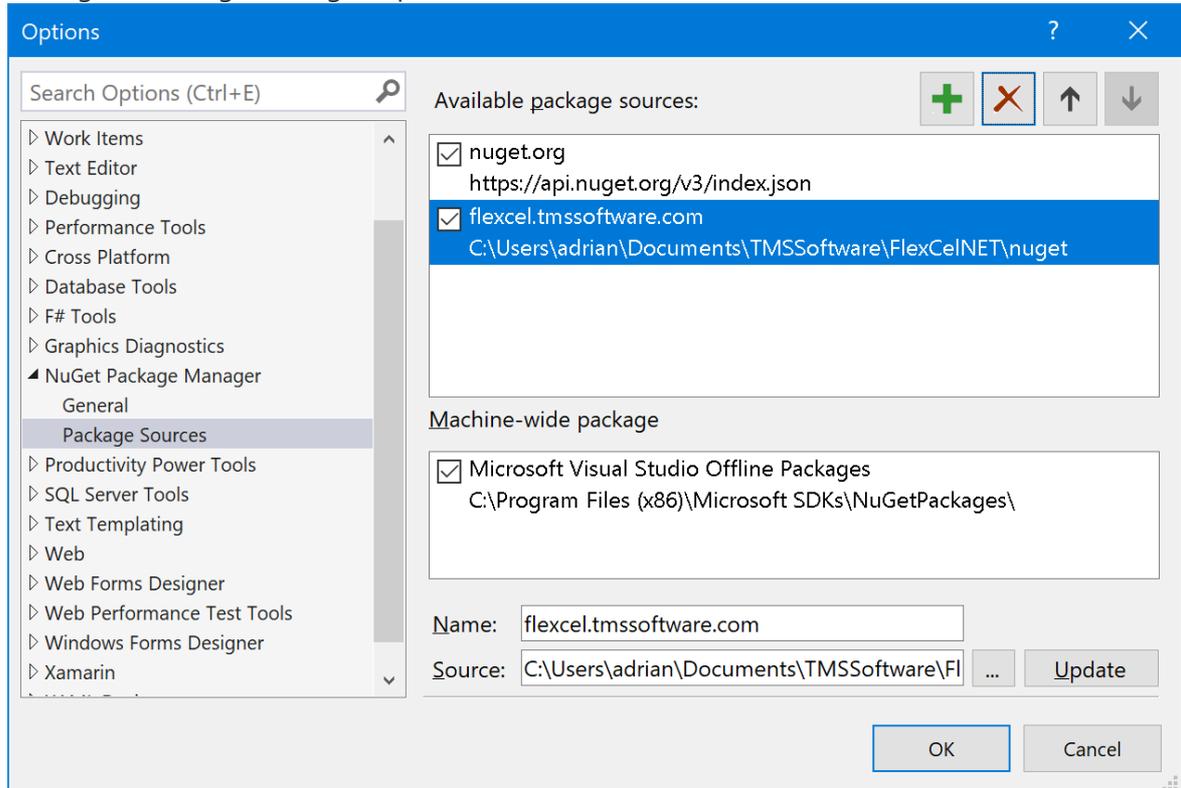
So if for example the FlexCel package is at C:

\\Users\adrian\Documents\TMSSoftware\FlexCelNET\nuget your screen should look as follows:



NOTE

You can also visually manage your package sources by going to Menu->Tools->NuGet Package Manager->Package Manager Options:



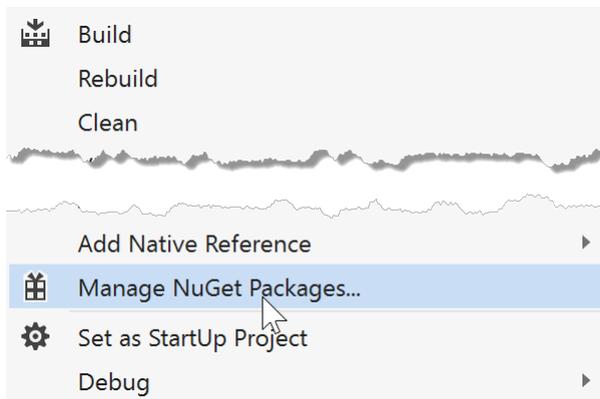
But in this guide we preferred to focus on the command line way, because it allows you to register nuget sources even if Visual Studio is not available.

For uninstalling the source, just go to the package manager options and remove it, or type:

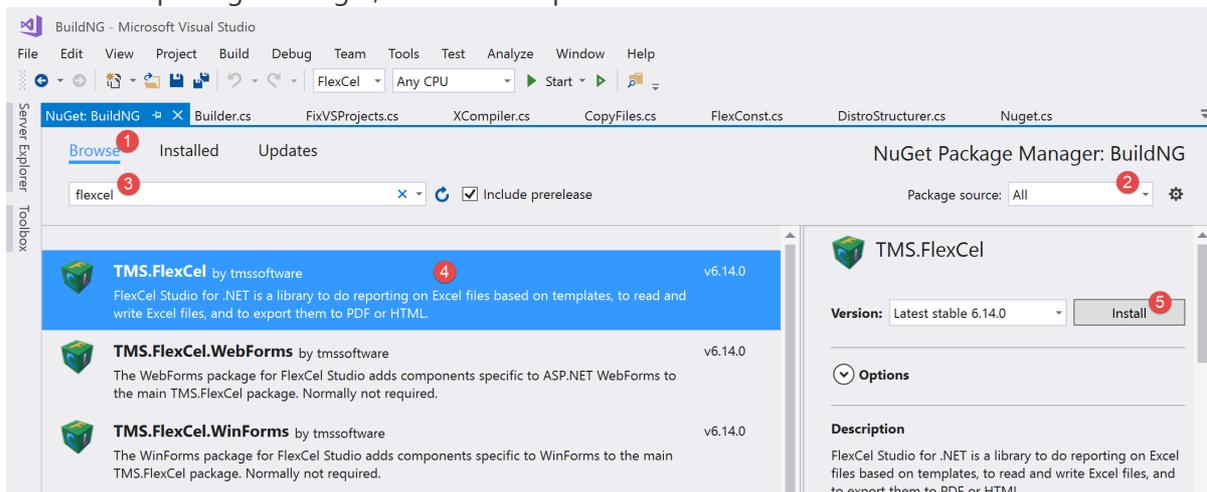
```
nuget sources Remove -name flexcel.tmssoftware.com
```

Referencing the FlexCel NuGet packages

Once you have the nuget sources set up, either because you used setup.exe or because you [set them up manually](#), you need to reference them from your project. To do that, right-click your project, and select **Manage NuGet Packages**:



Once in the package manager, follow the steps:



1. Select **Browse**
2. Select **All** as the package source.
3. Search for **FlexCel**
4. Select the TMS.FlexCel package.
5. Press Install.

NOTE

In step 2, you could have selected "**flexcel.tmssoftware.com**" as source instead of "**All**", to be able to find FlexCel faster. But if you selected flexcel.tmssoftware.com it wouldn't be able to resolve dependencies to other packages in nuget.org.

So to be safe, make sure to select All and search for FlexCel, instead of selecting the flexcel.tmssoftware.com feed

NOTE

Normally the only package you need to install is **TMS.FlexCel**. This package includes almost all the functionality in FlexCel. But when doing WinForms apps, there is an Excel previewer available in the **TMS.FlexCel.WinForms** package that you might also want to use. For WebForm apps, there is also a viewer which is available in the **TMS.FlexCel.WebForms** package.

The extra packages only work in Windows and in WinForms or WebForms apps. For all other kinds of apps, the only package that you need and can install is TMS.FlexCel.

Manually Referencing FlexCel

If you prefer not to work with nuget packages, FlexCel setup also installs the dlls directly. Instead of adding the NuGet package, you can just right click in the "References" entry in your project, select "Add Reference" and manually add the FlexCel.dll that is needed for your platform and framework type.

NOTE

In FlexCel for **.NET Core** you can't manually reference the assemblies and the only choice is via NuGet.

Getting Started with FlexCel Studio for the .NET Framework

0. Before starting: Choosing how to install and reference FlexCel

When installing FlexCel, there are 2 options:

1. Download the exe setup. This is the preferred way to **install** FlexCel in Windows, since it will install the NuGet package, the libraries, the examples and docs.
2. Download the NuGet packages. This includes only the NuGet packages (which are also included in the exe setup), but doesn't include example code or docs, and won't register the NuGet source in your machine. This is the preferred way to install FlexCel in platforms different from Windows, and you can find more information on how to install it on the [installation guide](#).

Once you have FlexCel installed, you need to decide how to **reference** it. There are 2 ways you can reference FlexCel:

1. Install via NuGet packages. This is a standard installation same as any other NuGet installation, but with the difference that FlexCel is not stored in nuget.org How to install via NuGet is detailed step by step in the [installation guide](#)
2. Install by manually referencing the assemblies. This is possible in all platforms except in .NET Core, where the only option is via NuGet.

How you decide to reference the assemblies is up to you: .NET is moving from a monolithic framework to a framework "on demand" via NuGet, and so we would recommend you to use NuGet too. But if you prefer to reference the assemblies directly, you can do that too.

TIP

If you are unsure, just install the exe setup and use the FlexCel NuGet packages.

1. Creating an Excel file with code

The simplest way to use FlexCel is to use the [XlsFile](#) class to manipulate files.

To get started, create an empty Console application and save it. [Add the TMS.FlexCel NuGet package to your application](#) or [add a manual reference to FlexCel.dll](#)

Then replace all the text in the file by the following:

```

using System;
using FlexCel.Core;
using FlexCel.XlsAdapter;

namespace Samples
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            //Create a new empty Excel file, with default formatting as if it was
            //created by Excel 2019.
            //Different Excel versions can have different formatting when they
            //create
            //an empty file, so for example
            //Excel 2003 will have a default font of Arial, and 2019 will use
            //Calibri.
            //This format is anyway the starting format, you can change it all
            //later.
            XlsFile xls = new XlsFile(1, TExcelFileFormat.v2019, true);

            //Enters a string into A1.
            xls.SetCellValue(1, 1, "Hello from FlexCel!");

            //Enters a number into A2.
            //Note that xls.SetCellValue(2, 1, "7") would enter a string.
            xls.SetCellValue(2, 1, 7);

            //Enter another floating point number.
            //All numbers in Excel are floating point,
            //so even if you enter an integer, it will be stored as double.
            xls.SetCellValue(3, 1, 11.3);

            //Enters a formula into A4.
            xls.SetCellValue(4, 1, new TFormula("=Sum(A2:A3)"));

            //Saves the file to the "Documents" folder.

            xls.Save(System.IO.Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Personal), "test.xls"));
        }
    }
}

```

And that is it. You have just made an application that creates an Excel file. Of course we are just scratching the surface: we will see more advanced uses later in this guide.

NOTE

This sample deduces the file format from the file name. If you saved as "test.xls", the file format would be **xls**, not **xlsx**. You can specify the file format in a parameter to the "Save" method if needed; for example when saving to streams.

2. Creating a more complex file with code

While creating a simple file is simple (as it should), the functionality in Excel is quite big, and it can be hard to find out the exact method to do something. FlexCel comes with a tool that makes this simpler:

2.1 Open APIMate

When you install FlexCel, it will install a tool named **APIMate**. You can access it by going to the Start Menu and searching for APIMate.

You can also download APIMate for your operating system from the following locations:

- [APIMate for Windows](#)
- [APIMate for macOS](#)

Or you can compile it from source (sources are included when you install FlexCel).

2.2. Create a file in Excel with the functionality you want.

To get the best results, keep the file simple. Say you want to find out how to add an autofilter, create an empty file in FlexCel and add an autofilter. If you want to find out how to format a cell with a gradient, create a different file and format one cell with a gradient.

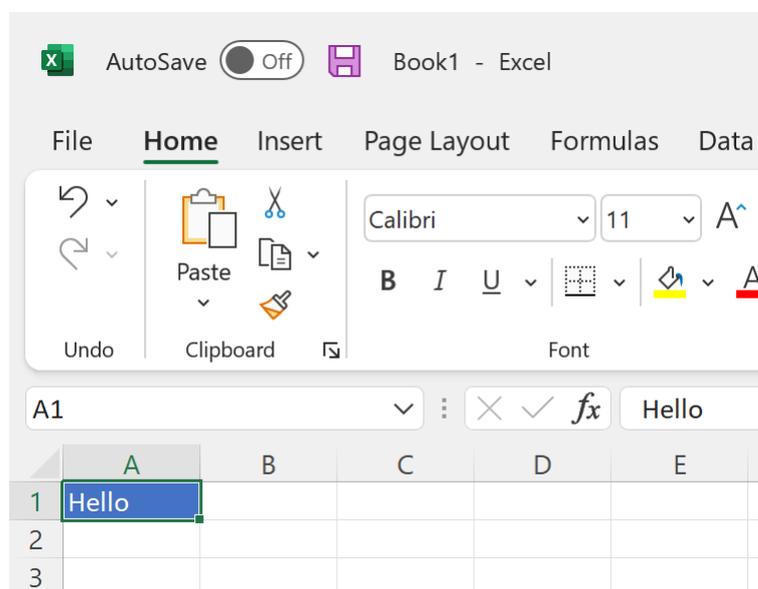
Using simple files will make it much easier to find the relevant code in APIMate

2.3. Open the file with APIMate

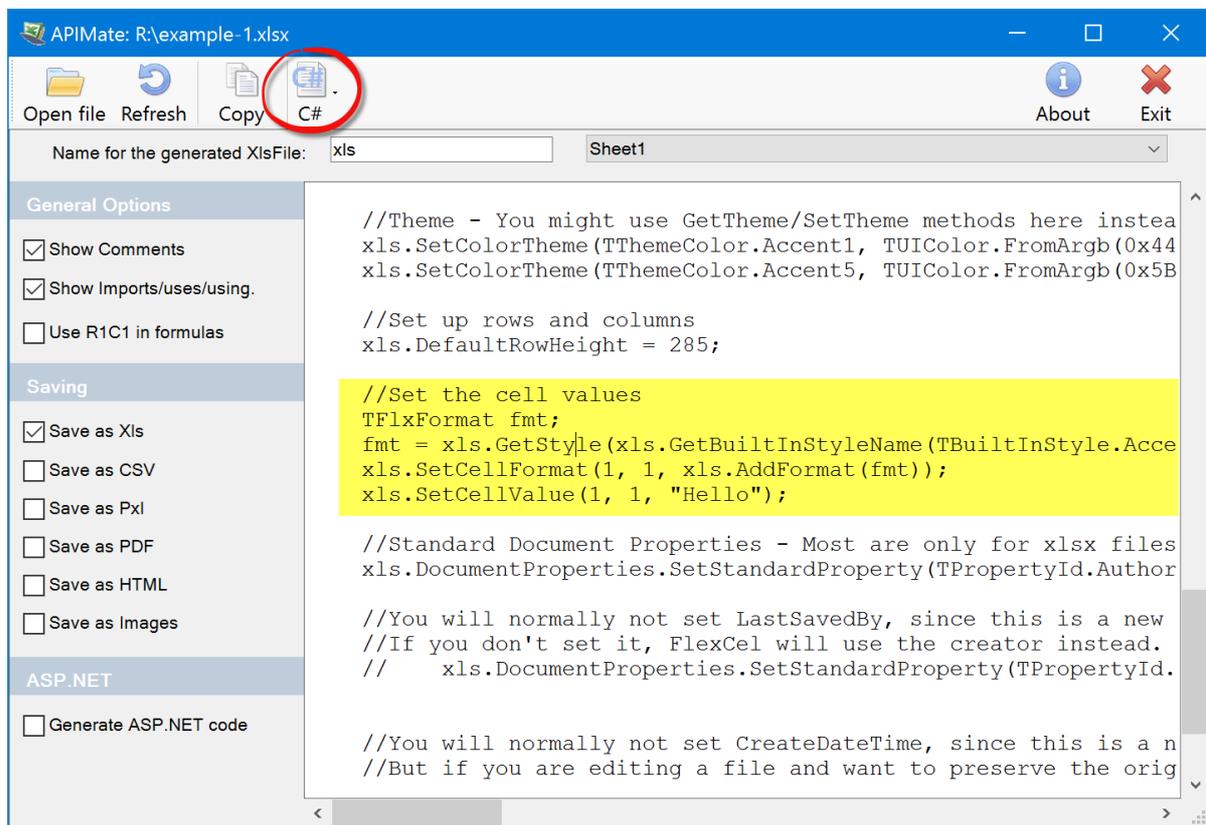
APIMate will tell you the code you need to recreate the file you created in Excel with FlexCel. You can see the code as C# or VB.NET.

You can keep the xls/x file open in both Excel and APIMate, modify the file in Excel, save, press "Refresh" in APIMate to see the changes.

Imagine you have this file, with a cell formatted in blue:



When you open it in apimate, you should see this code which is the code you need to write in FlexCel to generate the same file:



Note that there is a language button in the toolbar where you can choose which language you want the code to be.

3. Reading a file

There is a complete example on Reading files in the documentation. But for simple reading, you can do as follows:

Create a new Console application, and write the following code:

```
using System;
using FlexCel.Core;
using FlexCel.XlsAdapter;

namespace FileReader
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            XlsFile xls = new XlsFile(System.IO.Path.Combine(Environment.GetFolder
rPath(Environment.SpecialFolder.Personal), "test.xlsx"));

            xls.ActiveSheetByName = "Sheet1"; //we'll read sheet1. We could loop
over the existing sheets by using xls.SheetCount and xls.ActiveSheet
```

```

        for (int row = 1; row <= xls.RowCount; row++)
        {
            for (int colIndex = 1; colIndex <= xls.ColCountInRow(row); colIndex++) //Don't use xls.ColCount as it is slow: https://doc.tmssoftware.com/flexcel/net/guides/performance-guide.html#avoid-calling-colcount
            {
                int XF = -1;
                object cell = xls.GetCellValueIndexed(row, colIndex, ref XF);

                TCellAddress addr = new TCellAddress(row,
xls.ColFromIndex(row, colIndex));
                Console.WriteLine("Cell " + addr.CellRef + " ");
                if (cell == null) Console.WriteLine("is empty.");
                else if (cell is TRichString) Console.WriteLine("has a rich
string.");
                else if (cell is string) Console.WriteLine("has a string.");
                else if (cell is Double) Console.WriteLine("has a number.");
                else if (cell is bool) Console.WriteLine("has a bool.");
                else if (cell is TFlxFormulaErrorValue)
Console.WriteLine("has an error.");
                else if (cell is TFormula) Console.WriteLine("has a formula.");
                ;
                else Console.WriteLine("Error: Unknown cell type");
            }
        }
    }
}
}
}
}

```

4. Manipulating files

APIMate will tell you about a huge number of things, like how to paint a cell in red, or how to insert an autofilter. But there are some methods that APIMate can't tell you about, and from those the most important are the manipulating methods:

- Use [ExcelFile.InsertAndCopyRange](#) for inserting rows or column or ranges of cells. Also for copying ranges or cells or full rows or full columns. Or for inserting and copying cells/columns/rows in one operation (like pressing "Copy/Insert copied cells" in Excel). It can also copy the cells from one sheet to the same sheet, to another sheet, or to another sheet in another file. InsertAndCopyRange is a heavily overloaded method, and it can do many things depending on the parameters you pass to it.
- Use [ExcelFile.DeleteRange](#) to delete ranges of cells, full rows or full columns.
- Use [ExcelFile.MoveRange](#) to move a range, full rows or full columns from one place to another.
- Use [ExcelFile.InsertAndCopySheets](#) to insert a sheet, to copy a sheet, or to insert and copy a sheet in the same operation.
- Use [ExcelFile.DeleteSheet](#) to delete a sheet.

5. Creating Reports

You can create Excel files with code as shown above, but FlexCel also includes a reporting engine which uses Excel as the report designer. When using reports you create a template in Excel, write some tags on it, and run the report. FlexCel will replace those tags by the values from a database or memory.

1. Create an empty file in Excel
2. In cell A1 of the first sheet, write `<#Customer.Name>`. In cell B1 write `<#Customer.Address>`
3. In the ribbon, go to "Formulas" tab, and press "Name manager" (In Excel for macOS or Excel 2003, go to Menu->Insert->Name->Define)
4. Create a name "**Customer**" that refers to `"=Sheet1!A1"`. The name is case insensitive, you can write it in any mix of upper and lower case letters. It needs to start with two underscores ("_") and end with two underscores too. We could use a single underscore for bands that don't take the full row or "I_" or "I_" for column reports instead, but this is for more advanced uses.
5. Save the file as "report.template.xlsx" in the "Documents" folder
6. Create a new Console app, save it as "CustomerReport", and paste the following code:

```
using System;
using System.Collections.Generic;
using FlexCel.Core;
using FlexCel.Report;

namespace Samples
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            var Customers = new List<Customer>();
            Customers.Add(new Customer { Name = "Bill", Address = "555 demo
line" });
            Customers.Add(new Customer { Name = "Joe", Address = "556 demo line" }
);

            using (FlexCelReport fr = new FlexCelReport(true))
            {
                fr.AddTable("Customer", Customers);
                fr.Run(
                    System.IO.Path.Combine(Environment.GetFolderPath(Environment.S
pecialFolder.Personal), "report.template.xlsx"),
                    System.IO.Path.Combine(Environment.GetFolderPath(Environment.S
pecialFolder.Personal), "result.xlsx")
                );
            }
        }
    }
}
```

```

    }
}

class Customer
{
    public string Name { get; set; }
    public string Address { get; set; }
}
}

```

NOTE

If creating a macOS Console application, you will need to add a reference to System.Data, System.Xml and XamMac or MonoMac to the app. You might also need to copy XamMac.dll or MonoMac.dll to your output folder. And in a console application, you will need to initialize the Cocoa framework by calling MonoMac.AppKit.NSApplication.Init() For normal macOS applications you will not need to do anything: This applies only to Console macOS apps.

6. Exporting a file to pdf

FlexCel offers a lot of options to export to pdf, like PDF/A, exporting font subsets, signing the generated pdf documents, etc. This is all shown in the examples and documentation. But for a simple conversion between xls/x and pdf you can use the following code:

```

public static void ExportToPdf(string inFile, string outFile)
{
    XlsFile xls = new XlsFile(inFile);

    using (var pdf = new FlexCel.Render.FlexCelPdfExport(xls, true))
    {
        pdf.Export(outFile);
    }
}

```

7. Exporting a file to html

As usual, there are too many options when exporting to html to show here: Exporting as HTML 3.2, HTML 4 or HTML 5, embedding images or css, exporting each sheet as a tab and a big long list of etc. And as usual, you can find all the options in the documentation and examples.

For this getting started guide, we will show how to do an export with the default options of the active sheet.

```

public static void ExportToHtml(string inFile, string outFile)
{
    XlsFile xls = new XlsFile(inFile);

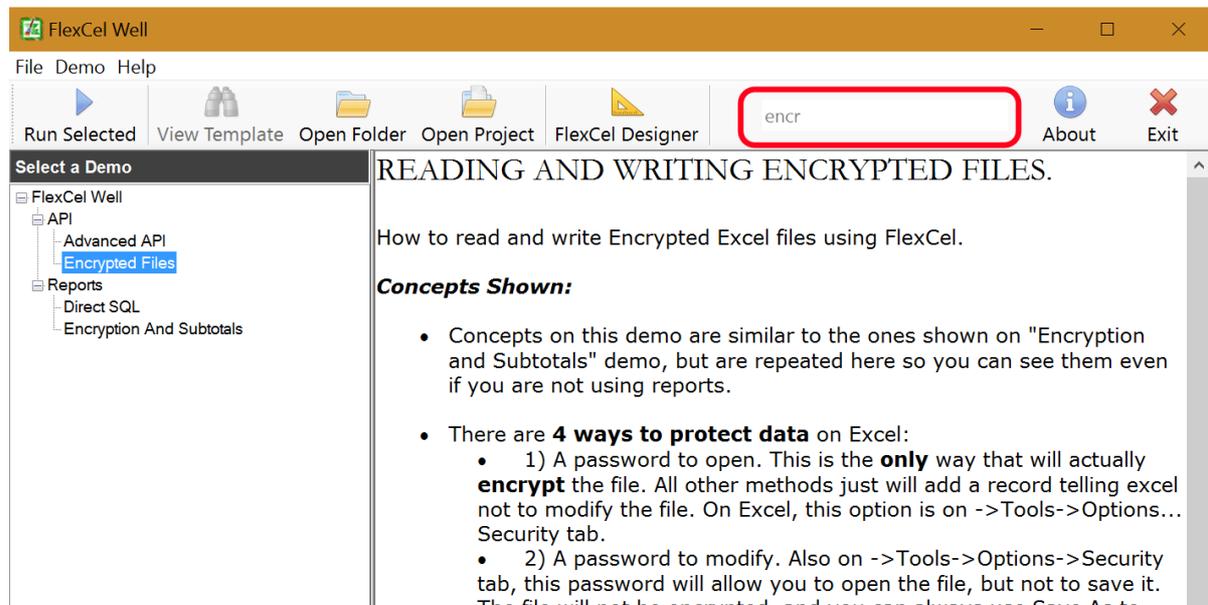
    using (var html = new FlexCel.Render.FlexCelHtmlExport(xls, true))
    {
        html.Export(outFile, null);
    }
}
}

```

8. Browsing through the Examples

FlexCel comes with more than 50 examples of how to do specific things. You can open each demo as a standalone project, but you can also use the included "Demo Browser" (this is MainDemo.csproj) to look at them all in a single place.

You can search for specific keywords at the top right of the main screen, to locate the demos that deal with specific features. So for example if you are looking for demos which show encryption, you will write "encrypt" in the search box:



9. This ends this small guide, but there is much more.

Make sure to take a look at all the other documents available here. You can use the tabs at the top of this site to read the different sections.

FlexCel API Developer Guide

Introduction

The FlexCel API (Application Programmer Interface) is what you use to read or write Excel files in a low-level way. By “low-level” we mean that this API is designed to work really “close to the metal” and there aren’t many layers between you and the xls/xlsx file being created. For example, FlexCel API doesn’t know about datasets, because datasets are a higher level concept. If you want to dump a dataset into an Excel file using the API, you need to loop in all records and enter the contents into the cells.

In addition to the FlexCel API we provide a higher level abstraction, [FlexCelReport](#), that knows about datasets and in general works at a more functional level; a declarative approach instead of an imperative approach. What is best for you depends on your needs.

Basic Concepts

Before starting writing code, there are some basic concepts you should be familiar with. Mastering them will make things much easier down the road.

Arrays and Cells

To maintain our syntax compatible with Excel OLE automation, most FlexCel indexes/arrays are 1-based.

That is, cell A1 is (1,1) and not (0,0). To set the first sheet as ActiveSheet, you would write

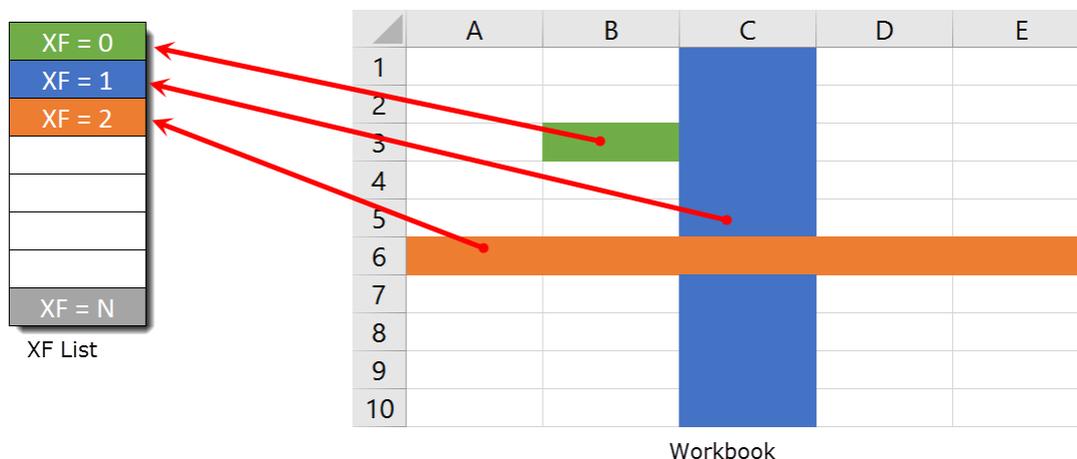
```
ActiveSheet = 1 and not ActiveSheet = 0.
```

So, in C# and C++ loops should read: `for (int i = 1; i <= Count; i++)`, and in VB.NET they should read `for i = 1 to Count`.

The two exceptions to this rule are XF and Font indexes, which are 0 based because they are so in Excel.

Cell Formats

All formats (colors, fonts, borders, etc.) on an Excel workbook are stored into a list, and referred by number. This number is known as the XF (**eX**tended **F**ormat) index. A simple example follows:



Here Cell B3 has XF = 0 and the XF definition for the background color is green. Row 6 has XF = 2, so all the empty cells on row 6 are orange. Column C has XF = 1, so all the empty cells on column C that do not have a Row Format are blue.

The priority goes, from higher to lower: Cell format->Row format->Column format. This means that if column A has a column format red and Row 1 has row format green, the cell A1 will be green. If you apply a cell format to cell A1, it will be used instead of either row or column formats as it has the highest priority.

NOTE

Sometimes when you work in Excel, it might look like Excel is handling priority differently. For example if you format a row green and after that a column red, you will see that the intersection between the column and row is red.

What is happening under the hood is that when you format the column red, Excel applies a **column format** red to the column, and after that a **cell format** red to the cell in the intersection. This way it can look like the column format is taking priority over the row, but it is actually the cell format applied to the cell at the intersection which has the priority.

Most formatting methods at FlexCel return an **XF** index, and then you have to look at the XF list (using the [ExcelFile.GetFormat](#) method) to get a class encapsulating the real format. There is a helper method: [ExcelFile.GetCellVisibleFormatDef](#) that obtain the **XF** index and return the format class in one step.

To create new formats, you have to use the [ExcelFile.AddFormat](#) method. Once you get the Id of the new XF, you can use it as you wish.

NOTE

You don't have to worry also on inserting a format 2 times: If it already exists AddFormat will return the existing id and not add a new XF entry.

Cell and Style Formats

XF formats can be of two types, "Cell" or "Style". Cell formats are applied directly to a cell and can have a "ParentStyle" which must be a style format. Style formats cannot have ParentStyle and cannot be applied to cells, but they can be used as the base for defining different Cell formats. You can know if a [TFlxFormat](#) contains a "Style" or "Cell" format by looking at its "IsStyle" property. Also, a cell style can link only parts of its format to its parent style, for example have the font linked so when you change the font in the style it changes the font in the cell, but not the cell borders. In order to set which properties are linked to the main style, you need to change the "LinkedStyle" property in [TFlxFormat](#).

You can create new Style formats with [ExcelFile.SetStyle](#), and new Cell formats with [ExcelFile.AddFormat](#). Once you create a new style and give it a name, you can get its definition as a [TFlxFormat](#), change its [TFlxFormat.IsStyle](#) property to true, define which properties you want to link with the [TFlxFormat.LinkedStyle](#) property, and add that format using [AddFormat](#) to get the cell format. Once you have the cell style, you can just apply it to a cell.

NOTE

The [TLinkedStyle](#) class has a member, [TLinkedStyle.AutomaticChoose](#), which if left to true (the default) will compare your new format with the style and only link those properties that are the same.

For example, let's imagine you create a style "MyHyperlink", with font blue and underlined. Then you create a Cell format that uses "MyHyperLink" as parent style, but also has a red background, and apply it to cell A1. If you leave [AutomaticChoose](#) true, FlexCel will detect that the cell format is equal in everything to its parent style except in the background, so it will not link the background of the cell to the style. If you later change the background of MyHyperlink, cell A1 will continue to be red.

This allows for having "partial styles" as in the example above: a "hyperlink" style defines that text should be blue and underlined, but it doesn't care about the cell background. If you want to manually set which properties you want to have linked instead of FlexCel calculating that for you, you need to set [TLinkedStyle.AutomaticChoose](#) to false and set the "Linked..." Properties in [TLinkedStyle](#) to the values you want.

Font Indexes

The same way we have an XF list where we store the formats for global use, there is a Font list where fonts are stored to be used by XFs. You normally don't need to worry about the FONT list because inserting on this list is automatically handled for you when you define an XF format. But, if you want to, you can for example change Font number 7 to be 15 points, and all XFs referencing Font 7 will automatically change to 15 points.

Colors

While colors in Excel up to 2003 were indexes to a palette of 56 colors, Excel 2007 introduces true colors, and FlexCel implements it too since version 5.0. One thing that is nice about the new colors is that they can be retrofitted in the xls file format, so while you won't be able to see true color in Excel 2003 (Excel 2003 just doesn't know about them), the data is there in the xls file anyway. So if you open the xls file with FlexCel 5 or later Excel 2007 and later you will see the true colors, even if the xls file format originally didn't know about them.

There are four kinds of colors that you can use in Excel:

- **RGB:** This is just a standard color specified by its Red, Green and Blue values.
- **Theme:** There are 12 theme colors, and each of them can have a different tint (brightness) applied. Theme colors are the default in newer Excel versions and you are likely using them even if not aware.

Given the importance of theme colors in newer Excel, and that they aren't trivial to describe, we will cover them in a separate section below.

- **Indexed Colors.** This is kept for background compatibility with Excel 2003, but we strongly recommend you forget about them.
- **Automatic Color.** An automatic color changes depending on whether the color is used as a background or as a foreground. For foreground colors (like a font color) the Automatic color is **black**. For background colors (like a cell back color) the automatic color is **white**

You can access any of these variants with the struct [TExcelColor](#).

[TExcelColor](#) also automatically converts from a `System.Drawing.Color`, so you can just specify something like:

```
MyObject.Color = Color.Blue;
```

If you want to use a theme color, you will have to use:

```
MyObject.Color = TExcelColor.FromTheme(...);
```

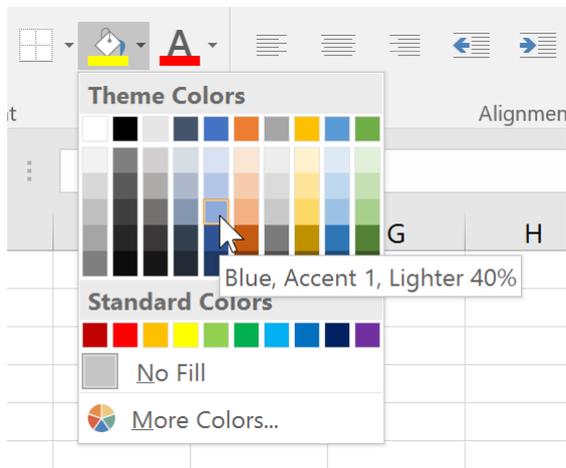
The "NearestColorIndex" used in older version of FlexCel has been deprecated and you shouldn't use it.

Theme colors

Theme colors are an interesting topic well worth its own section in this manual. Excel uses theme colors by default, which means that most spreadsheets you see around are using them, even if the people who created the files didn't know what a "theme" color was.

The first thing to notice is that Excel has 12 theme colors, which FlexCel maps inside the [TThemeColor](#) enumeration. Each one of those 12 colors can be made lighter or darker by changing its [TExcelColor.Tint](#) leading to millions of possible colors derived from those basic 12 colors.

When you open a color dialog in Excel, the colors that are shown in the main square are different variations of the 12 theme colors:



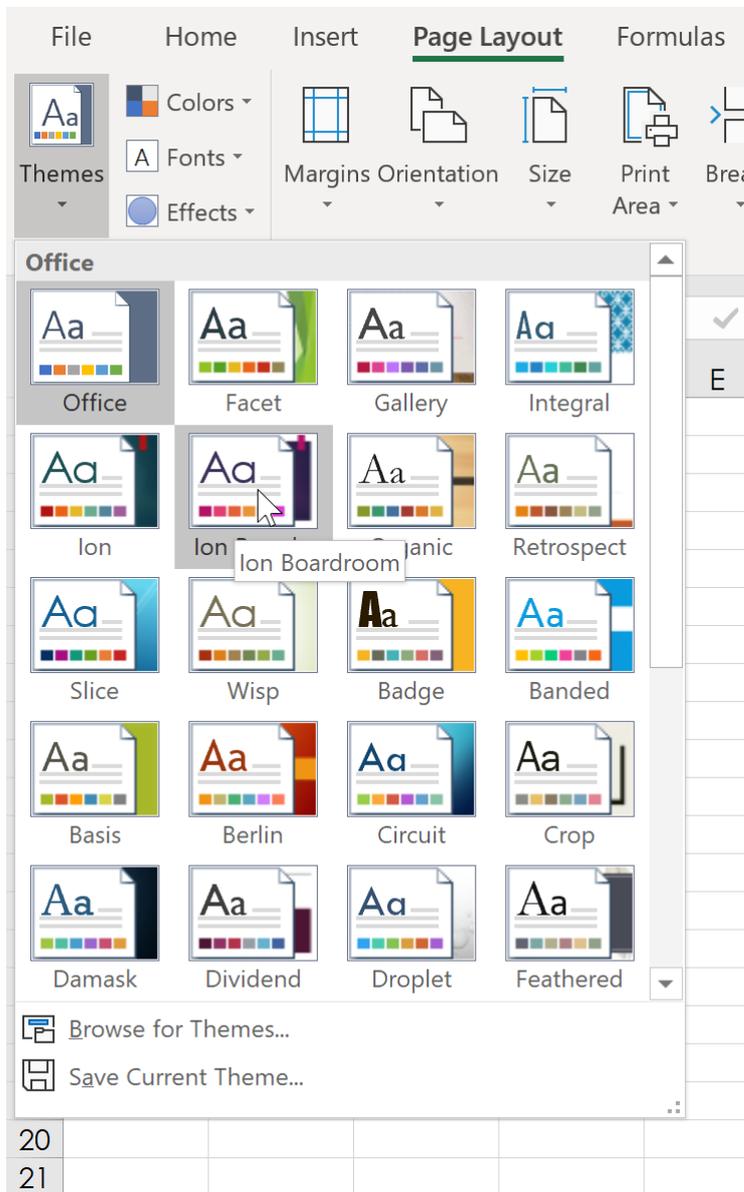
Each one of the 12 theme colors takes one column in the grid, and in the rows of the grid you can see variations of those colors with different tints.

NOTE

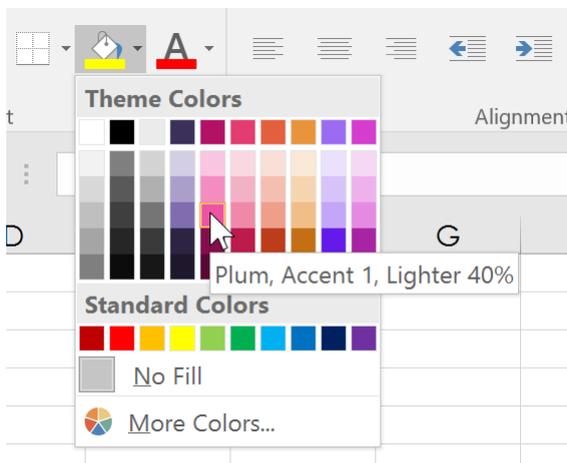
We kind of lied here: If you tried to count the columns in the image, you would see that there are 10 columns instead of 12. This is because even when Excel has 12 defined theme colors, only 10 of them are visible. The other 2 are the color for a **hyperlink** and a **visited hyperlink**, and they don't show in the Excel UI as usable colors.

The selected color you see in the image above is a color of the theme **Accent1** (all colors in column 4 are of theme **Accent1**). It also has applied a tint which makes it 40% lighter than the standard **Accent1** color, as shown in the Excel tooltip.

Themes are a nice way to specify colors, since you can change the theme later, and all colors will change to match. This means that you can go to the theme selector in the Excel ribbon (in the Page Layout tab) and change what **Accent1** (and all the other theme colors) mean:



If you select for example the "Ion Boardroom" theme as shown in the image, you will see the color palette changes to:



Note how the color **Accent1** is now **Plum** instead of **Blue**. When you change the theme of your spreadsheet, all cells which had **Accent1** theme colors will change from blue to plum, while cells which had a **Blue RGB color** set will continue to be blue.

When setting colors in a file you are creating, it is a nice thing to use theme colors instead of RGB colors, so your users will be able to change the theme. FlexCel has full support of theme colors, and you can easily find out how to use them in your code with APIMate.

You can find more information about Excel 2007 colors at

<http://tmssoftware.com/site/blog.asp?post=135>

Date Cells

As you might already know, there is no DATE datatype in Excel.

Dates are saved as a double floating number where the integer part is the number of days that have passed from 1/1/1900, and the fractional part is the corresponding fraction of the day. For example, the number 2.75 stands for "02/01/1900 06:00:00 p.m." You can see this easily at Excel by entering a number on a cell and then changing the cell format to a date, or changing the cell format of a date back to a number.

The good news is that you can set a cell value to a DateTimeValue, and FlexCel will automatically convert the value to a double, keeping in count "1904" date settings (see below). That is, if you enter

`XlsFile.SetCellValue(1, 1, DateTime.Now)`, and the cell (1,1) has date format, you will write the actual value of "now" to the sheet.

The bad news is that you have no way to know if a cell has a number or a date just by looking at its value. If you enter a date value into a cell and then read it back, you will get a double. So you have to look at the format of the cell. There is a helper function, FormatValue that can return if the cell has a date or not by looking at the format.

This would be the correct way to find out if a cell contains a date:

```
var fmt = xls.GetCellVisibleFormatDef(row, col);
if (TFlxNumberFormat.HasDateOrTime(fmt.Format))
{
    LogMessage("Cell " + new TCellAddress(row, col).CellRef + " contains a
Date and Time");
}
if (TFlxNumberFormat.HasDate(fmt.Format))
{
    LogMessage("Cell " + new TCellAddress(row, col).CellRef + " contains a
Date");
}
if (TFlxNumberFormat.HasTime(fmt.Format))
{
    LogMessage("Cell " + new TCellAddress(row, col).CellRef + " contains a
Time");
}
```

Note how the actual value of the cell doesn't matter, only the format of the cell.

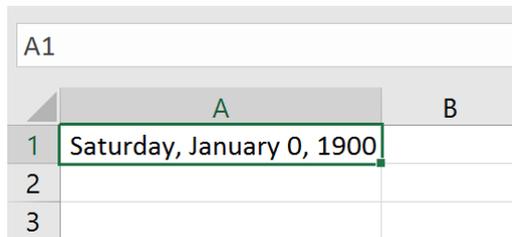
NOTE

Excel also has a "1904" date mode, where dates begin at 1904 and not 1900. This mode was used on **Excel for mac** before OSX: today both OSX and Windows use a 1900 date system. But you can change this option in Excel for OSX or Windows today too, so you might come across some file using the 1904 date system. FlexCel completely supports 1900 and 1904 dates, but you need to be careful when converting dates to numbers and back.

You can read more information here: <https://support.microsoft.com/en-us/help/214330/differences-between-the-1900-and-the-1904-date-system-in-excel>

NOTE

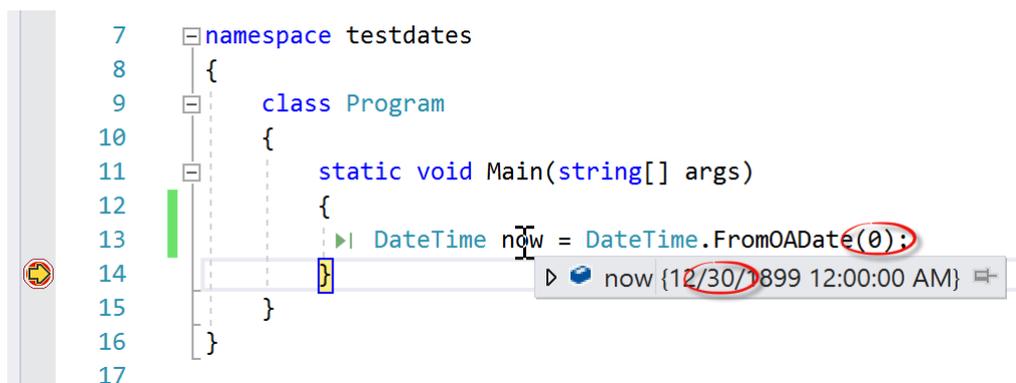
There is a bug in how Excel handles the 1900 year. It considers it to be a leap year but 1900 was a normal year and February 29 1900 never happened. This means that the serial number used to represent the date gets one day out of sync with the serial number used by C#. The serial number 0 will show in Excel as "January 0, 1900" (even if there was never a January 0 either)



	A	B
1	Saturday, January 0, 1900	
2		
3		

While in C# it will show as **December 30, 1899**

```
7 namespace testdates
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13            DateTime now = DateTime.FromOADate(0);
14        }
15    }
16 }
17
```



The screenshot shows a C# code editor with a debugger window. The code defines a namespace 'testdates' containing a class 'Program' with a static method 'Main'. Inside 'Main', the variable 'now' is assigned the value of 'DateTime.FromOADate(0)'. The debugger shows the value of 'now' as '12/30/1899 12:00:00 AM', with the year and month circled in red.

Note that even if Excel cared to display correctly the 0 as December **31**, 1899 instead of January 0, 1900 it would still be off by a day with the dates as used in C#, which displays December **30**.

FlexCel will try to do the best to replicate the Excel bug so you get the same values as Excel (ironically, Excel has the bug only because it was replicating a bug in Lotus 1-2-3). This means that for example `FlxDateTime.FromOADate(0, false)` will return a `DateTime` of January 31, 1899 which is the closest we can get to the value Excel shows. (we can't return a `DateTime` with a value of January 0, that is not possible). But whatever we do, the reality is that dates in Excel before March 1, 1900 are wrong. Avoid using them.

You can read more information on this bug here. <https://support.microsoft.com/en-us/help/214326/excel-incorrectly-assumes-that-the-year-1900-is-a-leap-year>

Copying and pasting native Excel data

[ExcelFile](#) has a group of methods that allow you to copy/paste from/to FlexCel to/from Excel in native Excel format (Biff8). You can copy data to the clipboard in Biff8, Tabbed Text, and HTML formats. You can paste from the clipboard in BIFF8 and Tabbed-Text format.

Normally copying in Biff8 is good for pasting the data in Excel, Html is good if you later want to paste in Word or PowerPoint, and text works for pasting into applications that don't understand Biff8 or HTML.

Copying and pasting in native BIFF8 format is a great advance over copying/pasting on plain text only. It allows you to keep cell formats/colors/rounding decimals/merged cells/etc. But it has its limitations too:

- It can't copy/paste images
- It can't copy/paste strings longer than 255 characters
- It can't copy the data on multiple sheets.

We would like to say that these limitations are not FlexCel's fault. The BIFF8 specification is correctly implemented; those are limitations on Excel's part.

Of course, Excel can copy and paste everything without problems, but this is because Excel doesn't use the clipboard to do the operation. If you close all instances of Excel, open a Worksheet, copy some cells to the clipboard, close Excel and open it again you will run into the same limitations. Copy/paste limitations on Excel don't show when it is kept in memory.

Reading and Writing Files

The native FlexCel engine is encapsulated on the class [XlsFile](#).

This class stores an Excel file in memory, and has methods allowing loading a new file into it, modifying it, or saving its contents to a file or a stream.

NOTE

[ExcelFile](#) is the abstract class that provides the interface that [XlsFile](#) implements. In this documentation, we use [ExcelFile](#) and [XlsFile](#) interchangeably, because both implement the same methods and properties. The first is an abstract class which specifies the functionality, and the second is the actual code.

IMPORTANT

Even when a FlexCel object is fully managed and you don't need to dispose it, keep in mind that it stores a full spreadsheet in memory. **Do not leave global XlsFile objects hanging around when you don't need them** because they can use a lot of memory. Use local XlsFile instances or set them to null as soon as you are done using them.

Opening and saving files

A simple code to process an Excel file would be:

```

private static void ProcessFile(string sourceFileName, string destFileName)
{
    XlsFile xls = new XlsFile(true);
    xls.Open(sourceFileName);

    //... process the file
    xls.Save(destFileName);
}

```

Here we can note:

1. By default, FlexCel never overwrites an existing file. So, before saving you always have to call `File.Delete`, or set `ExcelFile.AllowOverWritingFiles` property = true. On this example, we set **AllowOverWritingFiles = true** when we construct the object in the line: `XlsFile xls = new XlsFile(true);`.
2. While you can explicitly specify the file format when saving, normally you don't need to do it. FlexCel is smart enough to know that files with extension "xlsx" or "xlsm" must be saved in xlsx file format, and that files with extension "xls" must be saved as xls. **When saving to a stream, then you need to specify the file format**, or it will be xls by default.

Modifying files

Once you have loaded a document with `ExcelFile.Open` or created a new empty file with `ExcelFile.NewFile` you can proceed to modify it. Add cells, formats, images, insert sheets, delete ranges, merge cells or copy ranges from one place to another. It is not on the scope of this document to show each and every thing you can do because there are hundreds of commands and that would take another book. To learn about specific methods, take a look at the [examples](#) and use the **APIMate tool** as described [here in the Getting Started guide](#).

Inserting, Copying and Moving Cells / Rows / Columns and Sheets

While APIMate will cover much of the API, there is a whole category of commands that can't be shown there, and that are very important while manipulating files. Those commands are the ones used to insert/copy/move cells/sheets, etc. One of FlexCel strongest points is the big support offered for all those operations. Since those methods are used everywhere in FlexCel Reporting engine, they are at the core of the engine, not inserted as an afterthought. They perform very well, and they are deeply tested.

Now, something that might be nonintuitive if you are not aware of it, is that FlexCel provides that functionality in a few very powerful and overloaded methods, instead of a million standalone methods. That is, instead of having "InsertRow" "InsertColumns", "InsertCells" "CopyColumns", "CopyCells" "CopyRows", "InsertAndCopyRows" "InsertAndCopyColumns", etc., FlexCel just provides a single "[InsertAndCopyRange](#)" method that does all of this and more (like copying rows from one file and inserting it into another)

The reason we kept it this way (few powerful methods instead of many specialized and simpler methods) is that fewer methods are easier to remember. From the editor, you just need to remember to start typing "InsertAnd..." and intellisense will show most options. No need to guess if the method to add a new sheet was named "AddSheet" or "InsertSheet".

Naming probably isn't the best either, this is for historical reasons (because original implementations did only insert and copy, and then, as we added more functionality it was added to those methods), but also because a better naming isn't easy either. Calling a method `InsertAndOrCopyCellsOrColumnsOrRanges()` isn't much better. So just remember that whatever you want to do, you can probably do it with one of the following methods:

- **InsertAndCopyRange:** Probably the most important of the manipulating methods. Inserts or copies or inserts and copies a range of cells or rows or columns from one sheet to the same sheet, or to a different sheet, or to a different file. It works exactly as Excel does, if you copy a formula "=A1" down to row 2, it will change to be "=A2". Images and objects might or might not be copied too depending in the parameters to this method.
- **MoveRange:** Moves a range of cells, or a column or a row inside a sheet.
- **DeleteRange:** Deletes a range of cells, a row or a column in a sheet, moving the other cells left or up as needed. If `InsertMode` is `NoneDown` or `NoneRight`, cells will be cleared, but other cells won't move.
- **InsertAndCopySheets:** Inserts and or copies a sheet. Use this method to add a new empty sheet.
- **DeleteSheet:** Deletes a number of sheets from the file.
- **ClearSheet:** Clears all content in a sheet, but leaves it in place.

NOTE

`InsertAndCopy` operations could be theoretically implemented as "Insert" then "Copy", but that would be much slower. In many cases we want to insert and copy at the same time, and doing it all at once is much faster. Also in some border cases where the inserted cells are inside the range of the copied cells, inserting and then copying can give the wrong results.

Considerations about Excel 2007 support

FlexCel 5.0 introduced support for Excel 2007 file format (xlsx), and also for the new features in it, like 1 million rows or true color.

While most changes are under the hood, and we made our best effort to keep the API unchanged, some little things did change and could bring issues when updating:

- **The number of rows by default in FlexCel is now 1048576, and the number of columns is 16384.** This can have some subtle effects:

- **Named ranges that were valid before might become invalid.** For example, the name **LAB1** would be a valid cell reference in 2007 mode, (Just like **A1**). In Excel 2003 you can have up to column "IV", but in Excel 2007 you can have up to column "XFD", so almost any combination of 3 characters and a number is going to be a valid cell reference, and thus can't be a name. If you have issues with this and can't change the names, you will have to use the "Compatibility mode" by setting the static variable:

[ExcelFile.ExcelVersion](#)

- **"Full Ranges" might become partial.** Full ranges have the property that they don't shrink when you delete rows. That is, if for example you set the print area to be A1:C65536 in Excel 2003, and you delete or insert rows the range will stay the same. If you open this file with FlexCel 5 or Excel 2007, and delete the row 1, the range will shrink to A1:C65535, as it is not full anymore. To prevent this, FlexCel automatically converts ranges that go to the last row or column in an xls file to go to the last row or column in an xlsx file. If you don't want FlexCel to autoexpand the ranges, you can change this setting with the static variable:

[ExcelFile.KeepMaxRowsAndColumnsWhenUpdating](#)

Also, when using XlsFile, use references like "=Sum(a:b)" to sum in a column instead of "=Sum(a1:b65536)"

- **Conditional Formats might become invalid.** This is a limitation on how Conditional Formats are stored in an xls file. They are stored as a complement-2 of 2 bytes for the rows and one byte for the column. For example, if you have a formula in a conditional format that references the cell in the row above, this will be stored as "0xFFFF", which is the same as "-1" if you are using signed math.

And as you have exactly 2 bytes of rows and 1 byte of columns they are also the same. It is the same to add 65535 rows (wrapping) or to subtract one, you will always arrive to the cell above. Now, with more columns and rows this is not true anymore. And you can't know by reading a file if the value was supposed to mean "-1" or "+65535". So both Excel and FlexCel might interpret a CF formula wrong when the cells are separated by too many rows or columns.

Again the solution if you have this issue is to turn [ExcelFile.ExcelVersion](#) back to 2003.

- **Colors are now true color, not indexed.**

- In order to support true color, we needed to change all "**Object.ColorIndex**" properties to "**Object.Color**", so old FlexCel code won't compile right away. You need to do a search and replace for "ColorIndex" properties. See the section about colors in this document for more information about colors and themes.

- If you relied on changing the palette to change cell colors, this code might break. Before, you had only indexed colors, so if you changed the palette color 3 from red to blue, all red cells would turn blue. Now, **the color will only change if the cell has indexed color**. You might have a cell with the exact same red but with the color specified as RGB, and this won't change when you change the palette. It is recommended that you use themes now in situations like this.

- **Default File Format for Saving:** In FlexCel 4 finding the default file format for saving was simple; it saved everything xls. Now, it isn't so simple. By default, if you do a simple `xls.Save(filename)` without specifying a file format we will use the file extension to determine it. When it is not possible (for example if you are saving to a stream) we will use the format the file was originally in when you loaded it. If you prefer to change the default, you can change the static property

[ExcelFile.DefaultFileFormat](#)

- **Headers and footer changes:** Now headers and footers can be set differently for odd and even pages, or for the first page. While the properties [PageHeader](#) and [PageFooter](#) still exist in `XlsFile`, you are advised to use [GetPageHeaderAndFooter](#) and [SetPageHeaderAndFooter](#) instead for better control. Also the method to change the images in the headers and footers changed to accept these different possibilities. Look at [APIMate](#) to see how to use these new methods.

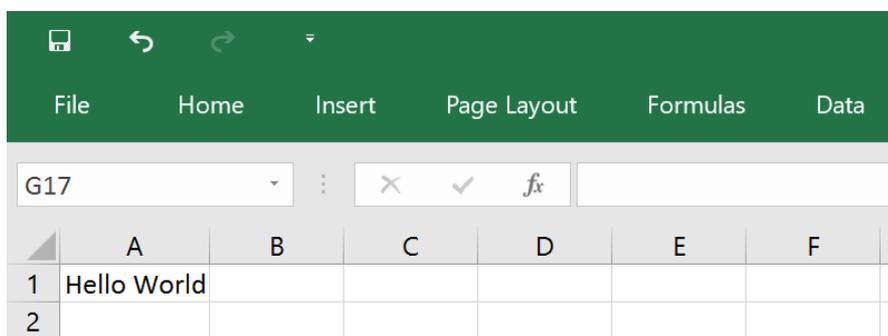
Autofitting Rows and Columns

FlexCel offers support for "autofitting" a row or a column, so it expands or shrinks depending on the data on the cells.

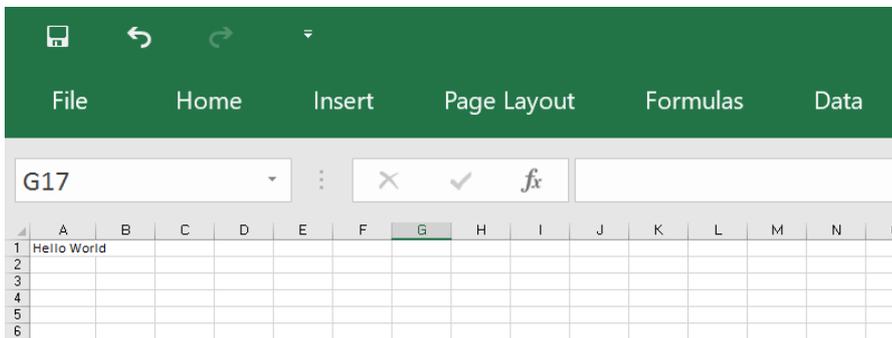
But **autofitting is not done automatically**, and we have good reasons for it to be this way.

To explain this better, let's try a simple example:

1. Imagine that we create a new Excel File, write "Hello world" on cell A1, and go to "Format->Column->AutoFit Selection". We will get something like this:



2. As you see, column "A" was resized so "Hello World" fits inside. Easy, isn't it? Well, not as much as we would like it to be. Let's now change the zoom to 50%:



3. Now the text "Hello world" is using part of column "B". We didn't change anything except the zoom and now text does not fit anymore, in fact, you can autofit it again and column "A" will get bigger.

What happened here? The easy answer is that Excel is resolution dependent. Fonts scale in "steps", and they look different at different resolutions. What is worse, printing also changes depending on the printer, and as a thumb rule, it is not similar at what you see on the screen.

So, what should a FlexCel autofit do? Make column A the width needed to show "Hello world" at 100% zoom, 96 dpi screen resolution? Resize column A so "Hello world" shows fine when printing? On a dot matrix printer? On a laser printer? Any answer we choose will lead us to a different column width, and there is no "correct" answer.

And it gets better. FlexCel uses GDI+, not GDI for rendering and calculating metrics, and GDI+ is resolution independent. But GDI and GDI+ font metrics are different: for example the space between letters on GDI+ for a font might be less than the space GDI gives to them. You will need less column width to fit "Hello world" on GDI+ than in GDI for that font, so the width calculated by FlexCel(GDI+) will be less than the width calculated by Excel(GDI).

As you can imagine, if we used all this space to describe the problem, is because there is not a real solution. Autofit on FlexCel will try to adapt row heights and column widths so the text prints fine from Excel on a 600 dpi laser printer, but text might not fit exactly on the screen. Autofit methods on FlexCel also provide an "Adjustment" parameter that you can use to make a bigger fit. For example, using 1.1 as adjustment, most text will display inside the cells in Excel at normal screen resolution, but when printing you might find whitespace at the border, since columns or rows are bigger than what they need to be.

Of course, autofit on FlexCel will work fine when printing from FlexCel, since FlexCel is resolution independent and it should always be the same. The problem arises when opening the files in Excel.

And this was the reason we do not automatically autofit rows (as Excel does). Because of the mentioned differences between GDI+ and GDI (and also between GDI at different resolutions), we cannot calculate exactly what Excel would calculate. If we calculated the row height must be "149" and Excel calculated "155", as all rows are by default autoheight, just opening an Excel file on FlexCel would change all row heights, and probably move page breaks. Due to the error accumulation, maybe in FlexCel you can enter one more row per page, and the header of the new page could land at the bottom of the previous.

The lesson, do the autofit yourself when you need to and on the rows that really need autofit(most don't). If you are using XlsFile, you have **XlsFile.Autofit...** methods that you can use for that. If you are using FlexCelReport, use the **<#Row Height(Autofit)>** and **<#Column Width(Autofit)>** tags to autofit the rows or columns you want.

By default, Excel autofits all rows. So, when opening the file in Excel, it will recalculate row heights and show them fine. But when printing from FlexCel, make sure you autofit the rows you need, since FlexCel will not automatically do that.

Autofitting Merged Cells

Merged cells can be difficult to autofit, for two basic reasons:

1. If you are autofitting a row, and a merged cell spans over more than one row, which row should be expanded to autofit the cell? Imagine we have the following:

	A	B	C
1	This is a large text wrapped inside a merged cell. It is not clear		
2			
3			
4			
5			
6			
7			

We could make the text fit by enlarging for example row 1, or row 5:

	A	B
1	This is a large text wrapped inside a merged cell. It is not clear	
2		
3		
4		
5		
6		

	A	B
1	This is a large text wrapped inside a merged cell. It is not clear	
2	which row	
3	should be	
4	used to do the	
5	AutoFit.	
6		

We could also enlarge all rows by the same amount instead of changing only one row.

FlexCel by default will use the last row of the merged cell to autofit, since in our experience this is what you will normally need. But you can change this behavior by changing the parameter **"autofitMerged"** when you call the "Autofit" methods, or, if you are using reports, you can use the **<#Autofit settings>** tag to do so.

2. The second issue is that probably because of issue 1, Excel will never autofit merged cells. Even in cases where there would be no problem doing so, for example you are autofitting a row and the merged cell has only one row (but 2 columns). In all of those cases, Excel will just make the row the standard height. So, if you apply FlexCel autofitting but leave the autofit on, when you open the file in Excel, Excel will try to re-autofit the merged cell, and

you will end up with a single-size row always. So, when autofitting rows with merged cells in FlexCel **make sure you set autofitting off for Excel**. You don't need to do this for columns, since they don't autofit automatically in Excel.

NOTE

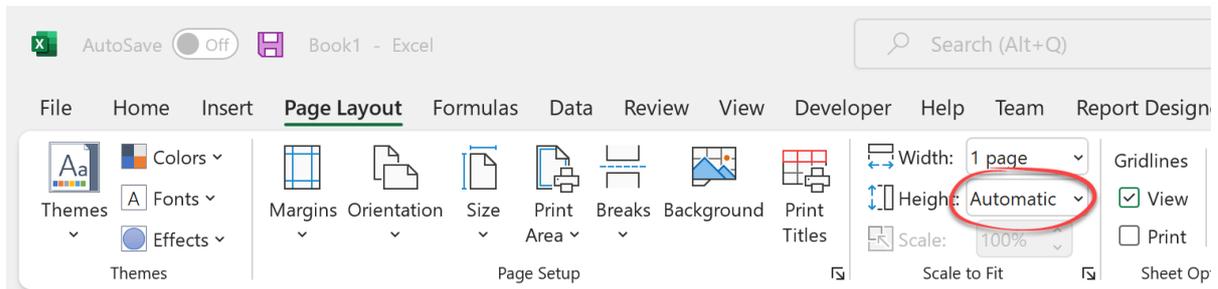
All the above was done in rows for simplicity, but it applies the same to columns and merged cells over more than one column.

Preparing for Printing

After creating a spreadsheet, one thing that can be problematic is to make it look good when printing or exporting to PDF.

Making the sheet fit in one page of width

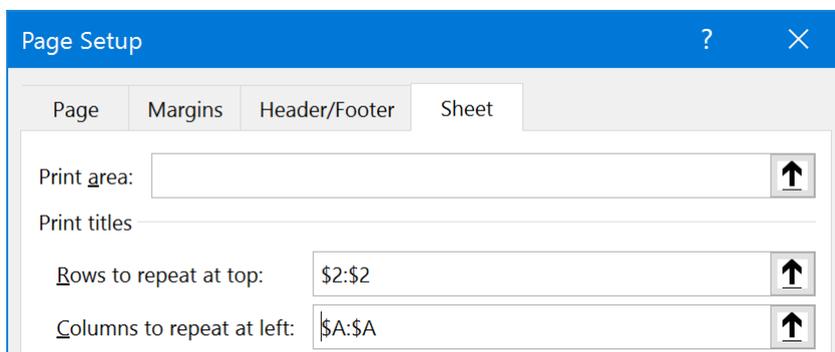
This is probably the first thing you should do when preparing most documents for printing. Go to the "Page Layout" tab in the ribbon, and look at the "Width" and "Height" boxes



Make the "Height" box "Automatic, to allow your document have as many pages as it needs. You can do this directly in Excel when using Templates to create your documents, or you can do this in FlexCel API by setting `ExcelFile.PrintToFit = true`, `ExcelFile.PrintNumberOfHorizontalPages = 1`, and `ExcelFile.PrintNumberOfVerticalPages = 0`.

Repeating Rows and Columns at the top

Another useful thing you can do is press the "Print Titles" button to access the "Page Setup" dialog. There you can set up some rows and columns to be repeated on every page.



This way your tables can keep their headers in every page. By the way, while you are in the "Sheet" tab, you might want to look at the option to print the gridlines or the column and row headings (The "A", "B", etc. at the top and "1", "2", etc. numbers at the left of the sheet)

You can do this directly in Excel when using Templates to create your documents, or you can do this in FlexCel API by writing something like this:

```
xls.SetNamedRange(new TXlsNamedRange(TXlsNamedRange.GetInternalName(InternalNameRange.Print_Titles), SheetIndex, 0, "=1:2,A:B"));
```

to set up the rows and columns to repeat, or set `ExcelFile.PrintGridLines = true` and `ExcelFile.PrintHeadings = true` to set up printing of gridlines and headings.

Using Page Headers/Footers

Besides repeating rows and columns, you can also add headers and footers from the page setup dialog. One interesting feature in Excel XP or newer is the ability to include images in the page headers or footers. As those images can be transparent, you can have a lot of creative ways to repeat information on every sheet.

From FlexCel API, use the `ExcelFile.PageHeader` and `ExcelFile.PageFooter` properties to set the header and footer text. If using a template, you can just set those things in the template.

Cell indentation

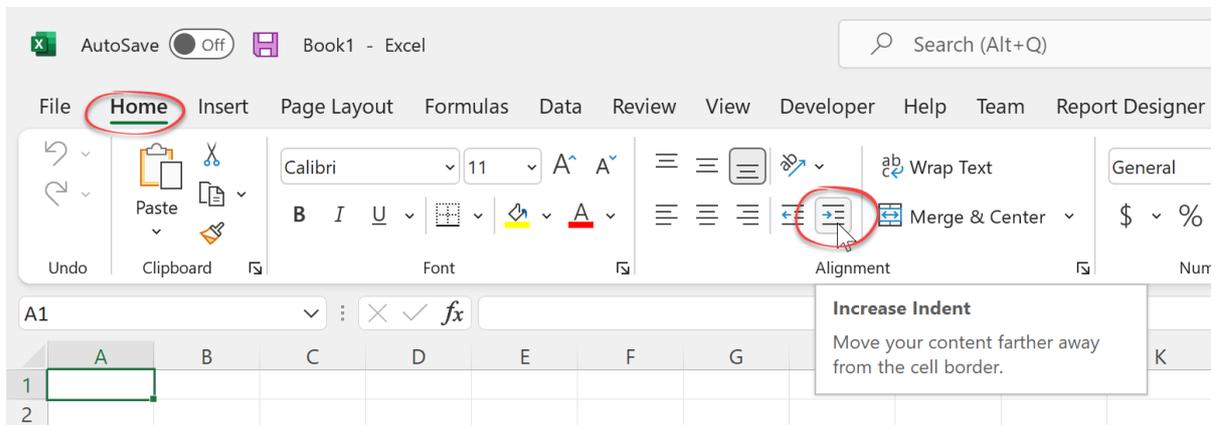
The problem

Cell indentation in Excel doesn't work the way most of us would expect it, and in this section we will cover what is wrong and how we tried to fix it.

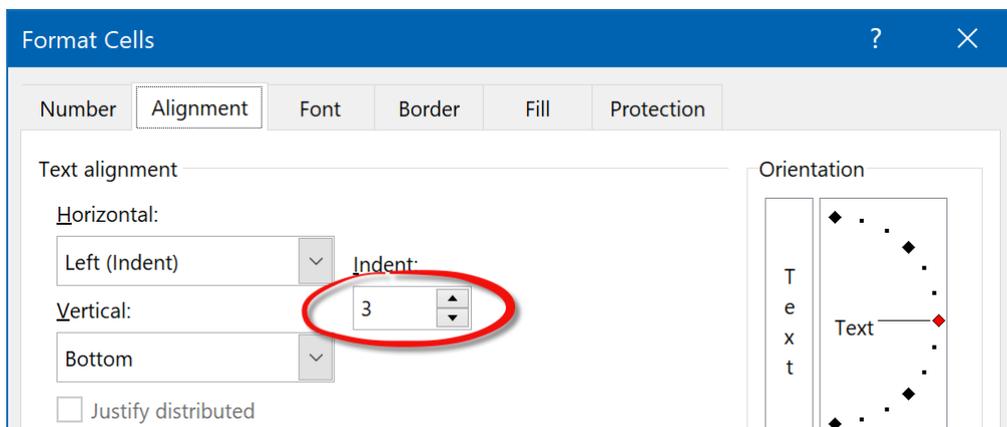
To understand the problem, let's create a new blank document in Excel, and write the word "HELLO" in cells A1 and A2:

	A	B
1	HELLO	
2	HELLO	
3		

Now, let's give the cell A2 an indentation of 3. You can do this by clicking the indent button in the ribbon 3 times:



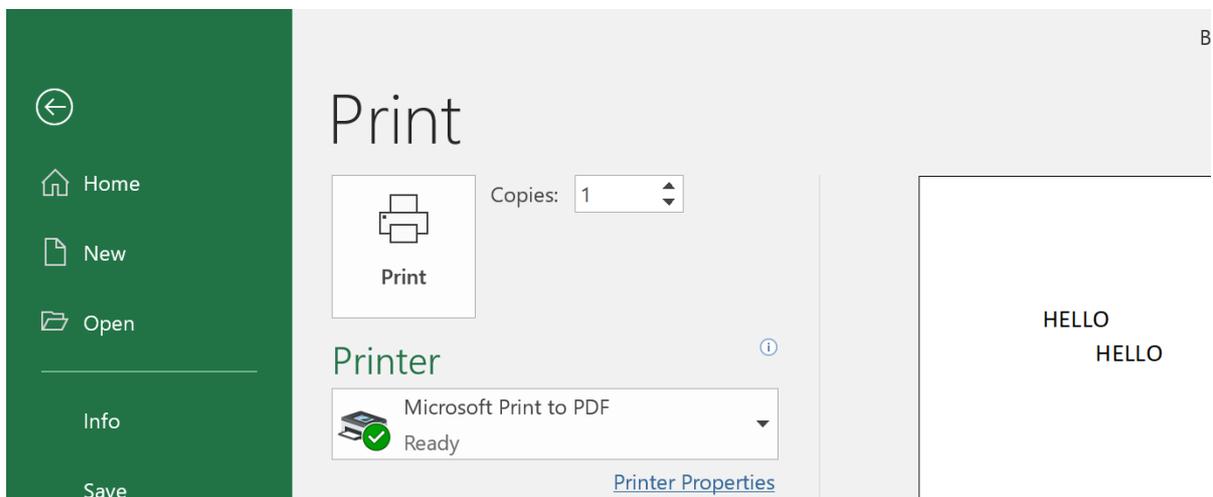
Or by right-clicking the cell, selecting "Format Cells..." and in the "Alignment" tab setting the indent to 3:



No matter how you did it, now the file should look like this:

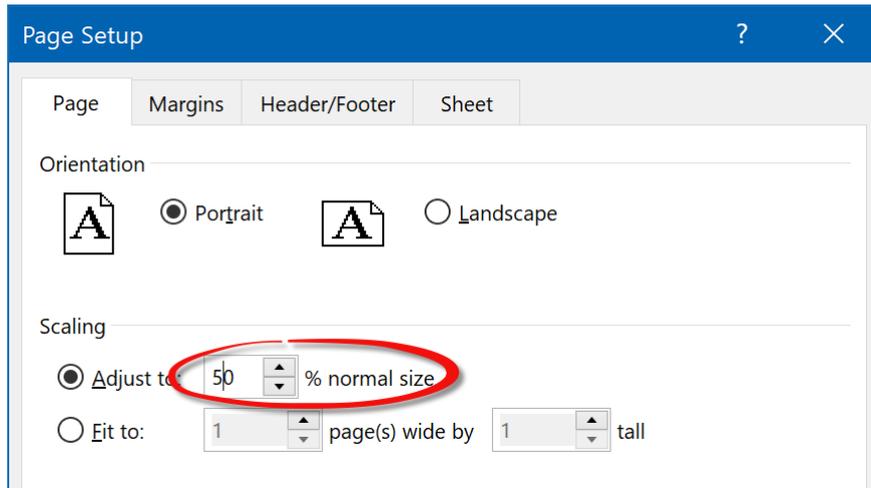
	A	B
1	HELLO	
2	HELLO	
3		

Now let's try to do a print preview. Go to File->Print and it should look like this:

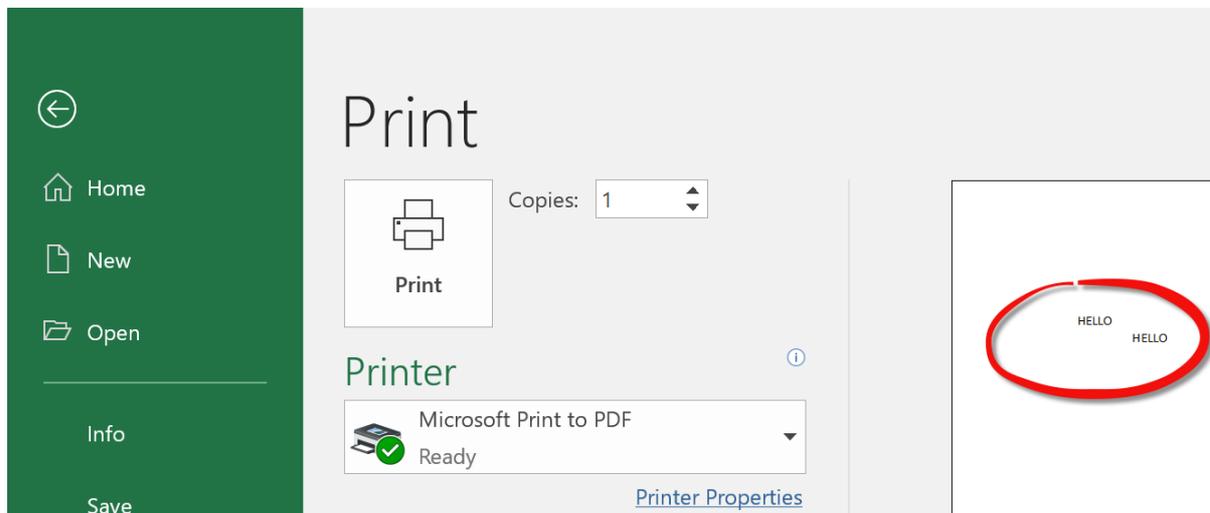


As you can see, the image looks similar (if not exactly the same) to what was shown on the screen, with the "H" in cell A2 roughly below the "O" in cell A1. We are used to small differences between what you see and what is printed, so up to now, everything is fine.

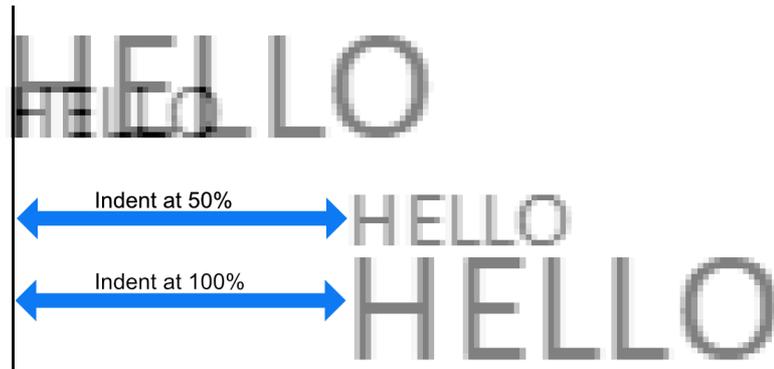
But now, let's change the **Print Scale** to 50%. Go to the page setup dialog, and change the print scale:



And now we get:



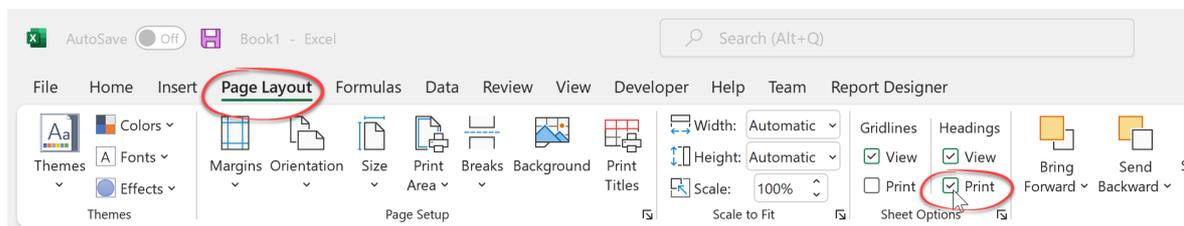
Now, the "H" in cell A2 is way more to the right from the "O" in A1 than before. What happened? We just changed the print scaling, and the layout of the document changed. This is not how we generally expect scaling to work. Going a little more in depth, what happened was that Excel reduced all element sizes, fonts, etc. by 50%, but left the indentation the same. We can see it more clear if we superimpose the 50% and 100% print preview images and zoom in:



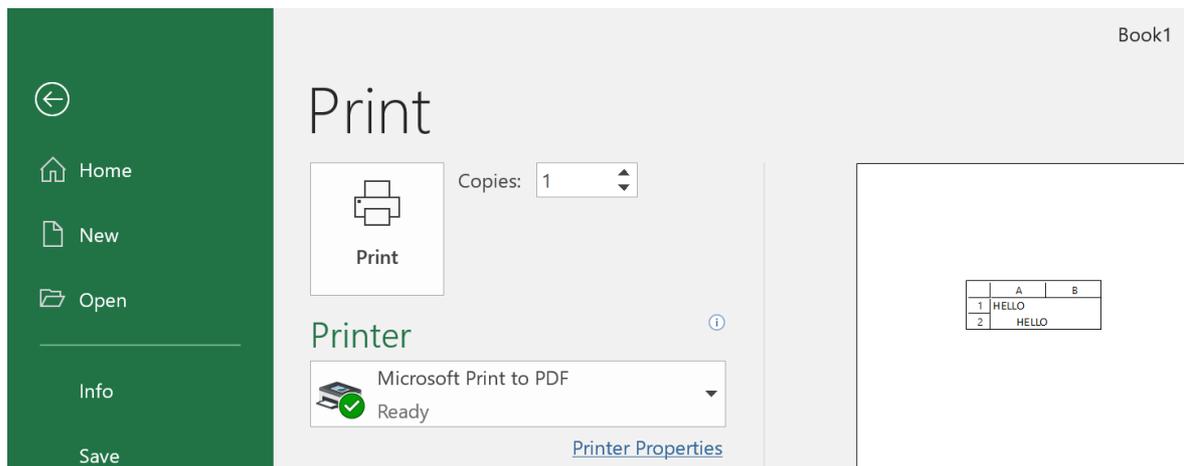
The **absolute** value of the indentation remained the same, and didn't reduce by 50% like everything else. This causes the layout to break.

NOTE

When "Print Headings" is selected:



Then Excel behaves differently and it scales the indentation proportional to the print scale:



But printing with "print headings" on isn't common, so for most cases, we can say that the indentation doesn't scale with the print scale.

The solution

Up to version 6.23, FlexCel copied the Excel behavior and didn't scale the cell indentation together with the rest of page elements (it wasn't the exact Excel behavior as FlexCel didn't scale either even if **print page headings** was selected, but printing with page headings is rare).

This original FlexCel behavior matched Excel (and FlexCel has always been about matching Excel even with Excel bugs), but it caused a lot of issues. The most evident one, of course, was that the layout would break when changing the page scaling, and text that was aligned at 100% scaling would become unaligned at 75%.

But there were more subtle issues too: For example FlexCel autofitting runs at 100% scaling (because you can't know at autofitting time which will be the final page scaling). So if you autofitted a column in a page designed to be printed at 75% scaling, it might happen that the exported pdf file couldn't fit the text anymore and had to be printed in 2 lines. Because the autofit was done at 100% and the exporting at 75%, and the indentation is not proportional to the scale like everything else, the free space in the cell is smaller at 75%.

So starting with FlexCel 6.24, we introduced a new Property named [ExcelFile.CellIndentationRendering](#) which you can now use to tell FlexCel how to deal with cell indentation and scaling. It offers 3 options:

- The new default behavior: Always scale indentation with page scale.
- The old FlexCel behavior: Never scale indentation with page scale.
- The Excel behavior: Scale indentation only if printing headings, don't scale otherwise.

IMPORTANT

This is a **breaking change** in FlexCel 6.24, in that it might break "pixel-perfect" layouts of files with small zoom and cell indentations. It isn't likely to affect you because the new indentations are smaller at smaller zooms, and so more text will fit than before. But if you want to revert to the old behavior, just change [ExcelFile.CellIndentationRendering](#)

Conclusion

What we can get from this section is: If you can control the files you are creating, **try to avoid using cell indentation in Excel**. Cell indentation in Excel is simply broken, and we can fix it on our side, but we can't fix Excel.

Cell indentation also doesn't work all all with Google docs (at the time of this writing), and in Libre/OpenOffice it scales with the print scaling. So using cell indentation will also reduce the portability of your files to other Spreadsheet apps.

If you can't avoid cell indentation, review which value of [ExcelFile.CellIndentationRendering](#) is better for you.

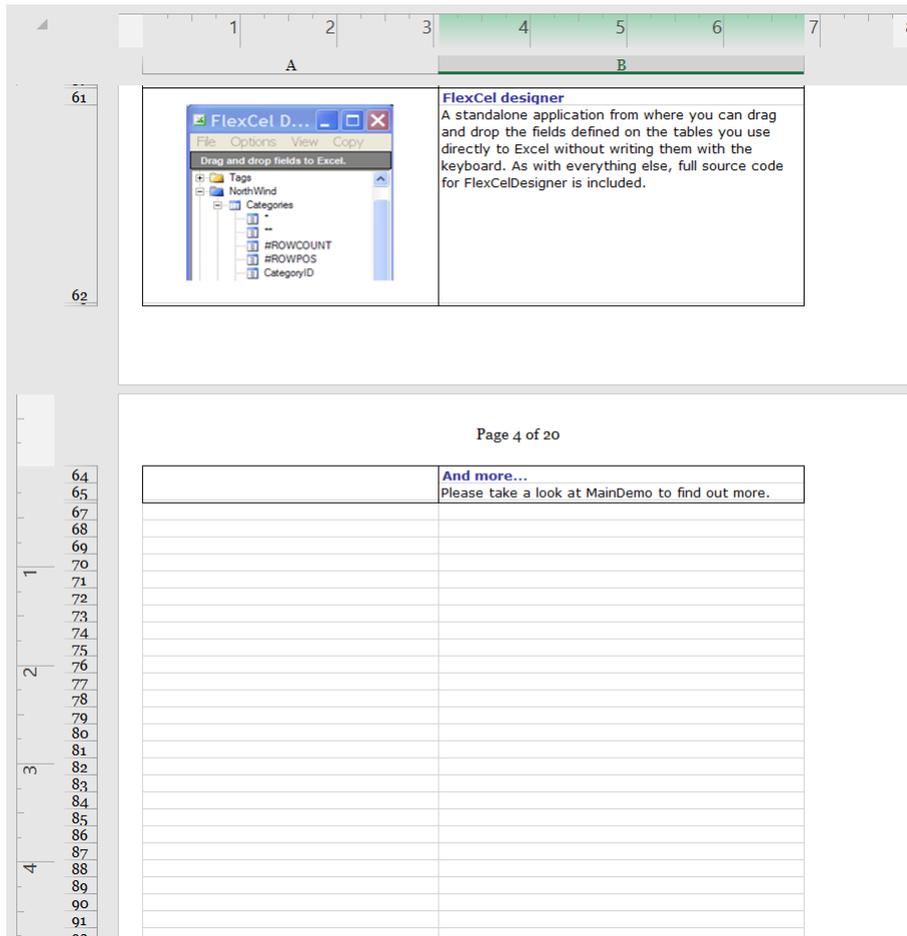
Intelligent Page Breaks

Excel offers the ability to include page breaks, but it has no support for any kind of "smart" page breaks that will only break if there is a need for it.

Let's look at an example:

The Widow / Orphan problem

When paginating a document there are actually two similar but different things: **widow** and **orphan** lines. In the example above we saw orphan lines, that is, rows that have their “parents” in the previous sheet. But there are also widow lines, and those are rows that are alone in a sheet because the lines below are in a different group, as shown in the following image:



In this example, sheet 4 is almost empty, because there are only two lines from the current group on it, and the next group starts at the next page.

When there is so little written on one page, you will normally want to start the next group on the same page instead of having an empty sheet.

And you can control this in FlexCel with the “PercentOfUsedSheet” parameter when you call [AutoPageBreaks](#). **PercentOfUsedSheet** defaults at 20, which means that in order to add a page break, the page must be filled in at least 20%.

In the example above, no page break would be made on page 4, since the 20% of the sheet has not been used yet, and so the next group would start on page 4 too. If you set the PercentOfUsedSheet parameter to 0% there will be no widow control, and the next group will start on page 5 no matter if page 4 has one single line.

Setting PercentOfUsedSheet at 100% means no widow or orphan control at all, since in order to set a page break the 100% of the page must have been used, so FlexCel has no margin to add the page breaks. It will not be able to set the page break somewhere before the 100% of the page, so all page breaks will be set at the last row, and it will not be able to keep any rows together. You are advised to keep the PercentOfUsedSheet parameter around 20%

The different printers problem

As explained in the [Autofitting Rows and Columns](#) section, Excel prints different things to different printers, and even with the same paper size, some printers have different printing sizes. Printers can have "hard margins" which are places of the sheet where they can't physically print (for example because that part of the sheet is what the printer uses to grab the paper) and this can reduce the effective printing size of the page.

This can be problematic, since a document calculated to have breaks every 30 cm, will have a lot of widow lines if printed in a page with an effective 29.5 cm printing area:

Page 2 of 20

Support for calendar and row report
FlexCel can automatically or vertically, allowing for multiple reports. You can use a calendar and include a row to create a cross reference report.

Support for any level of master detail
You can sort any number of reports, and FlexCel will automatically read the relationship between the reports.

Multiple sheet reports
Use as many sheets or worksheets as you want. You can sort them independently from the other, or make a master detail report where each sheet holds a different master record.

FlexCel
You can include a report inside another, allowing for the maximum of flexibility and modularity. This way you can use the same parts again and again. For example, you could have a Header, set file with the header of your report include all of them. So when you need to change the header, you can just change the header and all your reports will have the new header.

Row and Column Expressions
You can use (let your choice) full row or column expressions inside the reports to manage the text.

Formulas with text
You can use the &Formula to test replace a text inside a formula.

Custom expression sheets
As you can define your own tables, reports, reports and more. For example, you can define a table &Table() to use the new &Table in the template instead of the usual expressions. Everything done on the template, and can be modified by the final user.

Generic Report
For the case where you want to just dump a dataset, you can use the &Table() tag.

Direct SQL
You can make the SQL for your reports directly on the template, allowing for a "zero code" report. Changing anything on the report only implies changing the template.

Use a Table
If you would not prefer to embed direct SQL, but would like to be able to define the queries on the template, you can use an user table event. This allows for more control and environment than Direct SQL, while still allowing to change the data layer without modifying the code.

Page 3 of 20

Use a Table & Function
You can add your own functions on the code to the already built implementation of FlexCel, and use them as native functions in your report.

Use a Table & Function
You can use the &Function to test replace a text inside a formula.

30 cm effective page size

Page 2 of 20

Support for calendar and row report
FlexCel can automatically or vertically, allowing for multiple reports. You can use a calendar and include a row to create a cross reference report.

Support for any level of master detail
You can sort any number of reports, and FlexCel will automatically read the relationship between the reports.

Multiple sheet reports
Use as many sheets or worksheets as you want. You can sort them independently from the other, or make a master detail report where each sheet holds a different master record.

FlexCel
You can include a report inside another, allowing for the maximum of flexibility and modularity. This way you can use the same parts again and again. For example, you could have a Header, set file with the header of your report include all of them. So when you need to change the header, you can just change the header and all your reports will have the new header.

Page 3 of 20

Row and Column Expressions
You can use (let your choice) full row or column expressions inside the reports to manage the text.

Formulas with text
You can use the &Formula to test replace a text inside a formula.

29.5 cm effective page size

As you can see in the images above, reducing a little the page height might cause the last row not to enter on that page and be printed in the next. As FlexCel added an automatic page break after that row (so it would break correctly with the 30 cm it used for the calculation), you end up with an empty page with only one row.

To solve this issue, the second parameter to AutoPageBreaks is the percentage of the page that will be used by FlexCel to calculate the page breaks. It defaults at 95%, which means that it will consider a page 30 cm tall to be $30 * 0.95 = 28.5$ cm, and calculate the page breaks for that sheet. So it will print correctly in a 29.5 cm sheet.

When calculating the page breaks to directly export to PDF you can keep this parameter at 100%, since FlexCel is resolution independent, and it will not have this issues. But when calculating the page breaks to print from Excel, you need to have a smaller value here so it can fit in all the printers where you might print this sheet. Normally if targeting only laser printers you can have a larger value like 99%, but when other kinds of printers can be used it might be wise to lower this value.

Using different levels of “Keep together”

Normally just marking the rows you want to keep together and then calling [AutoPageBreaks](#) should be enough for most cases. But sometimes you might have more than one “keep together” level to keep, and you can set this as one of the parameters in `KeepRows/ColsTogether`.

Imagine you have a master-detail report with a list of stores, then the customers of each store and the orders for each customer. You might want to keep all customers of a store together on one page, but if that is not possible, at least keep the orders together. You can do this by assigning a level of “2” to the orders, and a level of “1” to the customers. The actual numbers don't matter, you could use a level of “10” for orders and “5” for customers. The only important thing is that one level is bigger than the other.

When you assign different levels to different rows, FlexCel will try to keep lower levels first, but if not possible, it will try to fit at least the higher ones.

Inserting and copying Rows with a “keep together” level

Inserting and copying rows works as expected, when you copy a group of rows with a level of 2 and insert them in another place, the “2” level will be copied. It also works when copying across different files, but you need to copy **full rows** in any case. When copying ranges that are not the complete rows, those values will not be copied, as expected.

Debugging Intelligent Page Breaks

Intelligent page breaks are a tested technology: They work well and we aren't aware of bugs on its implementation. But sometimes, they might produce results that look wrong at first sight. And when they aren't working as you expect them to, it helps to understand a little more in depth what is going on under the hood.

NOTE

For simplicity in this section we are going to speak always about row breaks, but the column breaks work in a similar fashion.

The first thing to notice is that intelligent page breaks work by marking the rows (or columns) to be kept together with a level. So let's imagine we have this file: A master-detail report with the best selling albums in each decade and some of their songs:

	A	B	C	D	E	F	G
1	1970-1979						
2	Eagles: Their Greatest Hits (1971–1975)						
3				Track	Title	Lead vocals	Length
4			1	Take It Easy	Frey	3:29	
5			2	Witchy Woman	Henley	4:10	
6			3	Lyin' Eyes	Frey	6:21	
7			4	Already Gone	Frey	4:13	
8			5	Desperado	Henley	3:33	
9	Pink Floyd: The Dark Side of the Moon						
10				Track	Title	Lead vocals	Length
11			1	Speak to Me	instrumental	1:30	
12			2	Breathe	David Gilmour	2:43	
13			3	On the Run	instrumental	3:30	
14			4	Time	Gilmour / Wright	6:53	
15			5	The Great Gig in the Sky	Clare Torry	4:15	
16	1980-1989						
17	Michael Jackson: Thriller						
18				Track	Title	Lead vocals	Length
19			1	Wanna Be Startin' Somethin'	Jackson	6:02	
20			2	Baby Be Mine	Jackson	4:20	
21			3	The Girl Is Mine	Jackson / McCartney	3:41	
22			4	Thriller	Jackson	5:57	

We want to keep together every decade on its own page, or if not possible, at least keep together every full album. For this we can define the following keep together ranges:

1. Rows 1 to 15, level 1
2. Rows 16 to 22, level 1
3. Rows 2 to 8, level 2
4. Rows 9 to 15, level 2
5. Rows 17 to 22, level 2

Or with code:

```
xls.KeepRowsTogether(1, 15, 1, false);
xls.KeepRowsTogether(16, 22, 1, false);
xls.KeepRowsTogether(2, 8, 2, false);
xls.KeepRowsTogether(9, 15, 2, false);
xls.KeepRowsTogether(17, 22, 2, false);
```

TIP

Of course the code above is not how you will set the intelligent page breaks in normal use. In real code you will be using some kind of loop, or inserting and copying rows, or setting keep together ranges in reports. But the effect once that is done is similar to manually calling [ExcelFile.KeepRowsTogether](#) with the values above.

To better understand what is happening, let's see now how FlexCel translates those `ExcelFile.KeepRowsTogether` commands into its data model. Conceptually it is simple: When you call `ExcelFile.KeepRowsTogether` for rows 1 to 15 to be level 1 (as in the first line of the example above), FlexCel will mark rows **1 to 14** to be level 1. There was no typo in the previous line: Marking rows 1 to 15 to be level 1 makes FlexCel mark rows 1 to 14 to be level 1. If we add a small column to visualize the marked rows after marking rows 1 to 15, the file should look like this:

	A	B	C	D	E
1	1	1970-1979			
2	1	Eagles: Their Greatest Hits			
3	1	Track		Title	
4	1		1	Take It Easy	
5	1		2	Witchy Woman	
6	1		3	Lyn' Eyes	
7	1		4	Already Gone	
8	1		5	Desperado	
9	1	Pink Floyd: The Dark Side of the Moon			
10	1	Track		Title	
11	1		1	Speak to Me	
12	1		2	Breathe	
13	1		3	On the Run	
14	1		4	Time	
15	0		5	The Great Gig in the Sky	
16	0	1980-1989			
17	0	Michael Jackson: Thriller			
18	0	Track		Title	
19	0		1	Wanna Be Startin' So	
20	0		2	Baby Be Mine	

So why is FlexCel marking one less row than what you marked? This should become clear when we call the second `KeepTogether` line to keep rows 16 to 22. After that call, the file will look like this:

	A	B	C	D
1	1	1970-1979		
2	1	Eagles: Their Gr		
3	1	Track		Title
4	1		1	Take It E
5	1		2	Witchy \
6	1		3	Lyin' Eye
7	1		4	Already
8	1		5	Despera
9	1	Pink Floyd: The		
10	1	Track		Title
11	1		1	Speak to
12	1		2	Breathe
13	1		3	On the F
14	1		4	Time
15	0		5	The Gre.
16	1	1980-1989		
17	1	Michael Jackson		
18	1	Track		Title
19	1		1	Wanna I
20	1		2	Baby Be
21	1		3	The Girl
22	0		4	Thriller
23				

Note how there is a 0 at row 15. If the first call to KeepRowsTogether had marked rows 1 to 15 instead of 14, then row 15 would also have a 1 there. And the full rows 1 to 22 would be marked as level 1, the same as if we had just called KeepRowsTogether from row 1 to 22. But we didn't make a single call from rows 1 to 22. We marked rows 1 to 15 and 16 to 22. We want FlexCel to know that it can break the page at row 15 if it needs to. When running the actual page break algorithm, FlexCel will take into account that missing row, and still know that row 15 belongs together with row 14 even if internally row 15 is marked with level 0.

Marking one less row than the rows you mark allows you to mark 2 consecutive ranges like 1:5 and 5:10 and have them still be separated ranges instead of becoming a single 1:10 range

IMPORTANT

You might be wondering why we spent so much time explaining an implementation detail like how FlexCel maps your calls to `KeepRowsTogether` to its internal data model. After all, it shouldn't really matter how FlexCel internally models your commands. And in most cases, it won't actually matter.

But this section is about debugging page breaks, and you will need to debug them when something went wrong. There are 2 reasons why it helps to understand the internal representation FlexCel does from your commands:

1. The fact that when you mark rows 1 to 2 only row 1 is marked is the **most common** cause of discrepancies between what you think FlexCel should do and what FlexCel actually does. In most cases, it will work just as you expect it to work and you don't need to think in the internals. Just mark the block you want to keep together and know they will be kept together. But if it is not working as it should, then it likely is caused by this.
2. As we will see shortly, when debugging reports you will be able to see this internal FlexCel representation, so you will need to understand how it is implemented.

Ok, now let's run the full code and add the lines that call `KeepRowsTogether` with level 2. This is how the file will end up marked:

	A	B	C	D	E	F	G	H
1	1	1970-1979						
2	2	Eagles: Their Greatest Hits (1971–1975)						
3	2		Track	Title		Lead vocals	Length	
4	2		1	Take It Easy		Frey	3:29	
5	2		2	Witchy Woman		Henley	4:10	
6	2		3	Lyin' Eyes		Frey	6:21	
7	2		4	Already Gone		Frey	4:13	
8	1		5	Desperado		Henley	3:33	
9	2	Pink Floyd: The Dark Side of the Moon						
10	2		Track	Title		Lead vocals	Length	
11	2		1	Speak to Me		instrumental	1:30	
12	2		2	Breathe		David Gilmour	2:43	
13	2		3	On the Run		instrumental	3:30	
14	2		4	Time		Gilmour / Wright	6:53	
15	0		5	The Great Gig in the Sky		Clare Torry	4:15	
16	1	1980-1989						
17	2	Michael Jackson: Thriller						
18	2		Track	Title		Lead vocals	Length	
19	2		1	Wanna Be Startin' Somethin'		Jackson	6:02	
20	2		2	Baby Be Mine		Jackson	4:20	
21	2		3	The Girl Is Mine		Jackson / McCartney	3:41	
22	0		4	Thriller		Jackson	5:57	

When FlexCel runs the intelligent page breaks in this page it will:

1. Try to fit all level 1+ rows on one page, adding an extra row because the rows marked are one less. So, the first block of levels ≥ 1 go from 1 to 14. Adding the extra row, FlexCel will try to fit the block **1 to 15** in a single page.
2. If it can't fit rows 1 to 15 in a single page, it will try with level 2+. The first block of level 2+ goes from rows 2 to 7. So it will try to fit the rows **1 to 8** in a single page. And so on.

After diving in the technical details, we can see it is doing what we told it to do. It first tries to fit rows 1 to 15 (the first KeepRowsTogether call), then 2 to 8 (the third KeepRowsTogether call) and then the others.

Now, wouldn't it be nice to be able to directly see FlexCel's internal representation for those cases where the page breaks aren't working as they should? Without having to do it manually as we did here?

The good news is that yes, it is possible. FlexCel has commands for both the API and reports that let you see the levels that FlexCel sees. **To understand those levels, you just need to remember that the last line of each keep together block isn't marked, but considered anyway to belong to the group.**

When using the API, FlexCel has 2 commands: [ExcelFile.DumpKeepRowsTogetherLevels](#) and [ExcelFile.DumpKeepColsTogetherLevels](#). The first one will write all row levels in the column you specify, and the second will do the same for columns. Looking at the actual numbers should help understanding how FlexCel sees your blocks.

When using reports you have two ways to enter intelligent page breaks debug mode, similar to [Report Debug mode and Errors in result file modes](#):

1. You can set [FlexCelReport.DebugIntelligentPageBreaks](#) to true in the code.
2. You can write a tag `<#Debug Intelligent Page Breaks>` anywhere in the Expressions column of the config sheet.

As with the debug mode and Errors in Result File mode, the second way to enter Debug Intelligent Page Breaks mode is better when you are editing a template and want to do a debug without modifying the code, while the first way is better if you are automating testing and do not want to modify the templates.

NOTE

When inserting and deleting rows, FlexCel doesn't just insert or delete them, but adapts the levels so they still behave as you would expect. This can cause issues understanding why a row ended up with a level. For example, if in our last screenshot we removed row 8, a naive implementation would just remove that row, and the group that ends at row 8 (Eagles) would be joined with the group that starts at row 9 (Pink Floyd). But when you remove that row, FlexCel is smart enough to make row 7 level 1, so the file still behaves as it should and you still have one group for the Eagles and one for Pink Floyd. The fact that you removed a song from the Eagles group is no reason for it to become one big group with Pink Floyd.

While FlexCel always tries to do "the right thing", sometimes the right thing might not be what you expected, or it might not be the right thing for a specific situation. This is why being able to see the actual values of levels added by FlexCel can be so helpful.

Using Excel's User-defined Functions (UDF)

Introduction

User-defined functions in Excel are macros that return a value, and can be used along with internal functions in cell formulas. Those macros must be defined inside VBA Modules, and can be in the same file or in an external file or addin.

While we are not going to describe UDFs in detail here, we will cover the basic information you need to handle them with FlexCel. If you need a more in-depth documentation in UDFs in Excel, there is a lot of information on them everywhere.

So let's start by defining a simple UDF that will return true if a number is bigger than 100, false otherwise.

We need to open the VBA editor from Excel (Alt-F11), create a new module, and define a macro inside that module:

```
Function NumIsBiggerThan100(Num As Single) As Boolean
    NumIsBiggerThan100 = Num > 100
End Function
```

And then write “=NumIsBiggerThan100(120)” inside a cell. If everything went according to the plan, the cell should read “True”

Now, when recalculating this sheet FlexCel is not going to understand the “NumIsBiggerThan100” function, and so it will write **#Name?** in the cell instead. Also, if you want to enter “=NumIsBiggerThan100(5)” say in cell A2, FlexCel will complain that this is not a valid function name and raise an Exception.

In order to have full access to UDFs in FlexCel, you need to define them as a class in .NET and then add them to the recalculation engine.

Step 1: Defining the Function in C#

To create a function, you need to derive a class from [TUserDefinedFunction](#), and override the [Evaluate](#) method. For the example above, we would create:

```

public class NumIsBiggerThan100 : TUserDefinedFunction
{
    public NumIsBiggerThan100() : base("NumIsBiggerThan100")
    {
    }

    //Do not define any global variable here.
    public override object Evaluate(TUdfEventArgs arguments,
        object[] parameters)
    {
        //Check we have only one parameter
        TFlxFormulaErrorValue Err;
        if (!CheckParameters(parameters, 1, out Err))
            return Err;

        //The parameter should be a double.
        double Number;
        if (!TryGetDouble(arguments.Xls, parameters[0],
            out Number, out Err)) return Err;

        return Number > 100;
    }
}

```

As you can see, it is relatively simple. Some things worth noting:

1. **Don't use global variables inside Evaluate:** Remember, the Evaluate() method might be called more than once if the function is written in more than one cell, and you cannot know the order in which they will be called. Also if this function is registered to be used globally, more than a thread at the same time might be calling the same Evaluate method for different sheets. The function must be **stateless**, and must always return the same value for the same arguments.
2. As you can see in the help for the [TUserDefinedFunction.Evaluate](#) method, the evaluate method has two arguments. The first one provides you with utility objects you might need in your function (like the XlsFile where the formula is), and the second one is a list of parameters, as an array of objects.

Each object in the parameters array might be a **null**, a **Boolean**, a **String**, a **Double**, a **TXls3DRange**, a **TFlxFormulaErrorValue**, or a 2 dimensional array of objects, where each object is itself of one of the types mentioned above.

While you could manually check for each one of the possible types by manually checking all possible types for each parameter, this gets tiring fast. So the [TUserDefinedFunction](#) class provides helper methods in the form of "TryGetXXX" like the [TUserDefinedFunction.TryGetDouble](#) method used in the example.

3. There is the convention in Excel that when you receive a parameter that is an error, you should return that parameter to the calling function. Again, this can be tiring to do each time, so the [TUserDefinedFunction](#) class provides a [TUserDefinedFunction.CheckParameters](#) method that will do it for you.

The only time you will not call [TUserDefinedFunction.CheckParameters](#) as the first line of your UDF is when you are creating a function that deals with errors, like "IsError(param)", that will return true when the parameter is an error.

4. **Do not throw Exceptions.** Exceptions you throw in your function might not be trapped by FlexCel, and will end in the recalculation aborting. Catch all expected exceptions inside your method, and return the corresponding [TFlxFormulaErrorValue](#) when there is an error.

Step 2: Registering the UDF in FlexCel

Once you have defined the UDF, you need to tell FlexCel to use it. You do this by calling [ExcelFile.AddUserDefinedFunction](#) in a [ExcelFile](#) object. Once again, some points worth noting:

1. You need to define the scope of your function. If you want it to be globally accessible to all the [ExcelFile](#) instances in your application, call [ExcelFile.AddUserDefinedFunction](#) with a "Global" [TUserDefinedFunctionScope](#). If you want the function to be available only to the [ExcelFile](#) instance you are adding it to, specify "Local". **Global** scope is easier if you have the same functions for all the files, but can cause problems if you have different functions in different files. **Local** scope is safer, but you need to add the functions each time you create a [ExcelFile](#) object that needs them. If you are unsure, probably local scope is better.
2. You also need to tell FlexCel if the function will be defined inside the same file or if it will be in an external file. This is not really needed for recalculating, but FlexCel needs to know it to enter formulas with custom functions into cells.

There are four things you can do with formulas that contain UDFs, and there are different things you need to do for each one of them:

1. **Retrieve the formula in the cell:** You do not need to do anything for this. FlexCel will always return the correct formula text even if you do not define any udf. If the file was calculated when saved in Excel (the default), then FlexCel will also return the correct value of the formula.
2. **Copy a cell from one place to another.** Again, there is no need to do anything or define any UDF object. FlexCel will always copy the right formula, even if copying to another file.
3. **Calculate a formula containing UDFs.** For this one you need to define a UDF class describing the function and register it. You do not need to specify if the formula is contained in the sheet or stored in an external addin.
4. **Enter a formula containing a UDF into a sheet.** In order to do this, you need to define and register the UDF, and you must specify if the function is stored internally or externally.

For more examples on how to define your own UDFs, please take a look at the [Excel User Defined Functions](#) API demo.

NOTE

Even when they look similar, UDFs are a completely different thing from Excel Built-in functions. Everything said in this section applies only to UDFs, you cannot use this functionality to redefine a standard Excel function like "Sum". If recalculating some built-in function is not supported by FlexCel just let us know and we will try to add the support, but you cannot define them with this.

Returning Arrays from UDFs

If you are using array formulas, you might need to return an array from your user-defined function. It is no really different from the standard UDFs, but you need to return a 2-dimensional array of objects from the [Evaluate](#) method.

So if we wanted to define a simple UDF that returns an array with 4 fixed numbers, we could define a function like the following:

```
public class HelloFlexCellImpl : TUserDefinedFunction
{
    public HelloFlexCellImpl() : base("hello_flexcel")
    {
    }

    //Do not define any global variable here.
    public override object Evaluate(TUdfEventArgs arguments, object[] parameters)
    {
        //Check we have no parameters
        TFlxFormulaErrorValue Err;
        if (!CheckParameters(parameters, 0, out Err))
            return Err;

        // Note that we return a 2-dimensional array of objects,
        // even if the array is 1-dimensional.
        return new object[,] { { 8, 9, 10, 11 } };
    }
}
```

And then, in FlexCel we could use the following code to enter formulas that use the UDF:

```
xls.AddUserDefinedFunction(TUserDefinedFunctionScope.Global, TUserDefinedFunctionLocation.External, new HelloFlexCellImpl());
xls.SetCellValue(2, 1, new TFormula("={HELLO_FLEXCEL()}", null, new TFormulaSpan(1, 4, true)));
```

Recalculating Linked Files

FlexCel offers full support for recalculating linked files, but you need to add some extra code to make it work. Everything we will discuss here is about [ExcelFile](#) objects, but you can also use this on reports. Just run the reports with [FlexCelReport.Run\(xls\)](#).

The main issue with linked files is telling FlexCel where to find them.

Normally, if your files are on a disk, there should be not much problem to find a linked file. After all, that information is inside the formula itself, for example if FlexCel finds the formula:

```
= '..\Data\[linked.xls]Sheet1'!$A$3 * 2
```

inside a cell, it could automatically find “..\Data\linked.xls”, open it, and continue with the recalculation.

In fact, there are two problems with that approach:

1. Files might not be in a filesystem. FlexCel allows you to transparently work with streams instead of physical files, and so you might have for example the files stored in a database. In this case, trying to find “..\Data\linked.xls” makes no sense, and FlexCel would fail to recalculate.
2. Much more important than problem 1, this approach could imply an important **security risk**.

Blindly following links in an unknown xls file is not a smart idea. Let's imagine that you have a web service where your user submits an xls file, you use FlexCel to convert it to PDF, and send back the converted PDF to him. If that user knows the location of an xls file in your server, he could just submit a hand-crafted xls file filled with formulas like:

```
= 'c:\Confidential\[BusinessPlan.xls\]Sheet1'!A1,  
= 'c:\Confidential\[BusinessPlan.xls\]Sheet1'!A2,  
...
```

On return you would supply him with a PDF with the full contents of your business plan. What is even worse, since FlexCel can read plain text files, he might also be able to access any text file in your server. (Imagine you are running on Linux and formulas pointing to /etc/passwd)

NOTE

You might argue that in a well-secured server your application should not have rights on those files anyway, but on security, the more barriers and checks you add the better. So you should have a way to verify the links inside an arbitrary file instead of having FlexCel opening them automatically.

Because of reasons 1) and 2), FlexCel by default will not recalculate any linked formula, and just return “#NA!” for them. Note that in most cases you will want to leave this that way, since most spreadsheets don't have linked formulas anyway, and there is no need to add extra security risks just because.

But if you do need to support linked files, adding that support is easy.

The first thing you need to do is to create a [TWorkspace](#) object. Workspaces are collections of [XlsFile](#) objects, and when you recalculate any of the files in a Workspace, all the others will be used in the recalculation (and be recalculated) too.

So the simplest and more secure way to recalculate linked files is to create a Workspace, add the needed [XlsFile](#) objects inside, and just recalculate any of the files as usual. For example:

```

XlsFile xls1 = new XlsFile("File1.xlsx");
XlsFile xls2 = new XlsFile("File2.xlsx");
XlsFile xls3 = new XlsFile("File3.xlsx");

TWorkspace work = new TWorkspace();
work.Add("xls1", xls1);
work.Add("xls2", xls2);
work.Add("xls3", xls3);

//Either work.Recalc, xls1.Recalc, xls2.Recalc or xls3.Recalc will
recalculate all the files in the workspace.
work.Recalc(true);

```

In the example above, we opened two files and added them to a workspace, giving each one a name to be used in the recalculation process. Note that here we don't have issues 1) or 2) at all. We could have opened those files from a stream and it would be the same, since the name "file1.xls" needed to calculate is actually given in the "work.Add()" method. The actual name and location (if any) of the file "file1.xls" is irrelevant. And also we don't have the security concern of FlexCel opening files by itself, since we opened the files we wanted, and FlexCel will not open any more files.

Now, in some cases you don't know a priori which files you are going to need in order to recalculate the file, and so you cannot use the approach above. In those cases, you still use the Workspace object, but you assign an event where you load those files when FlexCel asks for them. The code would be something like:

```

XlsFile xls1 = new XlsFile();
xls1.Open(FileName);

//Create a workspace
TWorkspace Work = new TWorkspace();

//Add the original file to it
Work.Add(Path.GetFileName(FileName), xls1);

//Set up an event to load the linked files.
Work.LoadLinkedFile += new
    LoadLinkedFileEventHandler(LoadLinkedFile);

//Recalc will recalculate xls1 and all the other files used in the
recalculation.
//At the end of the recalculation, all the linked files will be loaded in the
workspace,
//and you can use the methods in it to access them.
Work.Recalc(true);

```

In the LoadLinkedFile event you can load the needed files, checking that they are not trying to access folders they shouldn't be looking to. For example, remove all path information and only load files from the same folder the original file is. Or only allow paths that are children of the path of the main file, or maybe paths from an allowed list of paths. The choice is yours.

The LoadLinkedFile event would be declared like this:

```
private static void LoadLinkedFile(object sender, LoadLinkedFileEventArgs e)
{
    //Load the file here checking it is a allowed file.
    //In this example we will just load the linked file
    //Without checking it, but this is only to keep the example simple.
    e.Xls = new XlsFile(e.FileName);
}
```

You can take a look at the [Validate Recalc](#) demo to see a real implementation of this. In that example there is no validation of the loaded filenames, but just because it is an application designed to run locally with full trust.

IMPORTANT

If XlsFile objects can consume lots of memory, Workspace objects can consume much more, since they are a collection of XlsFiles themselves. So same as with XlsFile objects, **don't leave global Workspace objects hanging around**. Set them to null as soon as possible if they are global variables, or use local objects which are freed when going out of scope.

Take also a look at the example [Recalculation of linked files](#) for more information.

Miscellanea

Using .NET languages different from C#

While FlexCel.NET runs in any .NET language, and we try to provide most demos and documentation for most languages, you will sooner or later find code snippets, newsgroups posts, etc. available only in C#, since FlexCel itself is written in C#. This will probably be no big problem since all languages under .NET behave in similar ways, but we will try in this section to help a little with it.

Even if you do not have exposure to any C-based language, C# examples are not so difficult to read. But there are some basic concepts that might help you understand them, based on the feedback we get from our users, and we will explain them there. If you already are fluent at C#, just skip this section.

Operators

C-based languages are a little less verbose than others when specifying operators, and this might be a little cryptic when you are not used to them. For example, to specify an "if" statement in C# that evaluates to true if either a and b are true or c is true you would write:

```
if ((a && b) || c)
```

while in VB.NET it would be:

```
if (a and b) or c then
```

A small list of operators you might find while reading the source code is here:

Logical Operators

Operator	Meaning	Example
!	Not	if (!a) means "If not a"
	Or	if (a b) means "if a or b"
&&	And	if (a && b) means "if a and b"

Bitwise Operators

Operator	Meaning	Example
~	Not	~ 1 means "not 1"
	Or	2 1 means "2 or 1" (this is 3)
&	And	2 & 1 means "2 and 1" (this is 0)
^	Xor	1 ^ 1 means 1 xor 1 (this is 0)

We will not make a detailed explanation of the operators here, as the only idea here is to help you read the examples, but you can read the full documentation at: [http://msdn2.microsoft.com/en-us/library/6a71f45d\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/6a71f45d(VS.71).aspx)

Automatic conversion

Many times, the easiest way to understand a code snippet in C# is just to use an automatic translator to translate it to your language. You can find many online translators just by searching, but we will mention one of them here: For VB.NET: <http://www.developerfusion.co.uk/utilities/convertcsharp2vb.aspx>

Streaming Excel and Pdf files to the browser

A simple code snippet that you are likely to use, is the one to send the generated files to the browser without creating temporary files.

You can find it in the ASP.NET demos, but just in case you need a quick reference, here is the code to stream an Excel file:

```

static void StreamToBrowser(ExcelFile xls, string fileName)
{
    using (MemoryStream ms = new MemoryStream())
    {
        // Deduce how to save the file from the filename.
        bool IsXls = Path.GetExtension(fileName).Equals(".xls", StringComparison.OrdinalIgnoreCase);
        // Xlsx files can have multiple extensions: xlsx, xlsxm, etc.
        // We will assume that if it is not xls, then is xlsx.

        TFileFormats fileFormat;
        if (IsXls) fileFormat = TFileFormats.Xls;
        else fileFormat = TFileFormats.Xlsx;

        xls.Save(ms, fileFormat);
        ms.Position = 0;
        Response.Clear();
        Response.AddHeader("Content-Disposition", "attachment;" + "filename="
+ fileName);
        Response.AddHeader("Content-Length", ms.Length.ToString());

        if (IsXls) Response.ContentType = StandardMimeType.Xls;
        else Response.ContentType = StandardMimeType.Xlsx;

        Response.BinaryWrite(ms.ToArray());
        Response.End();
    }
}

```

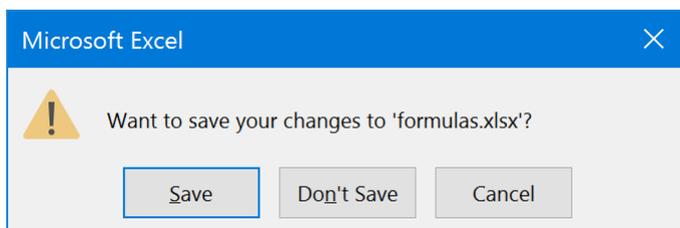
For PDF files, you would use similar code, but using **StandardMimeType.Pdf** instead.

WARNING

Make sure to use **ms.ToArray()** and not `ms.GetBuffer()`, since `GetBuffer` is not guaranteed to return everything.

Avoiding the “Want to save your changes?” dialog on close.

By default, FlexCel will not identify the xls/x files it generates as having been recalculated by any Excel version. This will cause Excel to recalculate the file on open, and when closing the file, it will show the following dialog:



This will always happen when you save a file with formulas. We choose this as the default mode because we can't know which formulas your file has, and while we support over 300 Excel functions, you might be using something we don't support, or some VBA macro function that you didn't re-implement in FlexCel, or some linked files without setting up FlexCel to recalculate linked files. So the safest default is to leave Excel to recalculate files on open, and this has the side effect of this dialog popping up when you try to close the file.

If you know the formulas you are using are all supported, and you would like to get rid of this dialog, you can tell FlexCel to identify the file as having been saved by a specific Excel version.

So you can for example do:

```
ExcelFile.RecalcVersion = TRecalcVersion.Excel2019;
```

or if using reports:

```
FlexCelReport.RecalcVersion = TRecalcVersion.Excel2019;
```

Once you do this, any Excel version equal to or older than Excel 2019 will **not** recalculate the file on open, trusting the recalculation FlexCel did, and it won't show the dialog on close.

Newer Excel versions will still recalculate and ask for save on close.

NOTE

If you want to use the always use the latest version supported by FlexCel, you can specify `TRecalcVersion.LatestKnownExcelVersion`. This value is set to the latest Excel version FlexCel knows about, so when you update FlexCel it will update automatically without you having to modify your code.

The other special value is `TRecalcVersion.SameAsInputFile`. This value means to use the value in the file you are using as a template. Let's imagine that you have an old FlexCel version which supports up to Excel 2013, and you had a file saved in Excel 2016. With this value, FlexCel will identify the file as saved as Excel 2016, even if it doesn't know about it. It will just copy the value from the input file which was in Excel 2016.

Closing Words

We hope that after reading this document you got a better idea on the basic concepts of using the FlexCel API. Concepts mentioned here (like XF format indexes) are basic to use FlexCel, so it is important that you get them right.

And one last thing. Remember that FlexCel API's main strength is that it **modifies** existing files; it doesn't use the traditional approach of one API for reading the file and another for writing. In fact, FlexCel doesn't even know how to create an empty file. When you call `ExcelFile.NewFile`, you are really reading an empty xls file embedded as a resource. **You are always modifying things.**

Take advantage of this. For example, let's say you want to save a macro on your final file. There is no support on FlexCel for writing macros. But you can create the macro on Excel and save it to a template file, and then open the template with FlexCel instead of creating a new file with `NewFile`.

Use Excel and not FlexCel to create the basic skeleton for your file. Once you have all of this on place, modify it with FlexCel to add and delete the things you need. This is what FlexCel does best. And of course, remember that you can use [FlexCelReport](#) for creating files, designing your files in Excel in a visual way.

FlexCel Reports Developer Guide

Introduction

This document is a part of a 2-part description on how to create Excel files by “reporting” instead of with code. In this part we will look at how to set up the coded needed to create the files, and in the next part [FlexCel Reports Designer Guide](#) we will look at how to modify the Excel file used as a template to create the reports.

About Excel Reporting

FlexCel gives you two ways to create an Excel file:

- With the [API](#) using the class [XlsFile](#)
- With a templating system using the class [FlexCelReport](#) .

Each method has its good and bad things, and it is good to know the advantages and drawbacks of each.

Creating a report using XlsFile is a low-level approach. As with most lower level approaches, it will be very fast to run (if coded right!), and you will have access to the entire API, so you can do whatever you can do with FlexCelReport and more. After all, FlexCelReport uses XlsFile internally to work its magic.

Whatever you can do in FlexCelReport, you can do it with the API.

But you need to take a lot of care when coding directly in the API: XlsFile reports can be a nightmare to maintain if not coded correctly. When you reach enough number of lines like:

```
xls.SetCellValue(3, 4, "Title");
var fmt = xls.GetFormat(xls.GetCellFormat(3, 4));
fmt.Font.Name = "Helvetica";
fmt.Font.Size20 = 14 * 20;

int XF = xls.AddFormat(fmt);

xls.SetCellFormat(3, 4, XF);
```

changing the report can become quite difficult. Imagine the user wants to insert a column with the expenses (and don't ask why, users always want to insert a column).

Now you should change the line:

```
xls.SetCellFormat(3, 4, XF);
```

to

```
xls.SetCellFormat(3, 5, XF);
```

But wait! You need to change also all references to column 5 to 6, from 6 to 7... If the report is complex enough, (for example you have a master-detail) this will be no fun at all.

But there is something much worse with using XlsFile directly. And this is that only the author can change the report. If your user wants to change the logo to a new one, or maybe make column C a little wider, he needs to call you and you need to recompile the application and send him a new executable.

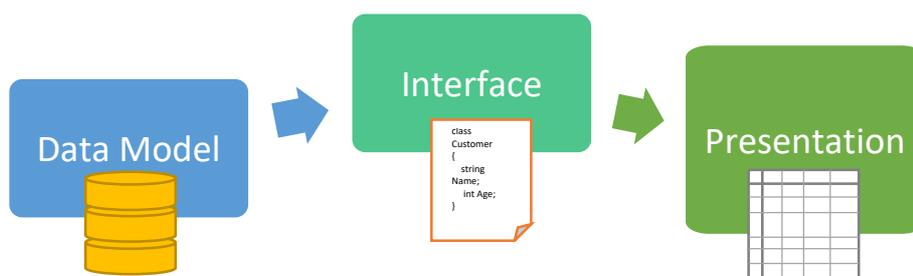
FlexCelReport is a higher level approach. The design is cleanly separated on three different layers, data layer, interface layer and presentation layer, and most of the work is done on the presentation layer, with Excel. You design the report visually on Excel, and you mess as little as possible with the data layer. If the user wants to change the logo, he can just open the template and change the logo. If he wants to insert a column, he can just open the template and insert a column. And the application does not even need to be recompiled.

As with any higher level approach, it is slower than using the API directly, and there are some things where it is more limited. But all in all, reports are really fast too and the things you cannot do are in many cases not worth doing anyway.

So, the option is yours.

Organization of a FlexCel Report

A FlexCel report can be seen as three different modules working together to create the Excel file. Different from a "Coded" report where there is not a clear separation between data access and presentation, here each part has its own place, and can be developed separately by different people with different skills.



Data Layer

This is the layer that contains the data model of the information we want to send out. The data might be stored at a database, or in lists of objects in memory.

Interface Layer

This layer works as the glue between the data and the presentation layers. It has to prepare the data in a way that the presentation layer can easily consume.

Presentation Layer

This is the most complex and changing layer of the three. Here is where you design all the visual aspects of the report, like data position, fonts, colors, etc.

The big advantage of this “layering” is that they are somehow independent, so you can work on them at the same time. Also, Data and Interface layers are small and do not change much on the lifetime of the application. Presentation does change a lot, but it is done completely in Excel, so there is no need to recompile the application each time a cell color or a position changes.

NOTE

The data flow goes **from** the Presentation layer to the Data layer and back. It is the presentation that asks FlexCel for the data it needs, and FlexCel that in turn asks for the data. It is not the application that tells FlexCel the data it needs (As in `SetCellValue(xx)`), but FlexCel that will ask for the data to the application when it needs it.

On this document we are going to speak about **Data** and **Interface** layers. The Presentation layer is discussed on a different document, [FlexCel Reports Designer Guide](#) because it is complex enough to deserve it, and because it might be changed by someone who only understands Excel, not .NET. And we don't want to force him to read this document too.

Data Layer

The objective of the data Layer is to have all the data ready for the report when it needs it. Currently, there are six ways to provide the data:

1. **Via Report Variables.** Report variables are added to FlexCel by using [FlexCelReport.SetValue](#). You can define as many as you want, and they are useful for passing constant data to the report, like the date, etc. On the presentation side you will write `<#ReportVarName>` each time you want to access a report variable.
2. **Via User defined functions:** User defined functions are classes you define on your code that allow for specific needs on the presentation layer that it can't handle alone. For example, you might define a user function named `NumberToString(int number)` that will return “One” when number=1, “Two” when number =2 and so on. On the presentation layer you would write `<#NumberToString(2)>` to obtain the string “Two”
3. **Via DataSets:** .NET DataSets are a collection of memory tables that can be sorted, filtered or manipulated. Whatever format your data is, you can use it to fill a DataTable and then use that data on FlexCel.
4. **Via IEnumerable:** If you have business objects that implement IEnumerable (or even better IQueryable) and you are in .NET 3.5 or up, you can use them directly with FlexCel. Just call [FlexCelReport.AddTable](#)(“name”, YourCollection) and you are ready to go.
5. **Via Direct SQL in the template.** For maximum flexibility, you can set up the data layer on the template too. In this case, you need to add a database connection to FlexCel, and all the rest is done on the template. This allows the users to completely edit the reports, even the data layer from the template, but also it allows users to do things they might not be intended to. Use it with care.

6. **Via Virtual Datasets:** If your business objects don't implement IQueryable, and the overhead of copying your existing objects to datasets is too much, you can implement your own wrapper objects to provide this data to FlexCel without copying it. But please take this option with care, as it might be simpler and more performing to just dump your objects into a DataSet and use that. FlexCel has an optimized implementation of a virtual dataset for DataSets, and if you create your own virtual datasets you will have to implement a similar level of performance (and functionality). Please read the Appendix II for a more detailed description of virtual datasets.

Linq based DataSources

With FlexCel you can use any object that implements IEnumerable<T> as a datasource for a report (in .NET 3.5 or newer). Internally, FlexCel will use Linq to query the objects and read the results.

To use Linq, you just need a collection of objects. For every object in the collection, you can use all the public properties as fields in the report. So if for example you have the following class:

```
class MyObject
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

And the list of MyObject:

```
var MyObjectList = new List<MyObject>();
```

You can add this collection as a datasource for the report with the line:

```
FlexCelReport.AddTable("MyName", Objs)
```

And then in the template write <#MyName.FirstName> and <#MyName.LastName> inside a "__MyName__" range to export the collection.

You can also use Entity Framework collections or any other of collection that implements IEnumerable as datasource.

NOTE

Record count: When FlexCel runs a report from an IEnumerable datasource, it needs to know the count of objects in advance, so it can insert the rows before filling the data. For normal collections like List<Object> this isn't a problem since they provide a "Count" method that will be used by Linq and it is very fast.

For Entity Framework collections, two different SQLs will be sent to the server, once to get the record count and the second to actually fetch the records. Other collections could have no way to know the record count in advance, and in those cases Linq will loop over all objects in the collection to get the count. In this third case, it will be faster if you provide the record count yourself. Look at the [Performance guide](#) to see how you can supply the record count.

NOTE

Transactions: When you are running a report that access the database (like a report from Entity Framework objects) you must be sure that the database isn't updated while the report is running.

As said in the note above, in database reports FlexCel sends two SQLs to the server for every table, the first to get the count and the second to get the data. If when you run the second SQL the number of records changed, you will see weird things in the report as the number of rows inserted won't be the same as the number of records written to the report. Also for master-detail and even a single table, you must make sure that data doesn't change while it is being fetched. To ensure this, the report should be run inside a "Snapshot" or "Serializable" transaction.

Snapshot transactions are preferred if supported because they don't block other users from modifying the tables while the report runs. Please take a look at the [Performance guide](#) and the [Entity Framework demo](#) for more information about transactions.

Dataset based DataSources

Besides Linq, FlexCel also allows using .NET DataSets as datasources. DataSets are less flexible than LINQ datasources and can use more memory, but on the other hand, they can be faster than Linq in many cases. Also, you might have already your data in DataSets, and in this case nothing will be faster than a direct report from it.

The reason why it uses more memory is that it has to fill the dataset before running the report, loading all the data into memory. The reason it can be faster many times is that it loads all the data just once from the database, and then works from memory, avoiding further access to the database which can be slow.

To use a DataSet in a report, just add the dataset or the tables with:

```
FlexCelReport.AddTable(DataSet);
```

Or any of the similar AddTable overloads.

As with LINQ, if there is a chance that the data can be modified while you are filling the datasets, those datasets must be filled inside a "Snapshot" or "Serializable" transaction. An advantage of datasets here is that those transactions will complete faster. Once the data is loaded in the dataset, you can run the report without worries about other users changing the data, as it is all loaded in memory.

Data Relationships

FlexCel supports Master-Detail reports, where you have a "master" datasource and a "detail" datasource where for every master record you have different detail records. You could have an "employee" table and for every employee, the orders they sold.

For these kinds of reports, you must tell FlexCel which is the master and which is the detail. Also, you must tell FlexCel how those two tables are related; for example, both tables could be related by an "EmployeeId" field that is used to know which orders correspond to which employee.

You can specify those relationships in two ways:

Implicit Relationships

When the individual objects inside a collection of objects contain a public property that is itself a collection of objects, the second collection is implicitly a detail of the first. For example, if you have the class:

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public List<Order> Orders { get; set; }
}
```

Then "Orders" is an implicit detail of "Employee" and when you create a Master-Detail report, FlexCel will output the correct orders for every employee.

Implicit relationships are used in Entity Framework, and are a simple way to provide master-detail relationships if you have your objects organized this way.

Explicit Relationships

Sometimes (as it happens with DataSets) you don't have implicit relationships in your data. You might have two different Lists of objects:

```
List<Employee2> Employees;
List<Order2> Orders;
```

Where the objects are defined as:

```
class Employee2
{
    public int EmployeeId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Order2
{
    public int EmployeeId { get; set; }
    public int OrderId { get; set; }
    public string OrderName { get; set; }
}
```

And related by an Id (in this case EmployeeId). If you were to create a report out of those two lists, you would get the full list of orders for every Employee. You need to tell FlexCel that the "Employees" collection is related to the "Orders" collection by the EmployeeId.

You can add explicit relationships with

`FlexCelReport.AddRelationship(...)`

But you will almost never need to do so. FlexCel automatically loads all existing Data Relationships in datasets, so if the tables on your dataset are related, then you don't need to do anything else.

For other objects different from datasets you will probably want to use implicit relationships. But there might be cases where you need explicit ones. Explicit relationships are more powerful than implicit, in the sense that you can have the same tables related in different ways. For example, The "Orders" collection could have also an explicit relationship with a "Date" table. If you run the report "Orders by Date" then that relationship will be used. If you run the report "Orders by Employee" then the employee relation will be used. If you had implicit relationships, then both "Date" and "Employee" collections would need an "Orders" field, and that could lead to repeated data.

Explicit data relationships can be defined with any objects, even between DataSets and LINQ collections. Whenever FlexCel finds an explicit Data Relationship, it will filter the detail table to show the correct records for the master.

TIP

You might also define the relationships [directly in the template](#) instead of by code. This is useful when using [Direct SQL](#), since the tables are defined directly in the Excel template and you can't set up the relationships in code. Also, you can add custom relationships not related to datasets, when using [virtual datasets](#).

Interface Layer

This is the simplest of the three, as all work is done internally. To set up the interface layer, you only need to tell FlexCel which datasets, tables and user functions it has available from the data layer. For this, you use "SetValue", "SetUserFunction", "AddTable" and "AddConnection" methods on FlexCelReport. Once you have told FlexCel what it can use, just call [FlexCelReport.Run](#) and this is all.

NOTE

It is worth spending some time thinking about what tables, functions and variables from the data layer you want to make available for the final user. Limiting it too much might mean cutting the power of the user to customize his report. And just adding everything might be too inefficient, since you are loading thousands of rows the user is not going to use, and might also have some security issues, as the user might get access to tables he is not expected to. You can also use SQL on the templates to allow maximum flexibility, but then you need to be extra careful about security.

Appendix: Virtual DataSets

This is an advanced topic, targeted to experienced developers with really specific needs.

As explained earlier, the easiest way to provide data to FlexCel is via IQueryable collections or DataSets. Both methods are fast, optimized, with lots of added functionality, like data relationships, filtering or sorting, and if you are experiencing issues with them, you are probably better using them.

But you might want to use your own objects directly, and they might not be stored in any of those containers.

You can do this by writing wrapper objects around your data, implementing the abstract classes `IVirtualDataTable` and `IVirtualDataTableState`. In fact, standard IQueryable collections and Datasets also interoperate with FlexCel by implementing those classes.

NOTE

If you are curious and have FlexCel source code, you can search for **TLinqDataTable**, **TLinqDataTableState**, **TAdoDotNetDataTable** and **TAdoDotNetDataTableState** to see how they are implemented.

We have two classes to implement:

1. On one side we have `IVirtualDataTable`. It is a "stateless" container, much like a dataset. Each virtual dataset corresponds with a table on the data layer that you would add with `FlexCelReport.AddTable`, or create by filtering existing datasets on the config sheet.
2. On the other side we have `IVirtualDataTableState`. Each `VirtualDataTableState` corresponds with a band on the presentation layer, and if 2 bands share the same data source they will have 2 different `VirtualDataTableState` objects associated (but a single shared `VirtualDataTable`).

This is probably easier to visualize with an example. Let's imagine we have the following report:

	A	B	C	D	E
1	Customers By Company				
2					
3	Band: Company				
4	Band: Customer				
5					
6					
7					
8					
9					
10	All Customers				
11	Band: Customer				
12					
13					

And this code:

```
FlexCelReport.AddTable(Customers);  
FlexCelReport.AddTable(Company);  
FlexCelReport.Run();
```

There are two `VirtualDataTables` here (Company and Customers), and three `VirtualDataTableStates` (One for the Company Band, one for the first Customers Band and one for the second Customers Band)

WARNING

Take note that the same `VirtualDataTable` is used by two different `VirtualDataTableStates`, and might be used by other `VirtualDataTableStates` in other threads.

This is why you cannot save any “state” information on the `VirtualDataTable`, and if you write to any private variable inside of it (for example to keep a cache) you should use locks to avoid threading issues.

Always assume that some other class might be reading your data.

`VirtualDataTableState` on the other hand is a simple class that will not be accessed by more than one class at the time, and you can do whatever you want inside it without worries of other threads trying to access it.

Creating a `VirtualDataTable` descendant

The first step into creating our own data access is to create a `VirtualDataTable` descendant and override its methods. You do not need to implement every method to make it work, just the ones that provide the functionality you want to give to your end users.

`VirtualDataTable` Required Methods:

On every `DataTable` you define, you need to implement at least the following methods:

- **GetColumn**, **GetColumnCaption**, **GetColumnName** and **ColumnCount**:

Those methods define the “columns” of your dataset, and the fields you can write on the `<#dataset.field>` tags on the template.

- **CreateState**: It allows FlexCel to create `VirtualDataTableState` instances of this `DataTable` for each band. You will not create `VirtualDataTableState` instances directly on your user code, FlexCel will create them using this method.

`VirtualDataTable` Optional Methods:

Now, depending on the functionality you want to provide to the end user, you might want to implement the following methods:

- **FilterData**: Will return a new `VirtualDataTable` with the filtered data.

You need to implement this method if you want to provide the user with the ability to create new datasets by filtering existing ones on the config sheet. If you do not implement it, any attempt to create a filtered dataset on the config sheet will raise an exception.

Also when FlexCel needs to create a dataset that is a copy of the existing one (for example for sorting it) it will call `FilterData` with `rowFilter` null. So even if you don't implement filtering, it is normally a good idea to at least implement the case for “`rowFilter`” = null and return a clone of the datatable.

Note that this only applies to standard filters. For `<#Distinct()>` or `<#Split>` filters you do not need to implement this.

- **GetDistinct**: Will return a new `VirtualDataTable` with only unique records.

Implement this method if you want to let your user write `<#Distinct()>` filters on the config sheet.

- **LookUp:** Will look for a record on the dataset, and return the corresponding value. This method allows the user to use `<#Lookup()>` tags on their reports.
- **GetRelationWith:** Use this method to return implicit relationships between your data tables. For example the `VirtualDataTable` implementation of Datasets uses this method to return the ADO.NET `DataRelations` between datasets. The Linq implementation returns as related any nested dataset.

Creating a `VirtualDataTableState` descendant:

Again, you do not need to implement every method in this class, and the methods you don't implement will just reduce functionality.

`VirtualDataTableState` Required Methods:

- **RowCount:** Here you will tell FlexCel how many records this dataset has on its current state. Remember that this might not be the total record count. If for example you are on a master-detail relationship, and the master is on record 'A', you need to return the count of records that correspond with 'A'. *Make sure this method is fast, since it is called a lot of times.*
- **GetValue:** Here you will finally tell FlexCel what is the value of the data at a given row and column. You can know the row by reading the "Position" property, and the column is a parameter to the method.

As with `RowCount`, this method should return the records on a specific state, not all the records on the datatable. If you are in a master-detail relationship and only two records of detail correspond the master position, `GetValue(position = 0)` should return the first record, and `GetValue(Position = 1)` should return the second. It doesn't matter if the total number of records is 5000.

`VirtualDataTableState` Optional Methods:

- **MoveMasterRecord:** This method is called each time the master changes its position when you are on a master-detail relationship. You should use it to "filter" the data and cache the records that you will need to return on `RowCount` and `GetValue`. For example, when the master table moves to record "B", you should find out here all the records that apply to "B", and cache them somewhere where `RowCount` and `GetValue` can read them.

You should probably create indexes on the data on the constructor of this class, so `MoveMasterRecord` can be fast finding the information it needs.

You do not need to implement this method if the `VirtualDataTable` is not going to be used on Master-Detail relationships or Split relationships.

- **FilteredRowCount:** This method returns the total count of records for the current state (similar to `RowCount`), but, without considering Split conditions.

If the dataset has 200 records, of which only 20 apply for the current master record, and you have a Split of 5 records, RowCount will return 5 and FilteredRowCount will return 20.

This method is used by the Master Split table to know how much records it should have. In the last example, with a FilteredRowCount of 20 and a split every 5 records, you need 4 master records. You do not need to implement this method if you do not want to provide "Split" functionality.

- **MoveFirst/MoveNext:** Implement these methods if you want to do something in your data when FlexCel moves the active record. For example, the LINQ implementation uses those methods to move the IEnumerator. The dataset implementation does nothing as it doesn't need to track the changes in position.

EOF: Return true if the dataset is at the last record. If you don't implement this method, the default implementation is to check Position == RecordCount. But RecordCount might be expensive to calculate, and in this case, if you explicitly implement this method, it will work faster.

Finally

When you have defined both classes, you need to create instances of your VirtualDataTable, and add them to FlexCel with [FlexCelReport.AddTable](#).

For examples of how the whole process is done, please take a look at the [Virtual DataSet demo](#).

FlexCel Reports Designer Guide

Introduction

This document has two different target audiences, developers using FlexCel and final power users that want to customize their reports. It covers how to design an Excel template, but not the code parts needed to run a report. Read the [Reports Developer Guide](#) for the code part.

Report Elements

There are three concepts you should understand to create or modify a report: **Tags**, **Named ranges** and the **Configuration sheet**. We will be covering all of them in the sections below.

Tags

A tag is text that you write in a cell and that will be replaced by a different value on the generated report. All tags are on the form `<#TagName>` when they don't have any parameters, and `<#TagName(param1;param2...)>` when they have parameters.

NOTE

Notice that the parameter separator is ";" not "," as it is on expressions. This was done so it is simpler to mix formulas in tags.

Tags are case insensitive, so you can write `<#tag>`, `<#TAG>` or `<#Tag>` as you prefer. The convention we usually use is all lowercase, but it is up to you.

You can write multiple tags on the same place, and the result will be the concatenated string. You may also apply different formats to different tags. For example, writing "`<#value1> and <#value2>`" inside a cell will be replaced by something similar to "**1 and 2**"

Tags will be replaced on Cells, Comments, Sheet names, Images, Hyperlinks, AutoShapes, Headers and Footers.

Tag Reference

The complete list of tags you can use and their descriptions is on the [Reports Tag Reference](#).

Evaluating Expressions

Expressions can be used inside `<#If>` and `<#Evaluate>` tags. They behave like standard Excel formulas, and you can use any formula that FlexCel can calculate. But, different from formulas, you can also enter tags inside expressions.

For example, you could write:

```
<#Evaluate(A1 + Min(A2, <#Value>))>
```

NOTE

As you might recall from the previous note, the parameter separator inside tags is ";". **But**, the parameter separator in Expressions is ",", not ";" This is to keep the expressions syntactically compatible with Excel, and also to allow you to mix tag parameters with expression parameters without issues.

The supported list of things you can write inside an expression is detailed on the following table:

Element	Syntax	Description	Example
Tag	<#Tag>	You can enter any tag inside an expression, and it will be evaluated. The tag might contain nested expressions too.	1 + <#Value> ; will return the report variable " Value " plus 1 .
References	A1, \$A1, Sheet1!A2, A1:A2, Sheet1:Sheet2! A1:B20 , etc	Standard Excel cell references. You can use relative and absolute references too.	A1 + A2 will return the sum of what is on cell A1 and A2. As the references are not absolute, when copied down this expression will refer to A3, A4, etc.
Parenthesis	()	Changes operator precedence. Standard operator precedence on expressions is the same as in Excel, that is " 1 + 2 * 3 " = 1 + (2 * 3) = 7 and not (1 + 2) * 3 = 9	(1 + 2) * 3 ^2 will be evaluated different than 1 + 2 * 3 ^2
Arithmetic Operators	+, -, *, /, %, ^ (power)	Standard arithmetic operators.	1 + 2 * 3 ^2 will evaluate to 19. 5% will evaluate to 0.05
Equality Operators	<, >, =, >=, <=, <>	Standard equality operators.	1 >= 2 will evaluate to false.
Functions	function(parameters)	You can use any formula function that FlexCel can recalculate inside an expression. For a list of supported functions, take a look at the list of supported Excel functions	

Named Ranges

While Tags allow you to replace complex expressions inside a sheet, with them alone we can only create "Fill in the blanks" type reports. That is, reports that are static, like a form, and where cells with tags will be replaced with their corresponding values.

Now we are going to introduce the concept of "**Band**". A Band is just a range of cells that is repeated for each record of a table. Imagine that you define a Band on the cell range A2:C2, and associate it with the table "Customer". Then, on cells A2:C2 you will have the first customer, on cells A3:C3 the second and so on. All cells that were previously on A3:C3 will be moved down after the last record of the dataset.

If table customer has six registers, and you have a template as follows:

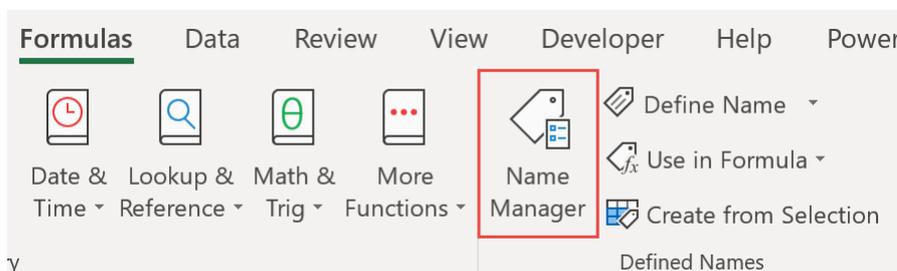
	A	B	C	D
1	Customer Name	Customer Id	Customer Address	
2	<#Customer.Name>	<#Customer.ID>	<#Customer.Address>	
3	This is some text below the band			
4				

After running the report you will get something like this:

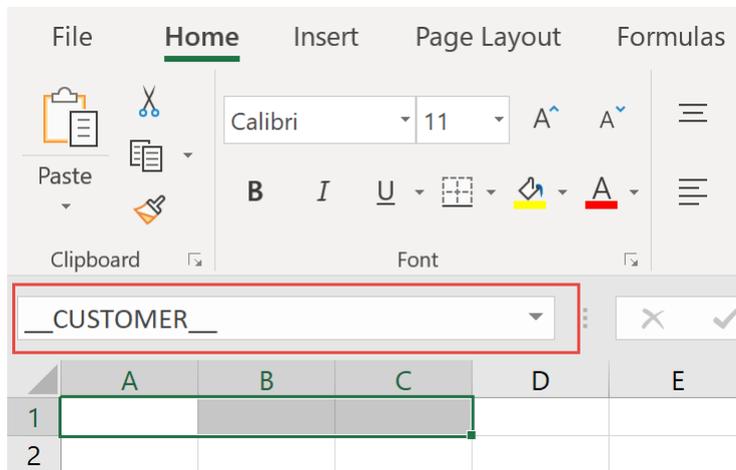
	A	B	C	D
1	Customer Name	Customer Id	Customer Address	
2	Customer 1		1 Address 1	
3	Customer 2		2 Address 2	
4	Customer 3		3 Address 3	
5	Customer 4		4 Address 4	
6	Customer 5		5 Address 5	
7	Customer 6		6 Address 6	
8	This is some text below the band			
9				

On FlexCel we use **Named Ranges** to indicate the bands. If you are not used to Excel Named Ranges, take some time to familiarize with them, as they are one of the things that can be a little confusing when starting with FlexCel. Different from tags, that you can immediately see when you open a workbook, named ranges are a little more hidden.

To create a Band on A1:C1, we would go to the "Formulas" tab, then choose "Name Manager":



Once there, we can define a Band **Customer** on cells **A1:C1**. And once the name is defined, we can easily see it on the Names combo:



Note the “_” at the beginning and at the end of the range name. We use this to indicate FlexCel that this is a horizontal range that inserts full rows down.

The rest of the name (**Customer**) should be the name of an existing data source (table, array of data, etc), or a custom table defined on the config sheet.

Range Types

You can define four different kinds of bands on FlexCel:

- “**__**” **Range**: This range moves down and inserts full rows. For example, to define a band that inserts full rows down for each record of the dataset “Customer” you would use a name `__Customer__`
- “**_**” **Range**: This range is similar to “**__**” but cells outside of the range won't move down. To define a band that inserts only a range of cells down for each record of the dataset “Customer” you would use a name `_Customer_`
- “**II**” **Range**: This range moves to the right and inserts full columns. Note that the first character is the letter i, not a pipe (|). To define a band that inserts full columns for each record of the dataset “Customer” you would use a name `II_Customer_II`
- “**I**” **Range**: This range is similar to “**II**” but cells outside of the range won't move right. To define a band that inserts cells to the right for each record of the dataset “Customer” you would use a name `I_Customer_I`

A “**__**” range is the same as a “**_**” range defined on the full rows, and the same is valid for “**II**” and “**I**” ranges for columns.

On the following example, if you name A1:D7 as “`_Customer_`” Cell E8 won't move when inserting down. If you name it as “`__Customer__`” Cell E8 will move to the last inserted cell, because a “`__Customer__`” range is equivalent to a “`_Customer_`” range on A1:XFD7.

	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					
8					A Value
9					

Master-detail

Named ranges can be placed inside others, and a master-detail relationship will be automatically created. For example, if you define a range “__Customer__” and inside it a range “__Orders__” and there is a relationship created on the application between “Customer” and “Order” tables, it will automatically group your orders by customer.

On the following example, the yellow cells are the range “__Customer__” and the blue ones are the range “__Orders__”

	A	B	C	D
1				
2	Customer	<#Customer.Name>		
3		Orders		
4		<#Order.Id>		
5				
6				

After running this report, you will get something similar to:

	A	B	C	D
1				
2	Customer	Customer1		
3		Orders		
4		Order 1 for Customer 1		
5		Order 2 for Customer 1		
6		Order 3 for Customer 1		
7				
8	Customer	Customer2		
9		Orders		
10		Order 1 for Customer 2		
11		Order 2 for Customer 2		
12				
13	Customer	Customer3		
14		Orders		
15		Order 1 for Customer 3		
16		Order 2 for Customer 3		
17		Order 3 for Customer 3		
18		Order 3 for Customer 3		
19				

As you can see, Orders are filtered for each customer, based on the [Data Relationships](#) defined on the application, and on the nesting on the ranges. In general, any range that is inside another is filtered by all of its parents. You can have as many levels of master-detail relationships as you wish, and each master band filters all of its children.

For example, if we wanted to group the customers by country we could define a `__Countries__` named range on A1:F6, and it would automatically filter the data on its child and grandchild. (Customer and Order)

Multiple-sheet Master-detail

There is a special Table that if present filters all the others on the sheet, acting as a parent of all the named ranges on the sheet. This is the table that you define on the name of the sheet, when doing a multiple sheet report.

You can see the [Multiple sheet report](#) demo for an example: On it every table on each sheet is filtered by category.

Ignoring Ranges

Sometimes, you might want to ignore processing in a range of cells or sheets. So for example if you have a sheet with `<#tags>` but which aren't real FlexCel tags, you might want to exclude that sheet from the report.

- To exclude full sheets, start the name of the sheet with a dot ("."): The dot will be removed in the final report. So for example if you name a sheet ".Totals", that sheet won't be used in the report, just left "as-is". In the final result, the sheet will be renamed to "Totals".
- To exclude just some cells from a sheet, you can create a band and name it starting with a dot ".". Ranges starting with a dot (".", "_.", "I_" and "II_") are ignored by FlexCel, so you could define a name ".ignore." and it won't be processed.

You can see an example of ignoring ranges of cells in the [Pivot Tables](#) demo.

Bidirectional Bands

As a general rule, bands in FlexCel can't intersect. You can have separated bands:

	A	B	C	D	E
1					
2		Band 1			
3					
4		Band 2			
5					

And they will each expand separately. You can also have one band completely inside another:

	A	B	C	D	E
1					
2		Band 1			
3		Band 2			
4		Band 1			
5		Band 1			

And the inside band will be a detail of the outside band. But if the bands intersect:

	A	B	C	D	E
1					
2		Band 1			
3			Band 2		
4		Band 1			
5		Band 2			
6		Band 2			

It is not really possible to do a report with this data. There is no master and no detail here, and if we run those two bands separately, one band would overwrite the results of the other. It is not possible to know if cell C3 should have data from Band 1 or Band 2.

So in general, FlexCel will raise an error if you try to do a report with bands that intersect: This is most likely a mistake in the template.

But there is one special case where FlexCel allows you to have two bands that intersect: If the bands form a cross, and the vertical leg of the cross is a column range, while the horizontal leg is a row range:

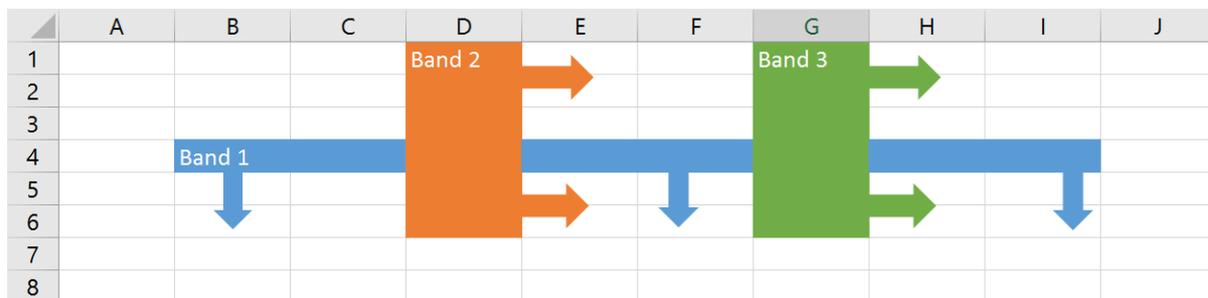
	A	B	C	D	E	F	G
1				Band 2			
2				Band 2			
3				Band 2			
4		Band 1					
5		Band 1					
6		Band 1					
7		Band 1					
8		Band 1					

This will create a bidirectional band, that grows both to the right and to the bottom at the same time. The conditions for creating a bidirectional band are:

1. One band must grow horizontally and the other vertically.
2. The bands must form a cross. This means that the top of Band 2 must be smaller or equal than the top of Band 1, the bottom of Band 2 must be bigger or equal than the bottom of Band 1, the left of Band 1 must be smaller or equal than the left of Band 2, and the right of Band 1 must be bigger or equal than the right of Band 2. You will normally make Band 1 a full row range and Band 2 a full column range.

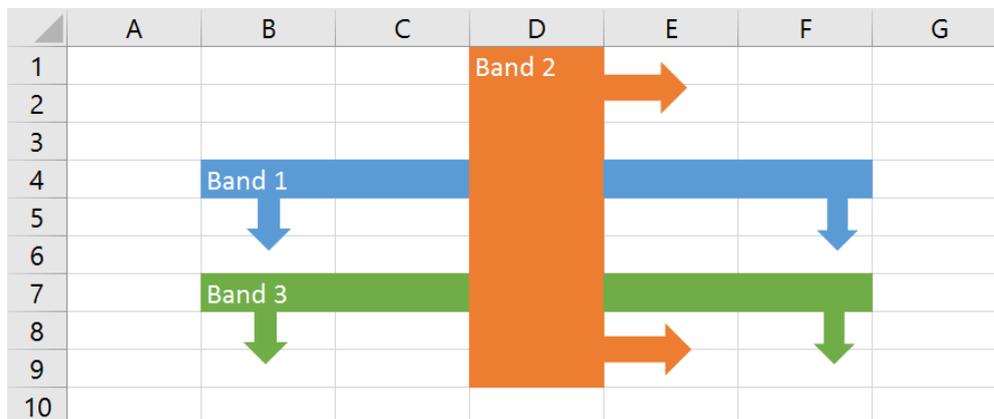
If both conditions are met, FlexCel will create a master-detail report where Band 1 is the master and Band 2 the detail. For bidirectional bands, **FlexCel will always make the horizontal band the master and the vertical band the detail.**

As the vertical band is the detail, you can have multiple vertical bands for the same horizontal band master. That is, you can have something like this:



And both Band 2 and Band 3 will be details of the master Band 1.

What you can't currently have is two row bands intersecting the same column band:



In this case, as FlexCel will always make the vertical bands the detail, Band 2 should be the detail of both Band 1 and Band 3. And while a master band can have many details, a detail band can have only one master.

A workaround if you really need such a thing would be to use `<#Includes>`. You could have two subreports: one with Band2 and Band1, and the other with Band3 and Band2. Then in your master report, you `<#Include>` both subreports one below the other. The effect will be as if you had run multiple horizontal bands with Band2, but there won't be multiple masters.

For more information on Bidirectional Reports, you can look at the [Bidirectional Reports](#) example.

X ranges

One issue that might appear when defining named ranges is how formulas on other ranges change when inserting the new cells.

Let's imagine we want to make a simple report on a list of items and their prices.

So, we create a new template, define a title row, and insert a named range on A2:B2 to propagate the data. But we also want to know all the total cost of all items, so we add a formula on B3:

	A	B	C
1	Item	Price	
2	<#Item.Name>	<#Item.Price>	
3		=Sum(B2:B2)	
4			

When you run this report, rows will be inserted between row 2 and 3, but the formula Sum(B2:B2) won't change. Nothing has been inserted between B2 and B2, so the sum range will remain constant.

So, we need to have a Sum range that can expand. We will define:

	A	B	C	D
1	Item	Price		
2	<#Item.Name>	<#Item.Price>		
3				
4		=SUM(B2:B3)		
5		SUM(number1, [number2], ...)		
6				

Now, when rows are inserted between row 2 and 3, the formula will be updated to reflect the new rows.

In this particular case, this solution might be enough. Just leave an empty row after the range so formulas expand, and then you can hide the extra row or just leave it there.

But, if we were for example creating a chart, this extra row will be on the chart too. If you are an old FlexCel VCL user, you know about the "...delete row..." thing just to avoid those cases.

Well, delete row does not work anymore on FlexCel 5.0 and up, because the row would be deleted before the range is expanded, and the formula will then point to B2:B2 again.

This is why we introduced the **"X" ranges**. X ranges are normal named ranges with an "X" at the end. On this case, instead of `"_Item_"`, we would call the range `"_Item_X"`. It will behave exactly the same as a normal range, but once it is expanded it will erase the row immediately after the range (or column if it is a column range). So if we try the last example with `"_Item_X"`, row 5 on the last screenshot will be deleted, and the formula would be `"=SUM(B2:B4)"`. Just what we were looking for. See the [Charts](#) demo for more information on using X ranges.

IMPORTANT

Don't include the row you want to delete inside the range you are creating!

The row (or column) deleted will be the one **after** the range, not the last row in the range. So if you define a range "`__MyBand_X`" in rows 2 and 3:

	A	B	C	D
1				
2	<#myband.Field>			
3				
4	← This row will be deleted			
5				
6				

FlexCel will repeat the 2-row range for each record, and then delete the row after the range, as seen below:

	A	B	C	
1				
2	Record1			
3				
4	Record2			
5				
6	Record3			
7				
8	← This is the row deleted			
9				

This is not likely what you want. To have one row per record, the `__MyBand_X` name must have one row only, and then the row **after** the band will be deleted.

Fixed Bands

By default, FlexCel will always insert cells when expanding ranges, and this is what you would normally want. If you have a template:

```
A1:Title  
  
A2:<#data>  
  
A3:Footer
```

You would expect that the generated report will have the Footer for example on cell A33 (if we had 30 data records), but not on A3.

But there is a situation where this is not what you expect, and this is on Fixed Form reports. Let's imagine that you want to fill out a form with FlexCel. Most fields will be just simple expressions, not related to datasets, but we might have a table too:

	A	B	C	D	E	F	G	H	
1									
2			<#Employee.FirstName> <#Employee.LastName> Data						
3			<i>An Example on how to use bands inside fixed forms</i>						
4									
5									
6									
7	Name:	<#Employee.FirstName> <#Employee.LastName>							
8	Title:	<#Employee.Title>							
9									
10		Top 10 orders from Employee							
11	Order	Customer				Shipped Date	Freight		
12	1	<#TopOrders.ShipName>				<#TopOrders.Sl	<#TopOrders.Freight>		
13	2								
14	3								
15	4								
16	5								
17	6								
18	7								
19	8								
20	9								
21	10								
22									
23	This is a Fixed form report and this line is expected to always print					Total	0.00		
24	on this particular place								
25	Note that we removed Autofit from rows so they do not change with the cell contents.								
26	<hr/>								
27									
28									
29									
30									
31									
32									

Here, no matter if the dataset has 1 record, two or 10 (it should not have more than 10) you want the "Total" line to be at row 23. You cannot do this with normal ranges, since you would be inserting rows. For this you can define a "__TopOrders__**FIXED**" named range, which will not insert any records.

FlexCel will treat any range that ends with the word "FIXED" as a fixed range, as long as the characters before the range mean the end of a range. That is, II_Range_IIFIXED will be a fixed range, since "_II" is the end of a vertical range. But __OrdersFixed__ is not a fixed range.

NOTE

The word "FIXED", as most stuff in Excel, is case insensitive. Both __Range__FIXED and __Range__fixed are ok. But in our docs and own templates, we use UPPERCASE modifiers so they stand out and are easier to spot.

WARNING

You can't use FIXED bands in master-detail reports. The problem is that master-detail works by inserting the rows of the details between rows of the master. As FIXED bands don't insert rows, you will end up with the master overwriting cells of the details or vice-versa.

To do a fixed master-detail report, you can just run the report with normal (not FIXED) names in a separate subreport, and then <#include> that subreport into the main report using a FIXED include so it doesn't insert rows or columns.

See the [Fixed Forms With Datasets](#) demo for more information.

Fixed N Bands

Sometimes, you might want the records to overwrite the first n cells (like in a "fixed" band), but after those rows are overwritten, insert the rest of the records (like in a normal report).

You can get this by using a "FixedN" range, where "N" is the number of rows or columns you want fixed. For example "`__data__FIXED2`" will overwrite the first 2 rows, and insert (data.RecordCount – 2) rows for the rest.

You can see an example of FixedN bands at the [Balanced Columns](#) demo.

Alias Bands

In some rare cases, you might want to use the same dataset twice in the same sheet. One way to do it is to define an alias in the config sheet, so you have, for example:

- DataSet1
- NewDataSet1 -> Alias to DataSet1

And then, in the report, you define a **DataSet1** and a **NewDataSet1** range.

But in some cases, like, for example, when you are creating a report from a list of objects and there are nested objects, you can't rename those nested objects. If a **Customer** object has a **Orders** nested object, then you can't rename **Orders** in the config sheet. You would need to define two **Orders** names, but only one is allowed by Excel.

To solve this, you can use the **..Alias..** postfix. Just write the word **..ALIAS..** (case insensitive) after the name, and then write whatever you want.

You could define the names as **Orders..ALIAS..First** and **Orders..ALIAS..Second**.

Alias are always defined at the end, and everything after the ..ALIAS.. word is ignored. So if you want to define say a fixed name, it would be defined as **Orders__FIXED..ALIAS..MyOrders**, and not as **__Orders..ALIAS..MyOrdersFIXED**.

NOTE

Alias just allow you to define different named ranges in Excel pointing to the same dataset by writing arbitrary text after the name. They don't change the name of the dataset, and in the example above, you would still write `<#Orders.Name>` in the template, not `<#First.Name>`.

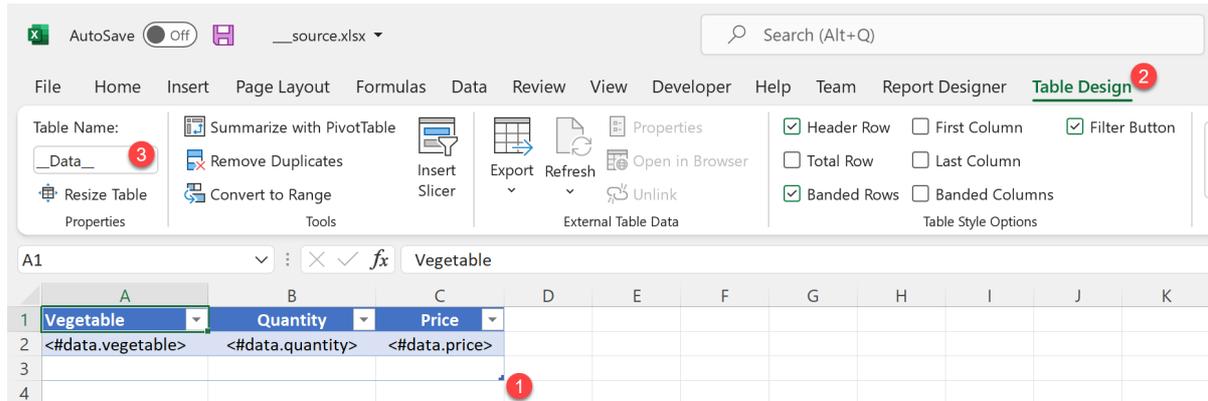
NOTE

When this feature was introduced in FlexCel 7.1, it used a different syntax: You would write something like "`__MyOrders__Alias_MyAlias`". The problem with this syntax is that you couldn't define something like `__MyOrders__XAlias_MyAlias`, since the "X" wouldn't be separated from the "Alias" definition.

To make it simpler, we changed the alias to use 2 dots (..) before and after the name. The old syntax still works for simple cases, but it is deprecated. We encourage you to use the new syntax for new developments.

Excel-Table Bands

Besides using names for the bands, FlexCel also allows the use of Excel tables as bands. To use an Excel table as a band, you need to follow the steps in this image:



1. Insert a table with two data rows. **You need an extra empty row after the row with the data** so when the report runs the table will grow with the inserted records. Bands defined from tables behave like [X Ranges](#) and they erase the empty row after the report is completed.
2. With the cursor inside a cell of the table, go to the "Design" tab to edit the name of the table.
3. Name the table "_Data_" as you would with any other "_" name. You can also name it as "_Data_" and FlexCel will not copy the full rows, just as it does with "_Names_". "_I_" and "_II_" names are allowed too but make no sense in tables, since tables are always organized to have the data in rows and the columns to hold different fields. You might also name the table "_Data_X" but as explained in point 1. table datasources are always "X" so there is no need to write the extra "X" at the end.

Excel-Table bands are exactly the same as an "X" named-range band defined over the data part of the table, so everything you can do with name bands you can do also with table bands. You can for example store a table inside a name to create a master-detail report, or anything you can do with normal name bands.

Customizing Excel-Table Bands

There are two configuration options that apply to Excel-Table bands:

1. To make FlexCel completely ignore table bands, you can set the property [FlexCelReport.UseExcelTablesAsBands](#) to false. If you turn off this property you won't lose any functionality, but you will have to design the report with `__Named_Ranges__` instead.
2. Table bands will normally rename themselves by removing the "_" part of the name. If your template table is named `__Customer__` the table in the result file will be named "Customer". To avoid this behavior you can change the property [FlexCelReport.RenameExcelTablesUsedAsBands](#) to false

There is a working example on reports using tables at the demo [Tables As Datasources](#)

Balanced Columns

Sometimes you might want to create a report that once generated looks like the following:

The diagram shows a report layout on a grid. At the top is a light green header bar labeled "Header". Below it are two columns: a yellow block labeled "Block 1" on the left and a blue block labeled "Block 2" on the right. At the bottom is an orange footer bar labeled "Footer".

But if you try to do this by creating a `_Block1_` and `_Block2_` parallel ranges, you will actually end up with something like this:

The diagram shows a report layout on a grid. At the top is a light green header bar labeled "Header". Below it are two columns: a yellow block labeled "Block 1" on the left and a blue block labeled "Block 2" on the right. The footer bar, labeled "Footer", is orange and is split into three segments: one on the left, one in the middle between the two blocks, and one on the right. This indicates that the footer is not aligned across the entire width of the report.

Because when `_Block1_` and `_Block2_` grow down, the cells are inserted only in the columns that are used by those ranges.

But there is a case where FlexCel doesn't work this way, and that is in master-detail. When you are doing a detail, cells are inserted in all the columns so they end at the same place, and the next master record won't be broken. So the solution in this case is to create a dummy "master" dataset with a single row, and use it as a master for both `_Block1_` and `_Block2_` ranges.

The last cell in every column of the "master" dataset will be copied down so all columns inside the master insert the same number of cells.

You can use the "ROWS" function in the config sheet to create a single row datasource.

Please take a look at the [Balanced Columns](#) demo for more information on how this is done.

Intelligent Page Breaks

Other kinds of ranges you might want to create are "KeepTogether" ranges.

FlexCel will try to keep rows or columns on those ranges together when printing by inserting page breaks at the needed places.

To create a "Row" KeepTogether range, you need to name it:

KEEPROWS_<Level>_<Whatever>

Where **<Level>** is the level of "keep together" of the group, and **"Whatever"** is anything, you can use it to have more than one "keep together" range in the same sheet.

For example, you might have the ranges:

KeepRows_1_customers

and

KeepRows_1_orders

in the same sheet, to tell FlexCel to group the rows in both ranges together when printing. You might also have different levels of "Keep together", and you will normally use higher levels for details in master-detail reports.

Once you have created the ranges, you need to write an `<#auto page breaks>` tag somewhere in the sheet, and FlexCel will add page breaks when it ends the report trying to keep those ranges together. You can customize the `<#auto page breaks>` to influence the way page breaks are created.

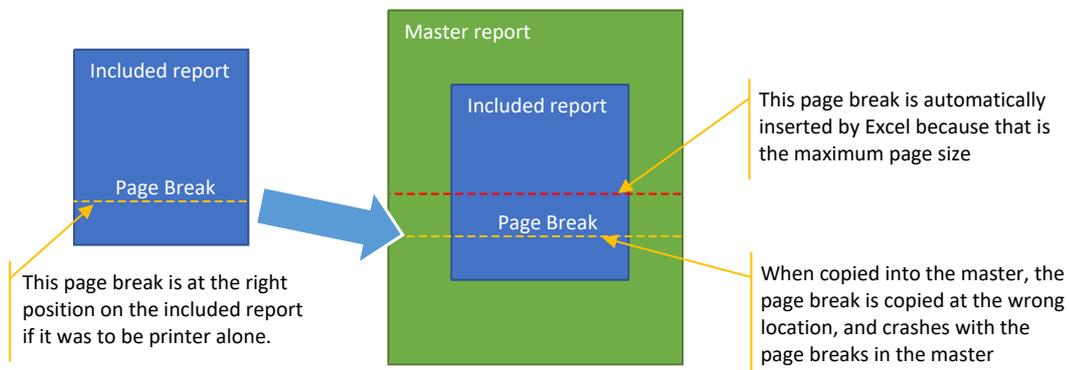
We are not covering in detail Intelligent page breaks here, since they are described in the [API guide](#), and it makes no sense to repeat that information here. We only cover here what is different for reports, and this is the "KeepTogether" ranges.

Make sure you read the section [Preparing for Printing](#) in the API guide, since most of the concepts apply also to reports. And of course take a look at the [Intelligent Page Breaks in Reports](#) demo.

Intelligent Page Breaks in Included Reports

You can use intelligent page breaks inside an `<#included>` report, but you must be aware of some differences.

The most important difference is that the final pagination must be done in the master report, not in the included one. Let's look at an example:



In this example, we can have "KeepRows_" and "KeepColumns_" ranges in the included report, but we should not create the actual page breaks in it. If we did it, those page breaks would be in the wrong position when inserted in the master. (unless the report is included at cell A1). As you can see in the drawing, the included report is copied into the master at a lower location. So if the report goes down, the page breaks will go down too, ending up in the wrong places. Excel needs to insert its own page break (marked with a red dotted line) in order to have the master page no bigger than the paper size, and the old page break will create a small page that we don't want.

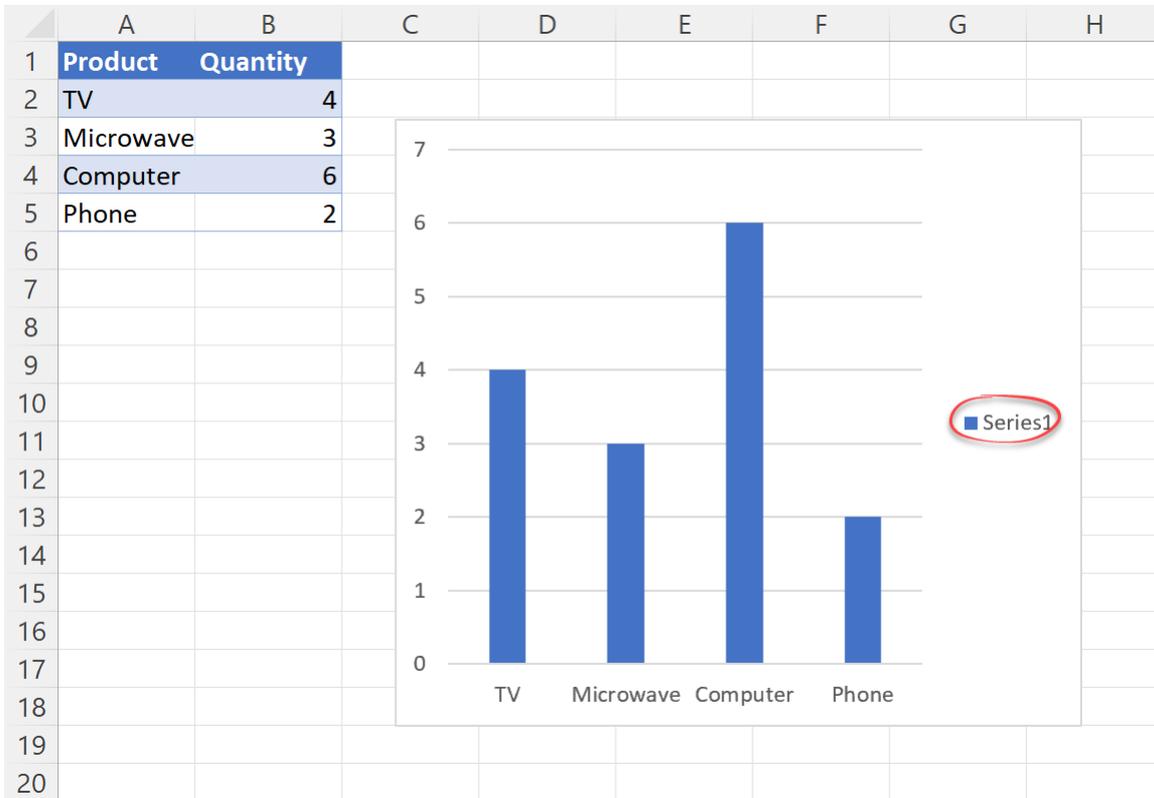
So, in order to have intelligent page breaks in the included report, you must follow the next simple rules:

- Create the "KeepRows_" and "KeepColumns_" ranges in the included report, as you would normally do it.
- **Do not write an <#auto page breaks> tag in the included report.** This will ensure FlexCel does not add the page breaks in the included report before copying it to the master.
- Make sure you include full rows (by using "_" or "II_" as parameter in the include tag). This way, the KeepRows and KeepColumns ranges will be copied to the master.
- Write an <auto page breaks> tag into the master. This way, when the master report is finished, FlexCel will paginate the master, keeping together the rows you marked in the included report. **Pagination must be done on the master, never in the included report.**

Creating charts with dynamic series

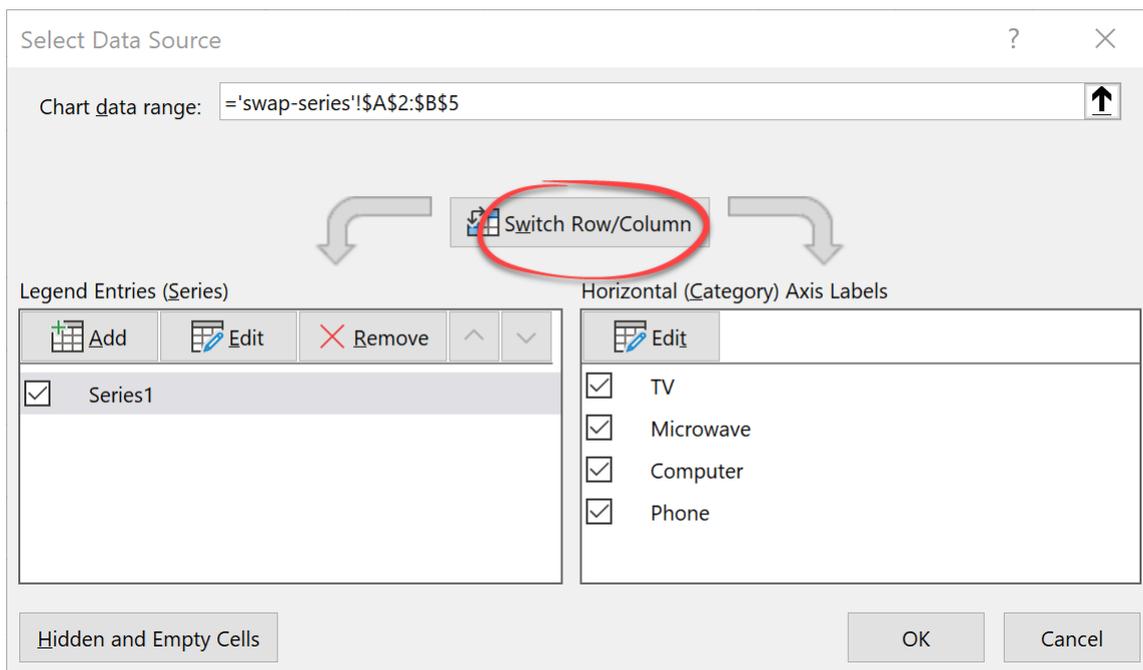
Series and data points

When dealing with charts, a rule of thumb is that you want as few series as possible. That's what Excel does when you select a range of cells and create a chart from it. So, for example, if we have the following table with four rows and two columns, Excel will use the columns to define the series.

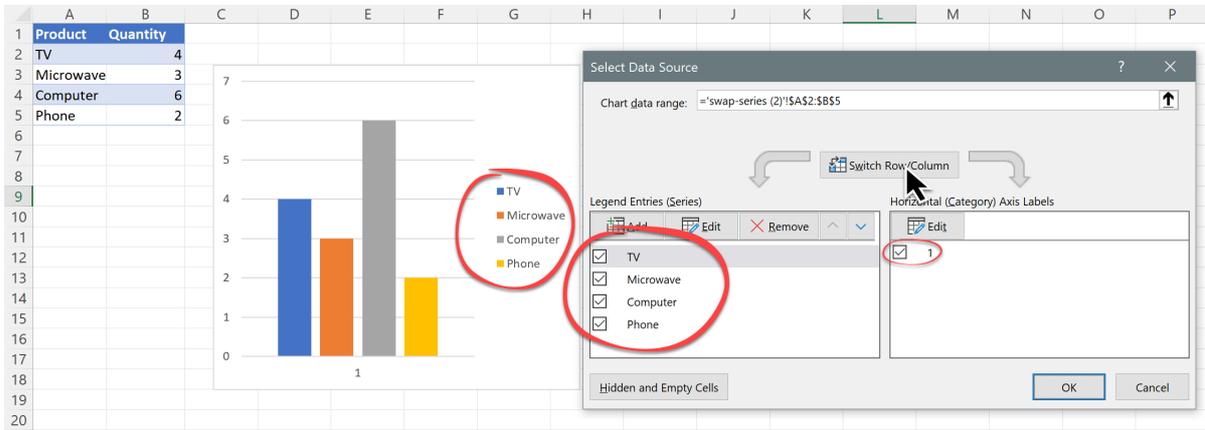


As you can see in the image, only one series was created, which goes from B2 to B5. If we added 100 extra rows to this table, the chart would still be a single series going from B2 to B105.

That makes sense, and you usually want the smaller dimension to have the series. So, for example, if the report had more columns than rows, you would like to have the series by rows, not by columns. But that's not always the case, and Excel provides a handy button: "Switch Row/Column" that you can use to decide if you want the series using the rows or the columns.



If you press that button, you will get the following:

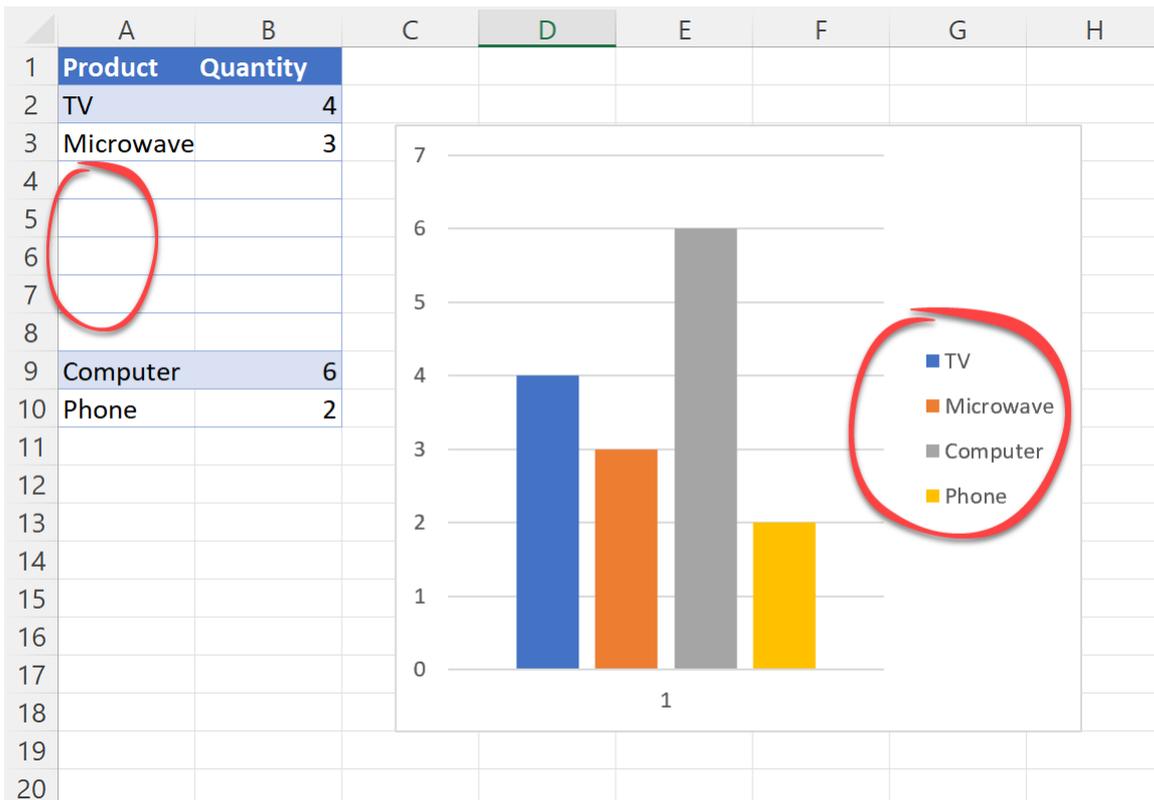


Now you have four series and one data point instead of one series and four data points.

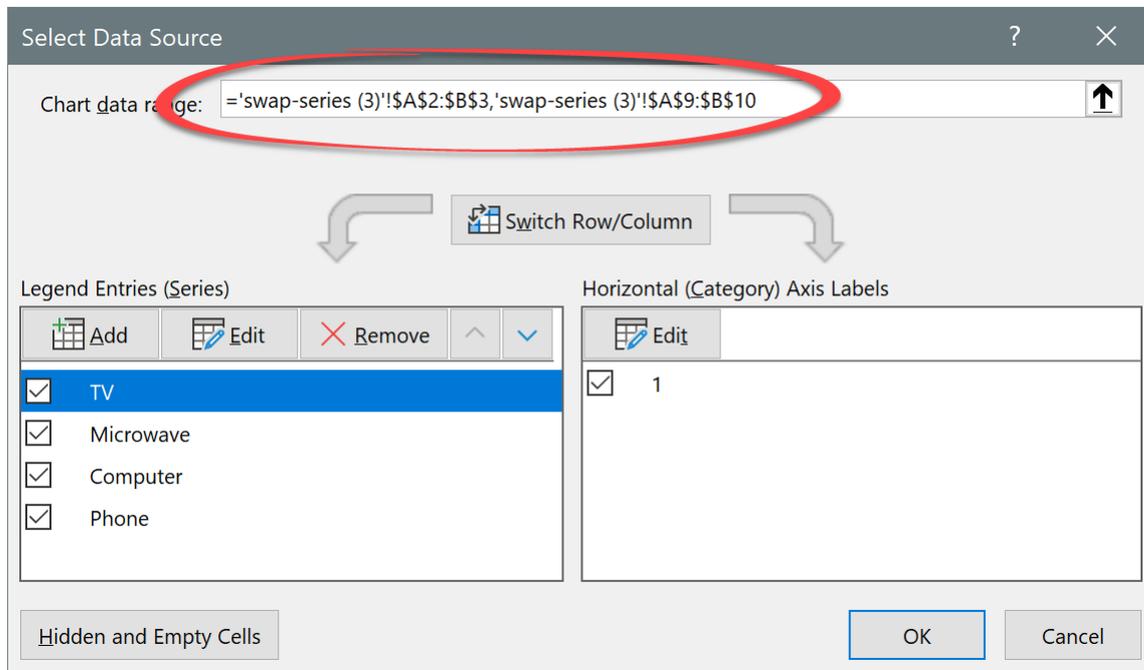
Series in reports

When doing a report that has charts, the typical case works just fine. Your series are in columns, like in the first image, and data points grow down as you insert records. So, for example, if the chart in your template had two series with two data points, and you insert 100 records, you will end up with two series and 102 data points.

But what if you wanted the series in rows? You should end up with 100 series and 2 data points. But that's not how inserting rows work in Excel. When you insert rows in the chart, and you have each series in a row, you will end up with a series at the first row, then a lot of ignored rows, then series at the end:



Here, you can see that even as we inserted rows between "Microwave" and "Computer", we still have only four series. And we can check it out by looking at the ranges:



As you can see at the top, the single range was split in two: One taking \$A\$2:\$B\$3 and the other taking A\$9:\$B\$10. So we still have four series, and this is not what we wanted. We wanted to have nine series now.

As reports work by inserting rows, you will have the same problem here if you try to do one. So how do we create a report where each inserted row is a new series?

The <#swap series> tag

The "swap series" tag behaves a little like the "Switch Row/Column" button at the top of this section. It will run after the full report is completed, and if your chart has the series in columns, it will swap them to be in rows. If they are in rows, they will switch to columns.

IMPORTANT

Usually, you don't want to use the "swap series" tag. We provide them for the **exceptional cases** where you want the series in rows instead of in columns as it would be expected.

Also, note that the maximum number of series in a chart is 255, so you can't do a chart with many rows and one series per row.

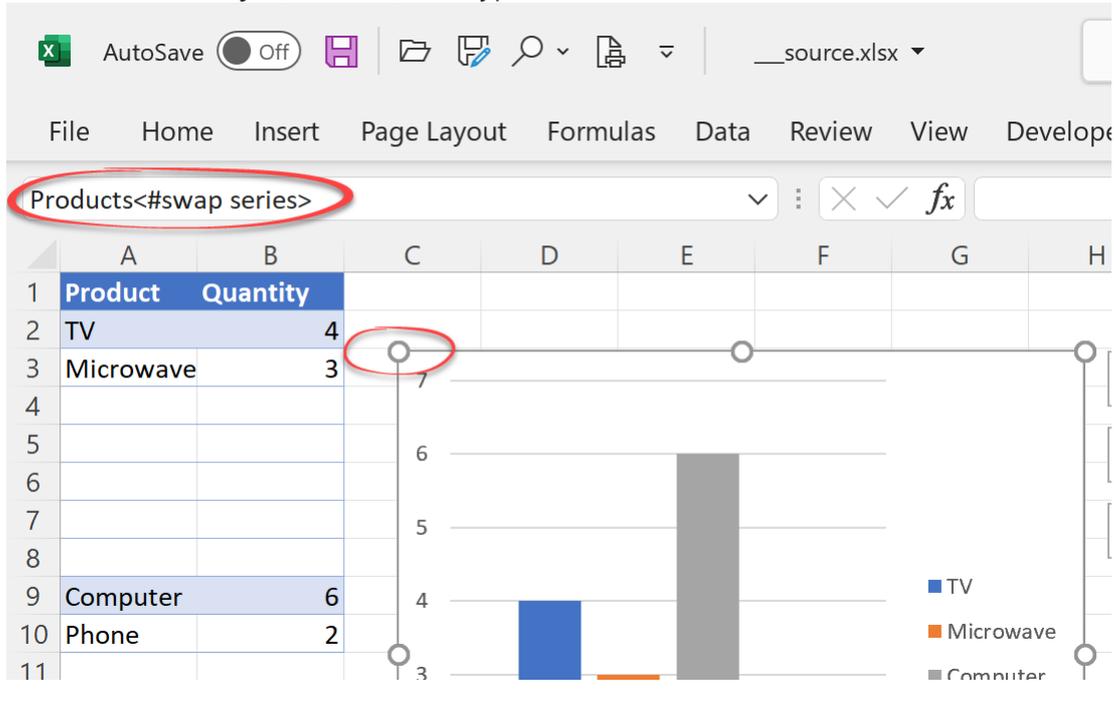
We provide this long explanation for a tag you will likely never use just because the topic is complex, and we want it to be well documented in case you need it. But don't assume that because the functionality exists, you have to use it!

To create a chart that has one series per row, you need to follow the steps:

Create a chart that has one series per column. If it is a chart sheet, name the sheet with some name, and append <#swap series> string at the end of the sheet name. If it is an embedded chart, name the chart as you wish, and append <#swap series> at the end of the chart name.

NOTE

To name a chart object, select it, then type its name in the "Name box":



When the report runs, it will create a "standard" chart with series in columns, but after it ends, the tag will recreate the series using one row per series. If this was a range expanding to the right instead of down, everything applies but reversed: You would create a standard chart with series in rows, and then "swap series" would swap the series to be in columns.

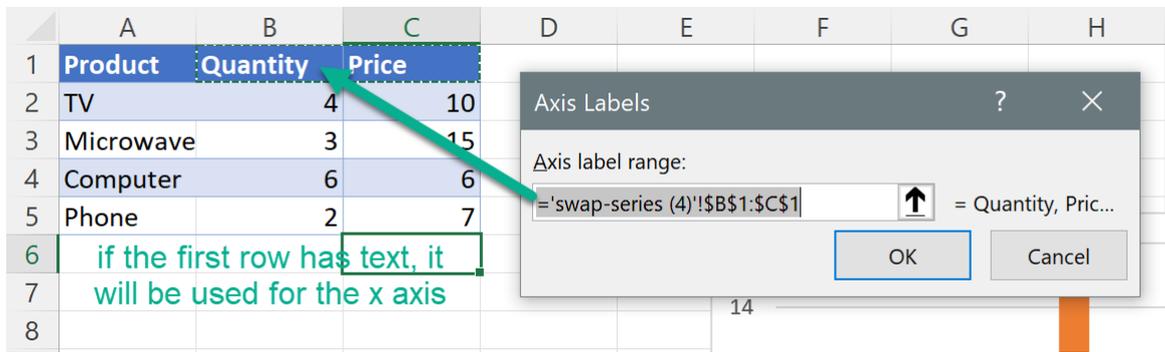
See the example [Charts With Dynamic Series](#) to see how it works.

A deeper explanation of <#Swap series>

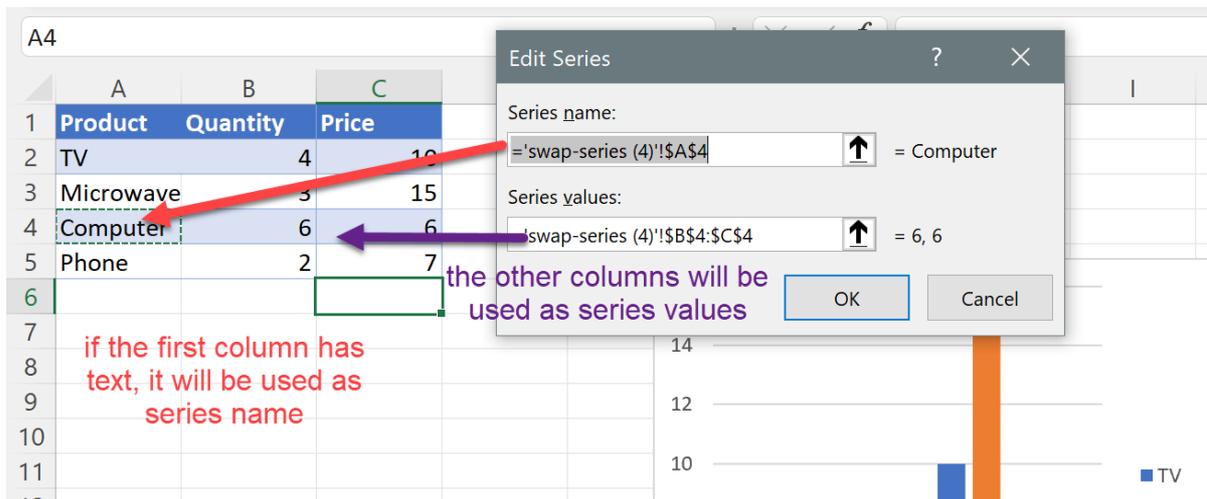
The <#Swap series> concept is simple: If you have your series in columns (which you need so the ranges expand down when inserting rows), it will switch them to rows. But in this section, we will dig deeper into what that really means.

When you select a range and create a chart, Excel decides where to apply the range parts depending on your data. If the range has more rows than columns, it will put the series in the columns. Otherwise, it will use the rows.

For a chart with series in rows, if the first row has strings, but the other rows have numbers, it will use that first row for the x-axis. The other rows will have one series each. If the chart is X-Y scatter, it will use the first row for the x-axis, and each of the remaining rows will hold a series.

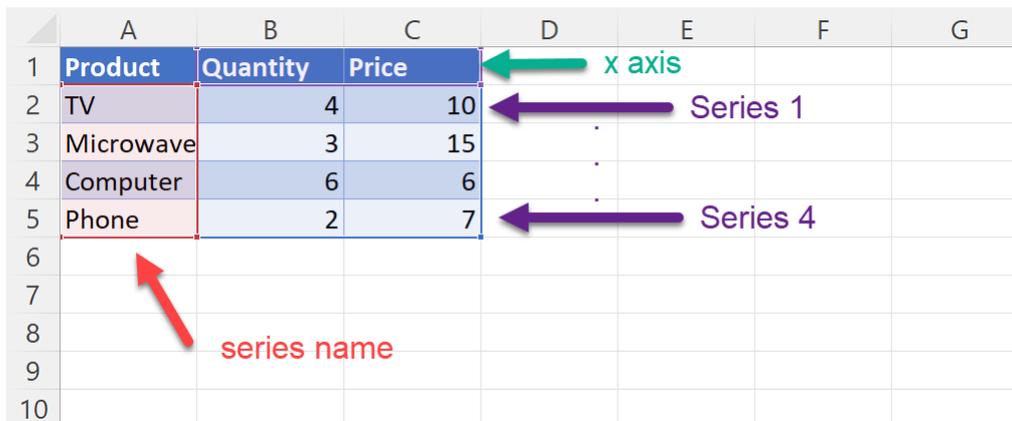


Again for a chart with series in rows, if the first column has strings, it will use it for the series name.



Everything applies mirrored for a chart with series in columns.

Putting it all together, you have the following:



Note that whether you have strings or numbers in the first row and column determines if there is a series name or an x-axis. If all the cells had numbers, the series name and x-axis would be empty, and the full range would be used for the series values.

The exact algorithm used by Excel can vary and could change in new Excel releases. So the algorithm FlexCel uses in `<#swap series>` is simpler: **Swap series won't try to guess how to use the range from the data at all.** Instead, it will use whatever you used in the original chart.

So, if you want to use the chart range only to fill the series, leave the x-axis and series name empty in the original chart in the template. FlexCel will then use the full range for the data points. If you set any value in the x-axis, FlexCel will use the first row for the x-axis. If you set the series name in the template, FlexCel will use the first column for the series name. FlexCel will never try to guess what to do depending on the data types of the first row or column.

NOTE

Remember that when setting the data for a <#Swap series> chart, you will set the series in columns, so if there is an x-axis, it will be the first column. But when <#swap series> changes the series to be in rows, it will use the first row for the x-axis.

Don't worry about the switch between rows and columns: If you want an x-axis, you need to set a column for the x-axis in the template, even if it will be a row in the final result. The same applies to the series name but reversing rows and columns.

Configuration Sheet

The configuration sheet is a repository where you can configure different things on your report, define commonly used expressions and so on. It is not required, you can create a report without a configuration sheet, but it is recommended that you have one except for very simple reports.

Once the report is run, the configuration sheet will be deleted.

NOTE

If there are any macros assigned to the configuration sheet, FlexCel will be unable to delete it and will just clear and "Very hide" the sheet. (Very hidden sheets are similar to hidden sheets but the user needs to write a macro to see them, they are not listed when you select to unhide a sheet from Excel). If you get a macro by mistake on the config sheet and you are unable to delete it in Excel, use the "MacroCleaner" tool included in the "Tools" folder.

The layout of the config sheet is free as long as you keep the positions, you can write the captions you want or change any format you need. Just one advice: keep it simple. The config sheet will be deleted from the final report, so the final user won't see it. And deleting a sheet with comments, images formulas and so on is slower than deleting a simple sheet.

An example of a config sheet is shown here:

Configuration Sheet							
Data							
Table Name	Source Name	Filter	Sort Fields (",", Separated), ASC or DESC	Format Name	Format Def	Name	Expression
OrderDetails	Order Details	ProductId=7		blue	test	LowCost	<#include(LowCostOrd
				red	test	NormalCost	<#include(NormalCost
						HighCost	<#include(HighCostOr
						TotalPrice	<#evaluate(<#OrderDe
						Order1	<#if(<#TotalPrice> < 50
						Order	<#if(<#TotalPrice> >= 1

We can note that:

1. For a sheet to be the configuration sheet it must be named "**<#Config>**" or any expression that evaluates to the string "**<#Config>**". You could conditionally make a sheet the configuration sheet depending on some parameter.
2. Cells A10 to C10 and all rows below (A11:C11... etc) are used to define new custom tables. You use a master table defined by the application (B10), and define a new name for it (A10). Then you can write a filter (Look at the "[Filtering Data](#)" section in this document). And also sort the table by some columns. Different search columns are separated by ";", and you can also define an "**ASC(ending)**" or "**DESC(ending)**" order. For example: "State, ZipCode DESC" Again, any Sort expression valid for a DataView is valid here. In some places you can use tags. For example, if you write on C10 "**<#mytag>**" instead of "ProductId=7" the value of mytag will be used as a filter.
3. You can leave empty rows. For example, you can define a new table on row 10 and another on row 12 without writing anything on row 11. This allows you to better separate your tables.
4. On columns H and I you define custom formats. The name of the format goes on column H, and the definition on column I. Then you can use those cell formats on **<#format>** tags.
5. You can write whatever you want on column I, the text is not important and you can use it to know how the format will look like.
6. Column K is used to list the report variables. It is not used at all by FlexCel and you don't really need to fill it, but it is used by FlexCel designer to list the available report variables. If you do not write them here, FlexCel designer will not show them.
7. Columns M and N are probably the most important on the config sheet and are used to create reusable expressions.

About Expressions

It is recommended that whenever you have complex chains of tags, instead of writing them directly on the cell, you create an expression and then refer to it.

On the screenshot, we defined a **<#order>** tag that itself calls other report expressions to calculate its value. Then, if you want to write the result of order on cell A1, instead of writing the full chain of tags, you just write **<#order>**. On some places, like for example the name of the sheet or an image name this might be the only choice, as the name length is limited to 32 characters. You can use also parameters on Expressions, and for Example define **<#order(row)>**. For more details on how to do this, see the [Expression parameters](#) demo.

Defining Custom Formats

Custom formats are normally straightforward to use.

For example, you might format I10 with a blue background, name it "Header" in cell H10, and then use the format inside a **<#format cell(Header)>** in the template. But this will apply the **full** format in cell H10 to the new cell. This means that besides the blue background, all other properties will be copied too. The new cell will also have the same font, same alignment, same numeric format than H10.

Now let's imagine we have a full row that we want to be blue if some condition applies. If we write **<#format row(header)>** in a cell on the template, then all other attributes besides the background will be applied to the row. This might mean that all the cells will be aligned to the left (if cell I10 was aligned to the left), and this is not what we want. We want to apply only the background color on I10, keeping everything else as it is defined on the template.

To do this, you can define **partial formats**. You define a partial format by adding one or more attributes to the name of the format. For example, if you name the format **Header(background;font.color)** only the font color and the background color of cell I10 will be applied, not the full format.

You can add as many properties as you want to be applied for the format separating them by semicolons (";"). Also, you can use **negative** properties by preceding them with a "-" sign. For example, the following definition: **Header(All; -Border; -Font)** will apply all attributes in cell I10 except the border and the font.

The list of properties you can write in a cell is the following:

Name	Description
All	Applies the all the formats in the cell. Defining a format named "header" is the same as defining one named " header(All) ". You will normally use the All format when excluding formats. For example, " header(All, -Border) " will apply all the formats except the border. " header(-Border) " would not apply any format.
Border	Applies the four borders. " header(Border) " is the same as " header(Border.Left;Border.Right;Border.Top;Border.Bottom) "
Border.Left	Applies the left border.
Border.Right	Applies the Right border.
Border.Top	Applies the Top border.

Name	Description
Border.Bottom	Applies the Bottom border.
Border.Exterior	<p>This is a special setting, used to apply the borders only on the outer bounds of the range. For example, if fmt is defined as "fmt(border; border.exterior)", and you write <#format range(a1:c5;fmt)> then the top row of the range (row 1) will be formatted with the top border format of the cell, the left column or the range (column a) will be formatted with the left border format of the cell, and so on. Inner cells in the a1:a5 range will not have any border applied.
</p> <p>This tag alone has no effect; you always need to use it together with other border tags. For example "fmt(border.exterior)" will not do anything, you need to write "fmt(border; border.exterior)", "fmt(border.top; border.bottom; border.exterior)" or something similar.</p> <p>Other properties (like background) are not affected by border.exterior, they will still apply to the whole range.</p> <p>Note that this tag only applies to the <#format range> tag. It makes no sense in <#format cell>, and you cannot use it in <#format row/column> since format for columns and rows does not support this.</p>
Font	Applies all the font properties. " header(Font) " is the same as " header(Font.Family;Font.Size;Font.Color;Font.Style;Font.Underline) "
Font.Family	Applies the font name.
Font.Size	Applies the font size.
Font.Color	Applies the font color.
Font.Style	Applies the font style (bold and <i>italics</i>) Underline is not included in Font.Style.
Font.Underline	Applies the underline.
NumericFormat	Applies the numeric format of the cell.
Background	Applies the background color of the cell. " header(Background) " is the same as " header(Background.Pattern;Background.Color) "
Background.Pattern	Applies the fill pattern for the cell.
Background.Color	Applies the fill color for the cell.
TextAlign	Applies the horizontal and vertical alignment in the cell. " header(TextAlign) " is the same as " header(TextAlign.Horiz;TextAlign.Vert) "
TextAlign.Horiz	Applies the horizontal alignment in the cell.
TextAlign.Vert	Applies the vertical alignment in the cell.
Locked	Applies the "Locked" attribute in the cell.

Name	Description
Hidden	Applies the "Hidden" attribute in the cell.
TextWrap	Applies the Word Wrap setting for the cell.
ShrinkToFit	Applies the Shrink to fit setting.
Rotation	Applies the Rotation.
TextIndent	Applies the indent.

IMPORTANT

You need to specify what the format applies to in the format definition, but not when calling the format. You might define a format as `header(border)`, but you need to call it with `<#Format Cell(header)>`, not `<#Format Cell(header(border))>`. The second way will not work.

With partial formats you can apply the different formats as "layers" in your document, one independent from the other. You can apply as many `<#format cell>` and `<#format range>` tags as you want in a cell, as long as the applied formats are different all of them will be applied.

WARNING

A last word of caution: As powerful as custom formats are, remember that they are only useful if you want to conditionally format cells. If the format is static, just format the cell as you want it. We have seen too many customers "over-engineering" reports with lots of format cell tags, when the simplest solution would be to format the cells directly in the template with Excel and not use any tag.

Also remember that you can use Excel's **conditional formatting** in almost every case you could need custom formats. But Excel's conditional formats are much simpler to use and maintain. So for example, if you wanted to paint a cell red when it is bigger than 100, you could define a custom format `Warning(Background)` and then use a `<#if(<#cell> > 100;<#format cell(Warning)>>` or you could just set a conditional format in the cell in the template making it red when it is bigger than 100.

FlexCel fully supports Excel conditional formatting, and you should always prefer to use that if possible.

You can see an example of how to use custom formats to have alternating rows in the [Multiple Sheet Report](#) demo.

NOTE

As an alternative to defining the formats in the config sheet, you could also define them as **User Defined Formats** and add them with `FlexCelReport.SetUserFormat`. For an example of using user defined formats, look at the [User Defined Formats](#) demo.

Filtering Data

You can filter the data you are going to use directly in the template by writing a filter in the config sheet:

8	Data		
9	Table Name	Source Name	Filter
10	VIPEmployees	Employees	VIP_LEVEL > 0
11			

Here you must be aware that generally **filtering is done locally**: that is, we fetch the full table from the database, and then apply the filter to discard the unwanted data. This makes Filters inefficient except for small tables. In general, you will want to filter in the SQL, not in the config filter column as a way to avoid fetching all records from the database. But sometimes when you can't modify the data, and there aren't too many rows, filtering in the config sheet can be an option.

NOTE

When using **LINQ to SQL** as datasources then **filtering is done in the server**, not in the client. The data fetched from the server will only be the data we need, not the full dataset. This is because LINQ delays sending the SQL to the server to after the filters have been applied, and so it sends the correct filtered SQL.

This makes filtering with LINQ to SQL very efficient, and not different from filtering directly in the server by changing the SQL.

Filter Syntax

The exact syntax of the expression you have to write in the "Filter" cell depends on the technology you are using:

1. **DataSets**: You can write any filter expression available on a DataView, for more info search at the DataView.Filter documentation on the .NET framework documentation.
2. **LINQ**: When using LINQ, FlexCel provides two different syntaxes for the filters:
 - Simple built-in parser: By default, any expression you write in the "filter" cell will be parsed by a simple built-in parser in FlexCel. The syntax of the built-in parser is described in the section [Built-in parser syntax](#) below.
 - Linq "Where" filter: If you are using Entity Framework or any other technology that includes "Where(string)" extension methods that allow sending direct SQL "where" strings to the server, you can write those strings in the filter, by adding an "@" before the string. For example: "@it.Field1 like '%test%'" When using this filter style, you need to prefix all fields with the table name. By default, Entity Framework uses "it" for all table names, and that's why we used "it.Field1" in the example.

Built-in parser syntax.

The filter expression consists of **operators**, **tokens** and **functions**.

- The operators you can use in a simple filter are, by order of precedence (lower to greater):
 - "**AND**", "**OR**": As in "Field > 1 AND Field < 2"
 - "<", ">", "<=", ">=", "<>", "=": As in "Field = 1"

- **"NOT"**: As in "NOT (Field = 3)"
 - **"+" , "-"**: As in "Field1 + 1 = Field2 - 1"
 - **"*" , "/"**: As in "Field1 / 2 = Field2 * 2"
 - **"(" , ")"**: As in "(Field1 + 1) * 2 = 4"
- The allowed Tokens are:
 - **FieldName, or [Field Name]**: Use the square brackets when there are spaces in the name.
 - **'Strings'**: Use single quotes for strings. To enter a single quote inside the string, repeat it twice, as in "Field1 = 'I'am me'"
 - **Numbers**: You should use "." for floating point numbers, no matter what your locale is. For example: "Field1 >= 1.5"
 - **SQL92 Date Literals**: You can use either DATE 'yyyy-MM-dd', TIME 'HH:mm:ss', TIMESTAMP 'yyyy-MM-dd HH:mm:ss'. For example, Birthday = DATE '1972-09-08'
 - **Ole Date Literals**: Ole date literals use the syntax: {d 'yyyy-MM-dd'}, {t 'HH:mm:ss'}, {ts 'yyyy-MM-dd HH:mm:ss'}. For example, Birthday = {d '1972-09-08'}
 - **Booleans**: You can use the constants "true" and "false", as in "Married = true"
 - **null**: This is used for null values, and it uses the .NET definition of a null (where null == null), not the SQL definition (where null <> null). This means that different from SQL, you can use this filter: "Field <> null" and it will work as expected.
 - The allowed Functions are:
 - **ISNULL**: Returns true if a field is null, and in this implementation it is similar to comparing the field to null. IsNull(field1) is the same as field1 = null
 - **YEAR**: Returns the year of a date. For example you could have a filter: Year(field1) = 1999
 - **MONTH**: Month from a date.
 - **DAY**: Day from a date.
 - **STREQUALS**: Syntax is StrEquals(str1, str2, ignorecase) or StrEquals(str1, str2, samecase). Returns true if both str1 and str2 are the same.

Relating Tables in the Template

[Data Relationships](#) are typically defined in code, but FlexCel also allows to define them in the template. You use the filter section of the config sheet to do that:

	A	B	C
1	Configuration Sheet		
2			
3	Included Conigs:		
4	Data Sources:		
5	Application to run:		
6			
7			
8	Data		
9	Table Name	Source Name	Filter
10	RELATIONSHIP	Countries->Quarters	saleyear->saleyear;country->country
11			

- In column A (Table Name), you have to write the word "RELATIONSHIP". The case is unimportant, but we use a convention of all upper-case.
- In column B (Source Name), you write the Master and the Detail tables that form the relationship, separating them with an arrow. The Master goes first, then the arrow, and finally the Detail: ** Master -> Detail**.
- In column C (Filter), you write the fields in the Master that relate to the fields in the Detail. If there is a single field that relates the Master and the Detail (the most common case), you would just write **MasterField->DetailField**. If there are multiple fields in the relation, you separate each with a semicolon: **MasterField1->DetailField1;MasterField2->DetailField2**

The example [Master detail on one table](#).

Grouping Tables

Sometimes you might have the data for a master-detail report into one table, and you want to create two different tables based on the value of a key field.

You can use the **DISTINCT** filter on the config sheet together with the **RELATIONSHIP** tag to get this effect. Look at the [Master detail on one table](#) demo for more information on how this is done.

NOTE

Whenever possible, do not use this grouping as the normal way to get data. While it simplifies the data layer, it also fetches a lot of repeated information from the database.

For example: If you

```
select * from table1 join table2 on (table1.key = table2.key)
```

you will get results like this:

key	field-from-table1	field-from-table2	field-from-table2
table1.value1	table1.value2	table2.value1	table2.value2
table1.value1	table1.value2	table2.value3	table2.value4

key	field-from-table1	field-from-table2	field-from-table2
table1.value1	table1.value2	table2.value5	table2.value6

In this simple case, you fetched 12 values from the database. If you had made 2 different selects, you would have fetched only 8 values (table1.value1 and table1.value2 for the first table, and table2.valueN for the second). Depending on the amount of data, there might be a lot of repeated fields on the join.

Splitting Tables

The same way sometimes you might want to group tables, other times you might want to split them into groups of n records. For example, you might want to create a 5 column report, and you need to split the master dataset into groups of 5 records in order to fill the columns.

This is where the **Split(source, number of records)** tag can be useful. In short, you write this tag on the "Source Name" column in the config sheet, specifying the table to split on the "**source**" parameter, and how many records you want on each group on the "**number of records**" parameter. This will create a new table that you will name on the "Table Name" column of the config sheet. You can then use this new table as the master on a master-detail relationship with the original table. The generated master has no columns, but you can use the pseudocolumns (#RowPos or #RowCount) just fine. Each record of the master is related to "number of records" records on the detail.

Note that the generated master table is a "pseudo table" in the sense that it has no columns or data, but it has $(\text{DetailRecords.Count} - 1) / \text{NumberOfRecords} + 1$ rows. Also the relationship between the master and the detail is not on real columns, since there are no columns on the master. This creates a limitation on how you can use those tables, and it is that the master should have the detail as a direct child. You cannot have other __ range__ between them, or FlexCel will complain.

Take a look at the [Split Datasets](#) demo for more information.

Retrieving TOP N Records from a table

You normally should filter the data when retrieving it from the database (for example with the SQL "Select top 10 * from customers"). But if this is not possible, you can use the **Top(source; number of records)** tag to filter this from the template.

8	Data	
9	Table Name	Source Name
10	orders	TOP(AllOrders;20)
11		

WARNING

Be careful with this tag. If your table has 10,000 records and you only need 10, fetching them all from the db in order to use only 10 is not a smart idea. Take a look at the [Fixed Forms With Datasets](#) demo for more information.

Ensuring a table has at least N records.

Sometimes you want to ensure that at least one record of a table exists, and if it doesn't, provide a default value for the missing fields.

You might do this with the **AtLeast(source; minimum number of records; [default value]; [multiple of])** tag. In a way, this is the opposite of the **TOP(...)** tag discussed above.

For example, you might define the following in the config sheet:

8	Data	
9	Table Name	Source Name
10	detail_1	ATLEAST(detail;1)
11	detail_5	ATLEAST(detail;5;---

In this example, the table detail_1 will have at least 1 record. If detail is empty, as we didn't specify a default value, it will return a record with all null values. The table detail_5 will have at least 5 records, and if detail has less, all records after the detail record count will return "---" in all their fields.

NOTE

You can't specify a default value for every field; it will be the same for all of them. But you can use some `<#if(detail_5.field="---";something;else)>` tag in the template to provide different values.

Ensuring the record count in a table is a multiple of a value.

There might be cases where you want to ensure that a dataset has for example 20 records, or 40 or 60, etc. This way, you can ensure that all pages in a master-detail report have the same number of rows. You can achieve this using the **AtLeast(source; minimum number of records; [default value]; [multiple of])** tag, and specifying the number in the "multiple of" parameter.

There is an example of this use at the [Fixed Footer](#) demo

Creating empty datasets with N rows.

Sometimes you might need a simple table with no columns and one or more rows to use as datasource of the report. For example, if you want to repeat a range n times, you could create an empty dataset with 3 rows, and put the range inside a `__dataset__` name. You can do that directly from the config sheet, by defining a datasource to be **ROWS(N)**. A dataset with a single row could be defined in the config sheet as `Dual = Rows(1)`. For an example on how this is done, please look at the [Balanced Columns](#) demo, the table "master" in the config sheet.

Joining Tables

When you have a "full row" report like the one in the image below, we insert a row for each record in the table.

	A	B	C	D
1				
2	Some range			
3				
4				
5	Merged cell			
6				
7				
8				

As you would expect, if you have a merged cell (in green) below the range being inserted (in blue), the merged cell will move down too:

	A	B	C	D
1				
2	Some range			
3				
4				
5				
6				
7				
8				
9				
10				
11	Merged cell			
12				

And the same will happen with ranges of cells like Print_Area or others. We are inserting full rows, so everything moves down.

But now, let's imagine we have 2 independent ranges we want to grow in parallel. We can do it with "-" ranges instead of "_" ranges:

	A	B	C
1			
2	Some range		some other range
3			
4			
5	Merged cell		
6			

In this case, when the blue range runs, the cells will only be inserted in columns A and B.

	A	B	C	D
1				
2	Some range		some other range	
3				
4				
5				
6				
7				
8				
9				
10				
11	Merged cell			
12				

But as a side effect of this, the merged cell will be broken (as you can see in the image above). After the blue range grows, the orange range will run and the cells will move down in column C too:

	A	B	C	D
1				
2	Some range		some other range	
3				
4				
5				
6				
7				
8				
9				
10				
11	Merged cell			
12				

And now the merged cell is all green again (as expected), but it isn't a full merged cell anymore: It is a merged cell in A:B and a different cell in C. Something similar happens with names, when the blue band grows it can't grow a name which goes from column A to C, because not all cells in that range have moved. And when the orange range grows, the name can't grow either since now columns A and B aren't moving.

There is no real solution to this: Due to the order the bands run, having two parallel _ ranges isn't the same of one big __ range. Merged cells might break and ranges might not update if they span over the two _ ranges running.

One solution is of course not to put any merged cell below the _ ranges which are not fully inside the columns of one of the _ ranges. And do the same with names.

But there is another solution, which is to "Join" the datasets inside "some range" and "some other range" so they form a bigger dataset with the columns of both, and then run a single full row report in the bigger range.

Say for example that you want to run two parallel reports on the tables:

Customer

Name	CustomerID
...	...

And

Employee

Name	EmployeeID
...	...

In the config sheet, you could define a new table "**CustomerAndEmployee**" as **JOIN(Customer;Employee)**

This will define a new table with 4 columns:

CustomerAndEmployee

Customer.Name	Customer.CustomerID	Employee.Name	Employee.EmployeeID
...

And you can use this new **CustomerAndEmployee** table to run the report.

Things to note:

1. The table "CustomerAndEmployee" has as many rows as the maximum of "Customer" and "Employee". Say for example that you have 3 customers and 5 employees, then CustomerAndEmployee will have 5 records. The last 2 records of CustomerAndEmployee will have Customer.Name and Customer.CustomerID null.
2. If "Customer" and "Employee" have columns with the same name, you will need to prefix them with the table name in order to show them. In our example, both Customer and Employee have a "Name" column. So you can't just write "<#CustomerAndEmployee.Name>" inside a cell, because you wouldn't know which Name it is referring to (Customer.Name or Employee.Name). For this case, you need to write "<#CustomerAndEmployee.**Customer**.Name>" and "<#CustomerAndEmployee.**Employee**.Name>".
3. Even if the prefix is only necessary for the case when there are repeated columns, it might be good practice to always prefix the table names in joined tables. This way you know you are always accessing the right column at the right table.
4. JOIN joins unrelated tables adding the columns of each one after the other, but it doesn't do any master-detail or relationships. It will just show record1 of table1 together with record1 of table2, and record2 of table1 with record2 of table2 and so on. When one of the tables runs out of records, the next records for the columns in that table will be null.

5. While this command doesn't use any extra memory (joined tables aren't copied in memory) and has no performance penalty, if you are doing SQL it will be simpler to just use SQL to construct the joined table.

Union of Tables

While "JOIN" in the section above creates a table with the columns of a list of datasets, UNION creates a table with the rows of a list of datasets. It works in a similar way as the union command in SQL.

In general union is not really needed: You could just write the 2 ranges one after the other for the two tables. But conceptually, sometimes it feels better to join the data of similar tables into a big one and have a single named range for that table.

NOTE

The "Union" table will have all the columns of all the tables used to create it. Normally you will use tables with the same columns, but if a column is present in one table but not in another, union will show the values of the column for the table that has it, and null for the other.

You can look at the "Join and Union" example for an example on how to use JOIN and UNION tags.

Direct SQL in templates

Depending on your needs, you might want to write SQL commands directly on the templates and avoid having a Data module on your code. This allows the users to modify the data they need by modifying the xls file, and without recompiling the executable.

IMPORTANT

Be aware that allowing your users to directly write the SQL commands can mean a big **security risk**.

For example, a user could use the connection you give him to execute the SQL: "drop table users" instead of a normal select. While FlexCel does a little validation on the SQL written by the user (for example, it cannot contain ";" or "--", it has to start with "Select" etc) SQL is a very powerful language and there can always be a way to execute a command on the server. And, even if the user does not manage to execute a command, he might always do a "Select user, password from users" or similar command, and get dbadmin access to the database.

You must supply a readonly connection, and with rights limited to the tables you want the user to see.

The steps for allowing Direct SQL on the templates are:

1. Provide a connection to the report, by using FlexCelReport.AddConnection()
2. On the config sheet, "Source Name" column, add a string like "**SQL**(Select * from clients)". Give this table a name, and you can use it as any other table on the report.

Note that also for security reasons, you can't replace expressions inside the SQL string. For example, you can't write "SQL(select * from customers where cust_id = <#custId>)" This would open another security hole and allow for SQL injection attacks.

SQL Parameters

You need to specify database parameters to be able to actually pass information to the SQL.

.NET can use both positional ("?",) and named ("@name" or ":name") parameters, and some data providers will accept one or the other. As we want to keep the template database independent (so you can replace the db backend without changing templates) **all parameters on the template are named with a preceding "@"** ("@name"). When using a db that needs positional parameters (like ADODB) or has another syntax for named parameters (like Oracle that uses ":name") the SQL will be automatically converted by FlexCel into the needed one before sending it to the db. FlexCel includes support for the most common cases, and when it does not know which db it is, it will "guess" that the db uses named "@" parameters.

When you use a db FlexCel does not recognize, you need to let FlexCel know which kind of parameters your db needs.

For this you use the [SqlParameterReplace](#) and [SqlParameterType](#) properties on [FlexCelReport](#).

You can see a demo on how to do it on the [Direct SQL](#) example; more information is also available there.

User Tables defined on the Template

Sometimes Direct SQL is not an option, because you have your own data layer that you want to use, or because it is too big a security risk to let your users run arbitrary SQL commands. But you would like the advantages of Direct SQL. That is, to specify the data directly on the template, so everything is self-contained on the template file and you can change your reports without recompiling the application.

You can use the **USER TABLE(Params)** tag to achieve this.

User table(Params) is a very simple tag, but it allows a lot of things. You write it on the Source Table column in the config sheet, and you can add an additional parameter on the Table Name column. You can leave the "Sort" and "Filter" columns empty or with values, their values do not matter.

For every "User Table(Params)" on the template, the **event UserTable** will be called on the report. Anything you write in "Params" will be passed as arguments to the event, without any further processing by FlexCel. Also, the value you write on the Table Name column will be passed to the event. You will normally want to use this second parameter to tell FlexCel how you want to name the table you are creating. Note that the name you write on the "Table name" column is not guaranteed to be created; this all depends on what you write on the event handler. Also, note that you **must** write something on the tablename column even if you will not use it on the event, or the row will be ignored.

So, how do you use it? Imagine you write a tag: `User Table(CUSTOMER)`, and define the event `UserTable` so each time it gets the parameter "Customer" it loads the customer table from the database and adds it to the report. This way you are actually telling FlexCel which tables it needs to use on the template, allowing you to add new ones (from a list of available tables on the application) or remove existing tables without changing the code.

Another way to use this tag could be if you have your own API to access the database, with your defined commands, permissions and validations. You could pass the API command as a parameter on the `<#User Table>` tag, and use this parameter inside the event to execute the API and add the generated table to the report. You would write something like `<#User Table(get table customer on customerId < 100)>` on the template, and on the `UserTable` event on the report, execute the parameter against your API, and add the resulting dataset.

WARNING

When using the "User table" tag to pass arbitrary API commands to the application, please remember to validate permissions on the `UserTable` event to see if the user is allowed or not to run the query. Forgetting to do so could generate a big security hole on your application, the same way as the `SQL` tag could.

For more information on this tag, see the [User Tables](#) demo.

Debugging Reports

Introduction

Whenever you hit enough complexity in your reports, you will get expressions that do not behave as you expect, and you will need tools to investigate what is really happening under the hood. This chapter speaks about those tools.

FlexCel Reports are declarative instead of imperative. This means you describe what you want using report tags, but you do not write code to tell the computer how to do it. In a way, FlexCel Reports are similar to SQL, and different from normal code.

An "imperative" report could be done with `XlsFile`, with some code like this:

```
XlsFile xls = new XlsFile();
xls.NewFile();
xls.SetCellValue(1, 1, "Table");
if (Table["field"] == null) xls.SetCellValue(1, 1, "error"); else xls.SetCell
Value(1, 1, Table["field"]);
```

A "declarative" report would be a template with the text "Table" in cell A1 and the tag

```
<#if(Table.field =;error;<#Table.field>>
```

in cell A2.

Declarative reports can sometimes be more "resistant" to bugs because they are simpler and so it is easier to see what is wrong. But on the other hand, they are also more difficult to debug when there is a bug, because there is no code you can step into with a debugger.

In the XlsFile example above, you could set a breakpoint in the last line and evaluate the values of the variables before execution. In the FlexCel Report there is no way to set a breakpoint, since there is no code to execute. But you can still find out what is going on, and this is what we will explain here.

There are two main causes of errors in FlexCel reports; syntax errors and logical errors. They are covered below.

Dealing with Syntax Errors

Those are the easiest to deal with. The same way a "Code" report will not compile if there is a syntax error, a FlexCel report will raise an Exception when "compiling" the template if it finds anything it cannot understand. The error message will normally tell you what is happening and where, and there should be no big problem in fixing it.

If for example you write "<#tag" in a cell, you will get an error telling you that there is a closing tag ">" marker missing.

NOTE

When dealing with syntax errors, FlexCel leans over the "pedantic" side. This means that it will not let pass anything that is ambiguous or not syntactically correct and report every little error.

We believe that by checking the errors when compiling the templates as much as possible, we minimize the runtime errors that are harder to catch and can have worse consequences.

We also made a big effort to try to have all error messages as helpful as possible: For example instead of a generic **invalid parameter** error, you should get a longer sentence indicating what parameter is invalid and why.

But this is not always possible. Sometimes FlexCel just forwards the error of a lower layer, and if the lower layer doesn't provide more information, neither can FlexCel. In this particular example, some ADO database providers might return **invalid parameter** when they fail to establish a connection, and in this case FlexCel will just forward that exception to you.

Now, in some situations it might be useful to see all syntax errors at once, and for this FlexCel offers a "**ErrorsInResultFile**" mode. In this mode, all errors related to tags will be written to the cell where the tag is, instead of raising exceptions, and the report will continue to generate. Errors will have a yellow background and red text.

So if you have the template:

	B
7	Value
8	<#invalld>

And "Invalld" is not defined, then instead of an exception when running the report it will complete successfully. In the result file you will see:

	B
7	Value
8	ERROR: Report Variable "INVALLID" is not defined.

This mode does not cover all errors (for example a named range for a non-existing table will still raise an error), but it covers the most common issues. The others still need to raise an exception, since if for example you have a range named “__table__” and no “Table” in the report, there is no place where FlexCel could write this error inside the xls file. The error must be in a cell for this mode to work.

There are two ways to enter **ErrorsInResultFile** mode. The first one is to change it in the code before running the report, by changing the `FlexCelReport.ErrorsInResultFile` property in `FlexCelReport`.

For example:

```
using (FlexCelReport report = new FlexCelReport())
{
    report.ErrorsInResultFile = true;
    report.Run("template_file.xlsx", "result_file.xlsx");
}
```

The second way is to write it directly in the template, inside the config sheet.

For example:

	M	N
8	Expressions	
9	Name	Expression
10	<#ErrorsInResultFile>	
11		

The `<#ErrorsInResultFile>` tag can be anywhere in the Expressions column, and you do not need to write anything in the Expression definition.

This second way to enter `ErrorsInResultFile` mode is better when you are editing a template and want to do a debug without modifying the code, while the first way is better if you are automating testing and do not want to modify the templates.

WARNING

Remember that this mode is a **debugging** mode, and you should turn it off for production. You do not want to ship a file containing error messages in cells to your customers.

Dealing with Logical Errors:

These kinds of errors are harder to deal with, are more subtle, and can pass without notice since they do not raise any error on FlexCel side.

For example, imagine that you have this expression:

```
<#if(<#tagval>=<#refval>;OK;Error)>
```

This is a valid tag, and the report will compile and run without issues. Let's imagine now that `tagval` is the number 1, and `refval` is "l" (lowercase L). So, when `tagval` is 1, the condition will evaluate to false, and you will get the "Error" label instead of "OK".

With this font in particular this can be a hard to spot problem, because as we said before, you cannot really put a breakpoint in the expression and see what is inside tagval or refval.

Here is where the “**Debug**” mode can help.

As with the “**ErrorsInResultFile**” mode, there are two ways to activate debug mode. The first is by code, by setting the “**FlexCelReport.DebugExpressions**” property in the FlexCelReport to true. Again, setting it in code is useful when automating tests because you do not have to change the template.

And the second way is to write “<#**Debug**>” in the configuration sheet, the same way as in ErrorsInResultFile. This second way is preferred in most cases, f.i. if you do not want to modify the code, for example when testing a template.

The same remarks mentioned for the <#ErrorsInResultFile> apply to the <#Debug> tag.

In this mode, tags will not write their value to the cell, but they will rather write the whole chain of calculations made to arrive to the value.

For example, for our problematic expression:

```
<#if(<#tagval>=<#refval>;OK;Error)>
```

We will get this result in the cell:

	B
	if(<#tagval>=<#refval>;OK;Error): <i>“Error”</i> refval: <i>“1”</i> tagval: <i>1</i> <#tagval>=<#refval>: <i>FALSE</i> 9 Constant: <i>“Error”</i>

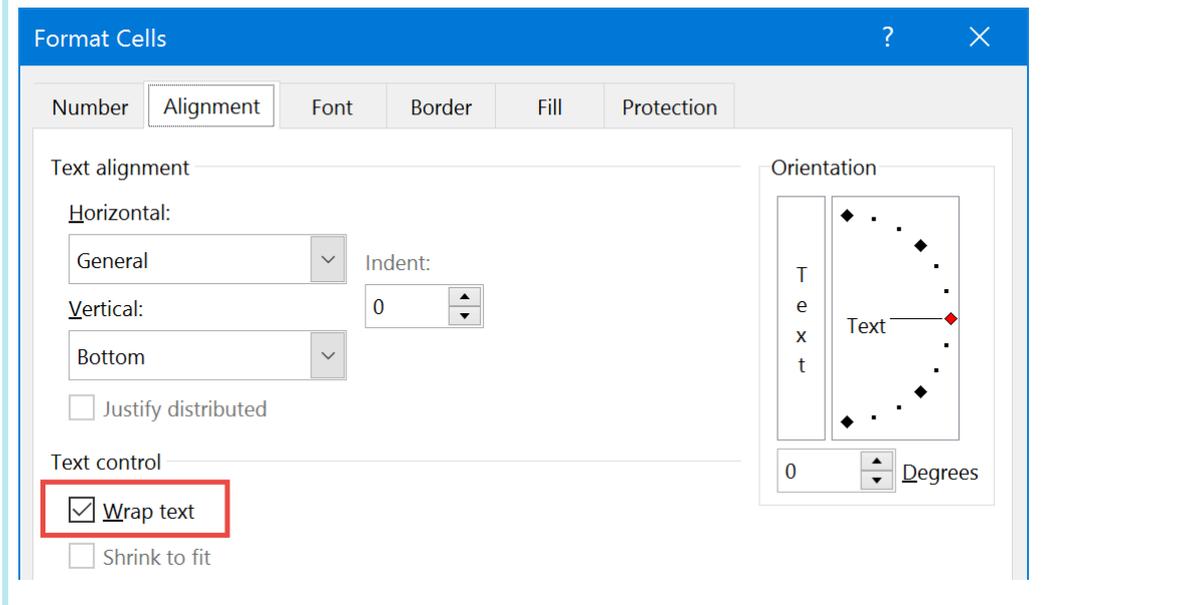
The format of the lines is as follows:

Each line has an **expression** in bold and a *value* in italics. Under that line and indented to the right we find the subexpressions used in the main expression and their values.

In this case, we can easily see that <#tagval> is “1”, <#refval> is “1”, and <#tagval> = <#refval> evaluates to false, so there is something wrong in the test. Having the full stack and all the intermediate values used to compute the main expression can be a valuable tool to find out what is going wrong, by allowing us to easily locate the exact point where the expression is not evaluating as it should.

NOTE

In order to correctly visualize the stack in the cell, you will have to set "Wrap Text" to true in the cell properties. If Wrap text is false, all lines will show in one line and it will be more difficult to read.

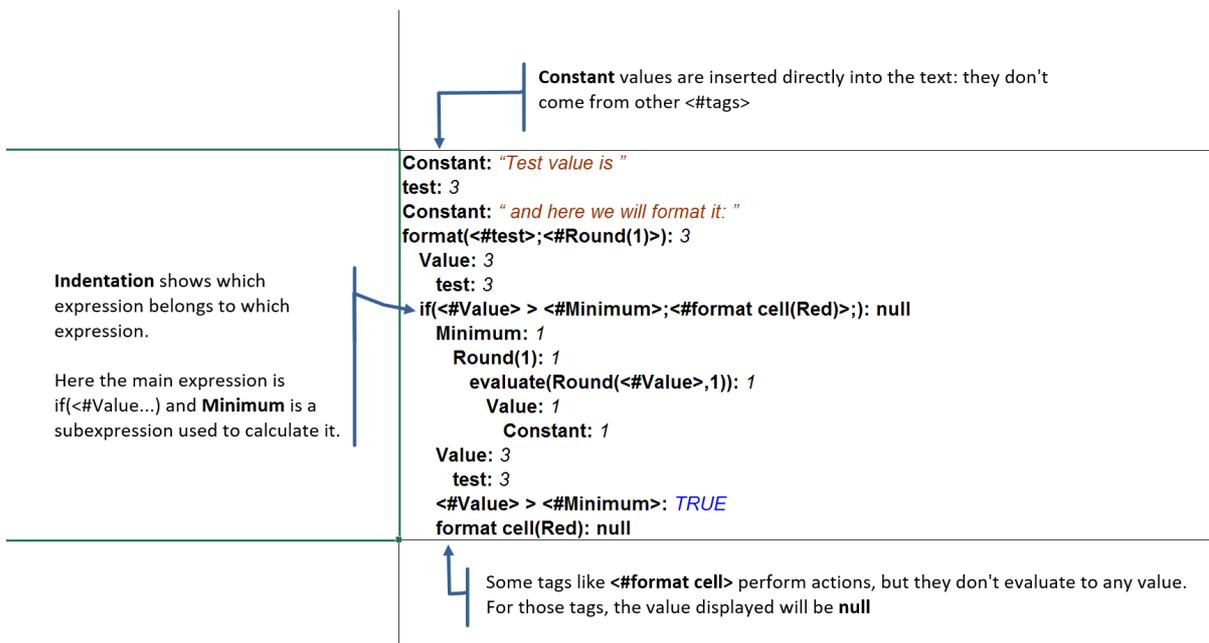


Understanding the stack can be a little difficult at the first time, but once you get used to it, it can be a great help. To end up this chapter, we will present a more complex expression and its corresponding stack explained. You can see it yourself at the [Debugging Reports](#) demo.

The original expression is this:

```
14 Test value is <#test> and here we will format it: <#format(<#test>;<#Round(1)>)>
```

And the result we get when running this template in "debug" mode is:



```
Constant: "Test value is "
test: 3
Constant: " and here we will format it: "
format<#test>;<#Round(1)>: 3
Value: 3
test: 3
if(<#Value> > <#Minimum>;<#format cell(Red)>): null
Minimum: 1
Round(1): 1
evaluate(Round(<#Value>,1)): 1
Value: 1
Constant: 1
Value: 3
test: 3
<#Value> > <#Minimum>: TRUE
format cell(Red): null
```

Constant values are inserted directly into the text: they don't come from other <#tags>

Indentation shows which expression belongs to which expression.

Here the main expression is if(<#Value...>) and Minimum is a subexpression used to calculate it.

Some tags like <#format cell> perform actions, but they don't evaluate to any value. For those tags, the value displayed will be null

You can see we have 4 main expressions here (they show without indentation). The first one is the constant text **"Test value is"**, the second is the tag **"<#test>"** that evaluates to 3, the third is other constant text **"and here we will format it: "** and the fourth is an expression **"Format(<#test>;<#round(1)>)"** that evaluates to 3. So the result in "normal" mode of this cell will be "Test value is 3 and here we will format it: 3". As there is a "format cell(red)" tag in the evaluation stack, the result will be formatted in red format.

All tags shown below the "format" line are subexpressions used to compute it.

Debugging Intelligent Page Breaks

Sometimes intelligent page breaks won't give the results you expect, and you might need to debug them. You can enter intelligent page breaks debug mode similar to Report Debug mode and Errors in result file mode:

1. You can set [FlexCelReport.DebugIntelligentPageBreaks](#) to true in the code.
2. You can write a tag `<#Debug Intelligent Page Breaks>` anywhere in the Expressions column of the config sheet.

The details on how to debug are explained in the [Debugging Intelligent Page Breaks section of the API Guide](#)

FlexCel Reports Tag Reference.

This document lists all the built-in tags in the report engine. You can use any of the tags included here, or define your own.

Tag syntax

Tags are special commands that go inside <#...> delimiter. Those will be replaced by FlexCel when running the report.

Tags cannot be escaped, but you can always insert literal <# or ; characters by defining report expressions an values. Say for example that you wanted to enter the following formula:

```
<#if(<#tag2>=""<#tag1>;<#tag1>;<#tag2>>>
```

In the following expression, if tag 2 is empty you will just output tag1, else you will output tag1;tag2. The problem with the formula above is that the ";" between <#tag1> and <#tag2> will be interpreted as a parameter separator, and the formula isn't valid. For this particular case, you can write the full string inside double quotes ("). <#tag1> and <#tag2> will still be replaced inside the double quotes, but FlexCel won't be misled into thinking the ; is a parameter separator. This formula will work fine:

```
<#if(<#tag2>=""<#tag1>;"<#tag1>;<#tag2>")>
```

As a more general approach for other cases (including if you want to enter a literal "<#" into a formula) the solution is just to define report expressions or report variables with the text. So in the example above we could define a report expression:

Semicolon = ";"

and then write:

```
<#if(<#tag2>=""<#tag1>;<#tag1><#semicolon><#tag2>>>
```

Using expressions or variables you can escape any text you might need to, and it is the more general solution.

Constant

Syntax

Any text

Description

Any text that is not inside <#... > symbols.

Example

On the string "`<#tag1>Hello<#tag2>`" "Hello" is a constant.

Report Variable

Syntax

1. `<#Value>`
2. `<#Value;default>`

Description

A variable set in code before running the report. You can specify a default value (second syntax) that will be used if the variable does not exist. If no default value is provided and the variable does not exist, an exception will be raised.

Example

If you have a line of code `FlexCelReport.SetValue("Date", DateTime.Now)` before `FlexCelReport.Run`, the tag `<#Date>` will be replaced by the current date. If you have `<#Date>` in the template but do not do the `SetValue` command, a runtime error will happen. If you write `<#Date;No date supplied>` in the template, this tag will be replaced with the date if you set it on code with `SetValue`, or with the string `"No date supplied"` if not. There will never be a runtime error.

NOTE

Report Variables, User-defined Expressions and User-defined Functions use the same syntax, so it is impossible to differentiate between them. If you define a Report Variable with the same name as a User-defined Expression and/or User-defined function, they will be used on the following order: 1) User-defined Expression. 2) Report Variable. 3) User-defined function.

User-Defined Expression

Syntax

1. `<#Expr>`
2. `<#Expr(param1;param2...)>`

Description

Text on the Column "User defined Expressions" on the config sheet. When using parameters, you can use the `<#param1>` name inside the expression definition.

Example

If you define an Expression `CompleteName = <#LastName>, <#FirstName>` on the config sheet, the tag `<#CompleteName>` will be replaced by the expression.

NOTE

Report Variables, User-defined Expressions and User-defined Functions use the same syntax, so it is impossible to differentiate between them. If you define a Report Variable with the same name as a User-defined Expression and/or User-defined function, they will be used on the following order: 1) User-defined Expression. 2) Report Variable. 3) User-defined function.

User-Defined Function

Syntax

1. `<#UDF>`
2. `<#UDF(param1;param2;...)>`

Description

A user-defined function on the code. See the [User Defined Functions](#) demo for details.

Example

If you define a function **Proper(name)** on the report code, `<#Proper(test)>` on the template will be replaced by the results of the function.

NOTE

Report Variables, User-defined Expressions and User-defined Functions use the same syntax, so it is impossible to differentiate between them. If you define a Report Variable with the same name as a User-defined Expression and/or User-defined function, they will be used on the following order: 1) User-defined Expression. 2) Report Variable. 3) User-defined function.

Dataset

Syntax

1. `<#DataSet.Field>`
2. `<#DataSet.Field;default>`

Description

The value of **Field** on the current row of **Dataset**. If a default value is provided, it will be used when the field does not exist in the table or the table does not exist. You will normally want to provide default values when using metatemplates, together with the defined and preprocess tag. If you do not provide a default value and the field does not exist, a runtime error will happen.

There are two defined "pseudocolumns" that you can use on any dataset:

1. `<#DataSet.#RowCount>` will return the number of rows on a dataset.
2. `<#DataSet.#RowPos>` will return the current position on the dataset (the first record is 0).

NOTE

Normally you can write any character in the column name, including a column named similar to **"Data(Max)"**. But in some cases, like when for example there is a simple parenthesis, or if you have the character ">" inside the column name, you might need to quote the name. For example, if your column name is **"a("** you will have to write the tag as **<#"DataSet.a(">**. When quoting a name, you can include a quote inside by writing 2 quotes. For example, the column **a"b** can be written as **<#"DataSet.a""b">**

NOTE

FlexCel always searches for the first dot in the string to separate the table name from the field name. So if you have for example **<#one.two.three>**, FlexCel will understand that the table is "one" and the field is "two.three". If you wanted the table name to be "one.two" and the field to be three, you would have to use square brackets to keep together the table name:

<#[one.two].three>

You can see square brackets in action in the [Advanced LINQ](#) demo.

Examples

If you have a table "Customers" with a column named "LastName", **<#Customers.LastName>** will replace the value on the report.

<#if(<#Customers.#RowCount> = 0;<#delete row>;)> will delete a row if the dataset has no records. You can use it to delete the detail captions on master-detail reports when the master has no details.

<#Customers.LastName;No customer> will enter the value of the field LastName if such field exists in the database, or the string "No customer" otherwise.

<#Customers.LastName;> will enter an empty value if the field LastName does not exist.

<#Customers.ByName.LastName> will refer to the table "Customers" and the field "ByName.LastName". On the other hand, **<#[Customers.ByName].LastName>** will refer to the table "Customers.ByName" and the field "LastName"

Full Dataset

Syntax

<#Dataset.*>

Description

The whole current row of the dataset. Cells to the right will be overwritten. You can write other **<#Dataset.*>** and **<#Dataset.**>** tags inside the same cell, and they will have the value of the first one.

NOTE

It is recommended that you don't use **<#column width(autofit)>** in a **<#Dataset.*>** tag, since the **<#autofit>** tag will be copied to every row and applied for every one. If you want to use it, it is best to put the autofit tag in the Full dataset captions tag (**<#Dataset.**>**)

Example

If the cell A1="<#DataSet.*>, after the report column A will have the first column of the dataset, Column B the second column, etc. Any text previously on Column B will be overwritten. If you write "<#DataSet.*> <#if(<#DataSet.**>="Date";<#Format Cell(blue)>;>" in a cell, all columns written will have autofit, and when the column is named "Date" it will be formatted in blue

Full Dataset Captions

Syntax

<#DataSet.**>

Description

The whole row of column captions for a dataset. You normally use this tag before a **<#dataset.*>** and outside any named range.

Example

If you write **<#Dataset.**>** on cell A1, cell A1 will have the first column name on the dataset, B1, the second, etc. Text previously on B1 will be overwritten. If you write "**<#Dataset.**>** **<#ColumnWidth(autofit)>**" in a cell, it will autofit all the columns in the range

DbValue

Syntax

1. <#dbvalue(table;row expression; column expression)>
2. <#dbvalue(table;row expression; column expression; default value)>

Parameters

- **table:** Datable with the data, without quotes
- **row expression:** An Excel expression (any valid Excel formula that can include other **<#tags>**) that must return a number or be empty. If empty, the record number will be the current record. If a number, then that number is the record you want to retrieve. (starting at record 0)

- **column expression:** An Excel expression (any valid Excel formula that can include other <#tags>) that must return a number or a string. If a number, this means the position of the column in the datatable (with the first column being column 0). If a string, this is the name of a column in the table.

NOTE

As this is an expression, not a string, if you want to enter a constant string here (like "customer") it must be between double quotes. The expression ="customer" with quotes evaluates to customer (without quotes). If you write just customer, it will be an invalid expression.

- **default value:** A default value that will be used if the row or column doesn't exist. If you don't specify a default value and try to access an invalid row or column, an exception will be thrown.

Description

Returns any value of a table, letting you specify the record and column for the value. Note: This is an advanced tag, and you most likely don't want to use it. The most common application for it is when you want to do something depending on the value of the previous record (For example merge the cell if the value is the same as previous). Don't use it as a general tool. <#db.field> tags, ranges and aggregates should be enough for most needs, and they are a much cleaner and "functional" abstraction.

Examples

<#dbvalue(customers;<#customers.#rowpos> - 1;"customerid");> will return the value of the previous value in customers.customerid. Note that "Customerid" is in quotes as it should be an expression that returns a string, not a string. Also note that we defined an empty default value, so no exception is thrown when we are at the beginning of the table.

<#if(<#dbvalue(customers;<#customers.#rowpos> - 1;"customerid");> = <#dbvalue(customers;<#customers.#rowpos>;"customerid");>;<#Merge range(a3:a4)>;<#customers.name>)> will merge the cell if the value is the same as the previous one, or write the new value if values are the same.

Aggregate

Syntax

1. <#aggregate(agg function; dataset name and column)>
2. <#aggregate(agg function; dataset name; agg expression; filter)>

Parameters

- **agg function:** It might be SUM for adding the values, COUNT for counting the values, AVG for finding the average, MAX to find the maximum value and MIN to find the minimum value.
- **dataset name (and column):** Name of the dataset in which we want to aggregate the values. Note that this dataset doesn't need to be inside any named range, since we will use all of its records anyway. If "agg expression" is present, you don't need to include the column name, as the columns to aggregate will be taken from the expression. If not present, you need to include the column in which you want to aggregate.
- **agg expression:** This parameter is optional. An expression that will be applied to every record in the dataset.(any Excel function is valid here, and you can use any combination of Excel functions) Null values will be ignored, but will count to the total records when calculating the average. If not present, the values of the column specified in "dataset name and column" will be used. Note that when aggregating on COUNT, this parameter is ignored.
- **filter:** This parameter is optional. If present, it should be an expression that returns true or false. Again, any Excel formula is valid here. Only those records where the filter value is true will be used in the aggregate. When calculating the average, filtered records will not be used in the count.

Description

Aggregates a dataset and returns a unique value for all its records. You can use this tag to find for example the sum on a column in a dataset. Note that this tag can have a bad performance, as you need to load all data in memory in order to calculate the aggregate. If possible, it is preferred to do the aggregate directly in the database, for example using a "Group by" clause in the select SQL. This tag can be of use when you can't modify the datasets and you already have the data loaded, so you need to do the aggregation from the template.

Examples

<#Aggregate(sum;orders.orderid)> will sum all the values in the column order id of the table orders

<#Aggregate(avg;orders;<#orders.quantity>*<#orders.orderprice>)> will calculate the average of the quantity multiplied by the order price in table orders

<#Aggregate(min;orders;<#orders.tag>;<#orders.tag>> 0)> will calculate the minimum tag in orders that is bigger than 0

List

Syntax

1. **<#List(dataset name and column)>**
2. **<#List(dataset name; list separator; agg expression; filter)>**

Parameters

- **dataset name (and column):** Name of the dataset in which we want to get the values as a list. Note that this dataset doesn't need to be inside any named range, since we will use all of its records anyway. If "agg expression" is present, you don't need to include the column name, as the columns to aggregate will be taken from the expression. If not present, you need to include the column in which you want to aggregate.
- **list separator:** This parameter is optional. If not present it will default to a single space. This is the character that will separate the elements in the list. Note that if you want to use a semicolon here (;) you will have to write it in quotes (";") so it is not considered a parameter separator
- **agg expression:** This parameter is optional. An expression that will be applied to every record in the dataset.(any Excel function is valid here, and you can use any combination of Excel functions) Null values will be ignored and not added to the list. If not present, the values of the column specified in "dataset name and column" will be used
- **filter:** This parameter is optional. If present, it should be an expression that returns true or false. Again, any Excel formula is valid here. Only those records where the filter value is true will be used in the aggregate.

Description

Returns a string with all the values of a table one after the other, and separated by a delimiter. If the table has only one record, you can use **<#List(table.field)>** to get the value of the only record without having to define any **__table__** named range.

Examples

<#List(Employees.Lastname)> will return a string like "Smith Brown Perez", when the dataset has 3 records with the lastnames of Simth, Brown and Perez. As we didn't specify a separator, a single space will be used. In cases where you know Employees has only one record, you could have used this to avoid defining a **__employees__** named range.

<#list(employees.lastname;,)> will return a string like "Smith, Brown, Perez".

<#list(employees;" ; <#employees.firstname> & " " & <#employees.lastname>)> will return a string like "John Smith; Carl Brown; Jorge Perez". Note that as we wanted to use ";" as the list separator, we had to write it inside quotes.

If

Syntax

<#if(Condition; IfTrue; IfFalse)>

Description

A conditional statement. When "condition" is true, "IfTrue" expression will be evaluated, if not "IfFalse" will. For a description of the "Condition" format, see [Evaluating Expressions in the Reports Designer Guide](#)

Example

`<if(<#value>=1;One;Not One)>` will write "One" if the report variable "Value" is 1, and "Not One" if not.

ifs

Syntax

`<#ifs(Condition1; value1; Condition2; value2...)>`

Description

An "if-chain" of different conditions. If condition1 is true then this tag will return value1, if it is false it will evaluate condition2 and if condition3 is true return value2, and so on. If no condition evaluates to true, then this tag will return #N/A!. If you want to provide a default condition, make sure that the last condition is the value "true"

NOTE

If the condition always compares against the same value, you might use `<#switch>` instead. The examples below are simpler using `<#switch>`

Examples

`<#ifs(<#value>=1;One;<#value>=2;Two)>` will write "One" if the report variable "Value" is 1, "Two" if value is 2, and #N/A! if the value isn't 1 or 2

`<#ifs(<#value>=1;One;<#value>=2;Two>true;Something else)>` will write "One" if the report variable "Value" is 1, "Two" if value is 2, and "Something else" if the value isn't 1 or 2

Switch

Syntax

`<#switch(SwitchValue; value1; result1; value2; result2...[default])>`

Description

This tag will compare "SwitchValue" against value1, value2, etc in order. If SwitchValue is equal to any of the value_n, then result_n will be returned. You can provide a default value as the last parameter. If no value matches SwitchValue and you have a default parameter, then the default will be returned. The default is inferred from the number of arguments: When the number of arguments is odd (3, 5, 7...) there is no default value. If the number of arguments is even, then the last parameter is the default.

Examples

`<#switch(<#value>;1;One;2;Two)>` will write "One" if the report variable "Value" is 1, "Two" if value is 2, and #N/A! if the value isn't 1 or 2

`<switch(<#value>;1;One;2;Two;Something else)>` will write "One" if the report variable "Value" is 1, "Two" if value is 2, and "Something else" if the value isn't 1 or 2

Evaluate

Syntax

1. `<#evaluate(expression)>`
2. `<#evaluate(expression;loop count)>`

Parameters

expression: An expression to evaluate. For a list of possible expressions, see [Evaluating Expressions in the Reports Designer Guide](#). loop count: How many times to evaluate expression. If this parameter is not used, expression will be evaluated one time.

Description

This tag will evaluate an expression and output the final result. You can see it as a "static" formula. Different than `<#=()>` tag, expression will be evaluated each time a value is needed, so you can have relative addresses.

A loop count bigger than 1 is useful if you store tags in the database itself. For example, if you have a field in the database "db" named "Expression" with value "`<#other tag>`", then

`<#evaluate(<#db.expression>;2)>` will evaluate first the value of expression, find out it is `<#other tag>`, then evaluate again `<#other tag>` and write the value of other tag in the cell. ↵

Example

`<#evaluate(A1+A2*2 & left(a3,2))>` will output a string consisting on `A1+A2*2` concatenated with the 2 first characters of `a3`.

If a report variable customer has the value "`<#FirstName> & <#LastName>`", then

`evaluate(<#customer>;2)` will first evaluate `<#customer>` and return the string "`<#FirstName> & <#LastName>`". Then, as loop count is 2, it will evaluate the string "`<#FirstName> & <#LastName>`" and return the first and last name concatenated.

Equal

Syntax

`<#=("Cell")>`

Description

Replaces the tag with the referred cell content. "Cell" might be a reference to another sheet. Note that "Cell" will be evaluated at compile time, so it won't change when you copy the range. (It always behaves like an absolute reference, on the style \$A\$1). If you want to have a cell reference that is dynamically evaluated at fill time, use `<evaluate("Cell")>` tag. Note: Almost the only place where this tag makes sense is on sheet names. For other expressions it is better to define an expression on the config sheet and use it instead of a cell reference. When using a sheet name you can not always know which one is the config sheet, so it is not safe to use expressions, and you need the "=" tag.

Example

If you name a sheet `<#=(Sheet2!A1)>` the sheet name will be replaced by whatever you write on cell A1 on Sheet2. On cells, define a named expression on the config sheet and use it instead of this tag.

Include

Syntax

1. `<include(file; named range; shift type)>`
2. `<include(file; named range; shift type;static/dynamic)>`
3. `<include(file; named range; shift type;static/dynamic;CopyRowsOrCols)>`

Parameters

- **file:** Filename to include. The path is relative to where the current template is. If you are inserting from a stream (for example from a database) you need to assign the GetInclude event. See [Templates On The Exe](#) demo for more info.
- **named range:** Named range on the included file that determines which cells will be included. If you leave this parameter empty, the full used range in the active sheet will be included.
- **shift type:** How the existing cells will be shifted to insert the new ones. There are five possibilities: "_", "_", "I_", "II_" and "FIXED", to move existing cells the full row down, only down, full column right, only right or not insert anything respectively.
- **static/dynamic:** This can be the string "Dynamic" or "Static" (without quotes), or omitted, in which case it is assumed to be "Dynamic". Dynamic includes will run inside the main report when inserted; this is the normal behavior. Static includes will just insert the child file inside the main report without running it. Static can be used if you want to include a previously generated report, to make sure FlexCel does not try to run it again.

- **CopyRowsOrCols:** this parameter can be "R", "C" or "RC" (without quotes). By default, included reports will get the column widths and row heights of the parent report (unless you are inserting a full column or row). If you specify "R" here, row heights and format from the included report will be copied to the parent, modifying the parent rows. If you specify "C", column widths will be copied, modifying the parent columns. If you specify "RC", both columns and rows will be copied.

Description

Includes a subreport inside the current one. An included file can include other files itself. The subreport is precompiled and runs on its own sandbox, so it cannot access cells on the parent. For example, a `<#delete range>` tag will never erase something outside the include. The subreport does have access to all report variables, expressions and user-defined functions of the parents, as it has to the parent databases.

Example

On the following include, cells will be moved the whole row down.

	A	B	C	D	E
1					
2		<code><#include(File;Main;__)></code>	B	C	
3		D	E	F	
4					

The generated file will be:

	A	B	C	D	E
1					
2		(cell overwritten)	(cell overwritten)	(cell overwritten)	
3					
4					
5					
6		D	E	F	
7					

Note how cells C2 and D2 are overwritten on the final report. As a general rule, do not write anything at the right of the file being included, except if you know for sure the include won't overwrite the cells. When inserting columns (I_ and I_), cells on the same column as the include will be overwritten.

Configuration Sheet

Syntax

`<#config>`

Description

This tag will identify the current sheet as the configuration sheet. It will only have an effect when written on a sheet name.

Example

Just name the sheet `<#config>`

You can also conditionally define a sheet as the configuration sheet. If you write `<#if(<#value>=1;<#delete sheet>;<#config>)>` as the name of one sheet and `<#if(<#value><>1;<#delete sheet>;<#config>)>` as the name of another, the configuration sheet will be the first or the second one depending on the value of `<#value>`.

Delete Range

Syntax

`<#delete range(range address; shift type)>`

Parameters

- **range address:** The range of the cells to delete. This might be a string like "A1:B5" or a named range like "myrange"

NOTE

Whenever possible, use named ranges instead of strings in the range definitions. If you define a named range "myrange" in cells A2:A3 and use it as a parameter for this tag, when you insert a row in A1 the range will move to A3:A4. If you had written the string "A2:A3" as the parameter for this tag, it would still point to A2:A3 after inserting the row.

- **shift type:** How the existing cells will be shifted when deleting. There are four possibilities: `"_"`, `"_"`, `"l_"` and `"ll_"`, to move existing cells the full row up, only up, full column left and only left respectively.

Description

Use it to delete a range of cells. The range will be deleted after all the cells on the band have been replaced.

Example

`<#if(<#value>=1;<#delete range(a1:a5;_)>;)>` will delete the first five rows on the band when `<#value>=1` and shift rows up.

`<#if(<#value>=1;<#delete range(myrange;l_)>;)>` will delete the named range **myrange** when `<#value>=1` and shift columns to the left.

Delete Row

Syntax

1. `<#delete row>`
2. `<#delete row(full)>`
3. `<#delete row(relative)>`

Parameters

- **no parameters:** When called with no parameters, the full row will be deleted.
- **full:** This is the same as calling it without parameters, the full row will be deleted
- **relative:** Only the cells inside the range being processed will be deleted, not the full row. Use this call if for example you have 2 side by side reports, and wish to delete the row in one of the reports but not in the other. Older FlexCel versions used this mode by default

Description

Use it to delete the current row. If you are a FlexCel 2.x user, note that `<#delete row>` behaves differently from the old `...delete row...` Now `<#delete row>` is processed at the same time as the ranges, so you can't use it to expand ranges. Use "X" ranges instead.

Example

`<#if(<#value>=""<#delete row>)>` will delete the current row when value is empty.

Delete Column

Syntax

`<#delete column>`

Parameters

- **no parameters:** When called with no parameters, the full column will be deleted.
- **full:** This is the same as calling it without parameters, the full column will be deleted
- **relative:** Only the cells inside the range being processed will be deleted, not the full column. Use this call if for example you have 2 reports one above the other, and wish to delete the column in one of the reports but not in the other. Older FlexCel versions used this mode by default

Description

Use it to delete the current column.

Example

`<#if(<#value>=""<#delete column>)>` will delete the current column when value is empty.

Delete Sheet

Syntax

`<#delete sheet>`

Description

This tag will delete the current sheet. It will only have an effect when written on a sheet name.

Example

If you write `<#if(<#value>=1;<#delete sheet>;Food)>` as the name of a sheet, this sheet will have the name **Food** when the report variable **value** is not one, and will be deleted when **value=1**

NOTE

As the maximum sheet name size is 31 characters, you will probably need to write the expression including the `<#delete sheet>` tag on the config sheet, and name the sheet as `<#=(<#Config>!A1)>` or similar, as shown in the picture:



Sheet Visible

Syntax

1. `<#sheet visible(show)>`
2. `<#sheet visible(hide)>`
3. `<#sheet visible(very hide)>`

Description

This tag will hide or show the current sheet. It can be written in any cell. The very hide option corresponds with [Excel's very hidden](#) sheets.

Example

If you write in a cell `<#if(<#value>=1;<#sheet visible(very hide)>)>` then the sheet will be hidden when value is 1.

NOTE

If you have multiple `<#sheet visible>` tags in a file, then it is not determined which one will be last used and have the final effect. It depends on the order of evaluation of the cells inside the sheet. Try to have a single tag per sheet, and not inside any range that could be copied.

Format Cell

Syntax

```
<#format cell(format name; [user format parameter 1; user format parameter 2;...])>
```

Parameters

- **format name:** The name of a format defined on the config sheet, or a User-defined Format defined with [FlexCelReport.SetUserFormat](#).
- **user format parameters:** This is a list of **optional** parameters and they are only used if *format name* is a User-defined format. It is a list of parameters that will be passed to the user-defined format function.

Description

Use it to format a cell with a defined format. You define all format settings (fonts, borders, patterns, etc) on the config sheet, and then you can freely use them. Note that you can define "partial formats" that will only apply part of the format. (for example the cell background, but keeping the font of the destination cell). Look at the [Reports Designer Guide](#) for more information on partial formats

Examples

```
<#if(mod(row(a1),2)=0;<#format cell(Yellow)>;)>
```

 will format the cell as yellow for odd rows.

You need to define a "Yellow" format on the config sheet:

E	F	G	H	I	J	K	L	M
			Format					Expressions
			Format Name	Format Def				Name
			Yellow(Background)					

NOTE

In this example we defined a **Yellow(Background)** partial format, so it only applies the background color of the cell and not other attributes like the font size or color.

You could also use a User-defined Format added with [FlexCelReport.SetUserFormat](#). For example: `<#format cell(MyUserFunction;<#category.color>)>` will call a function registered as `MyUserFunction` with [FlexCelReport.SetUserFormat](#) and pass the value of `<#category.color>` to the function. The function will return the format for the cell.

Format Row

Syntax

```
<#format row(format name; [user format parameter 1; user format parameter 2;...])>
```

Parameters

- **format name:** The name of a format defined on the config sheet, or a User-defined Format defined with [FlexCelReport.SetUserFormat](#).
- **user format parameters:** This is a list of **optional** parameters and they are only used if *format name* is a User-defined format. It is a list of parameters that will be passed to the user-defined format function.

Description

Use format row to format the current row with a defined format. Note that the order on that the `<#format>` tags will be applied is:

1. format row / format col
2. format range
3. format cells

So, if you format row 1 as "Red", and cell A1 as "Blue", the cell format will have priority over the row format and A1 will be Blue. B1:XFD1 will be Red.

Example

```
<#format row(blue)>
```

 will format the current row with the user-defined format "blue".

Format Column

Syntax

```
<#format column(format name; [user format parameter 1; user format parameter 2;...])>
```

Parameters

- **format name:** The name of a format defined on the config sheet, or a User-defined format defined with [FlexCelReport.SetUserFormat](#).

- **user format parameters:** This is a list of **optional** parameters and they are only used if *format name* is a User-defined format. It is a list of parameters that will be passed to the user-defined format function.

Description

Use format column to format the current column with a defined format. Note that the order on that the `<#format>` tags will be applied is:

1. format row / format column
2. format range
3. format cells

So, if you format column A as "Red", and cell A1 as "Blue", the cell format will have priority over the column format and A1 will be Blue. A2:A1048576 will be Red.

Example

`<#format column(blue)>` will format the current column with the user-defined format "blue".

Format Range

Syntax

`<#format range(range address; format name; [user format parameter 1; user format parameter 2;...])>`

Parameters

- **range address:** The range of the cells to format. This might be a string like "A1:B5" or a named range like "myrange"

NOTE

Whenever possible, use named ranges instead of strings in the range definitions. If you define a named range "myrange" in cells A2:A3 and use it as a parameter for this tag, when you insert a row in A1 the range will move to A3:A4. If you had written the string "A2:A3" as the parameter for this tag, it would still point to A2:A3 after inserting the row.

- **format name:** The name of a format defined on the config sheet, or a User-defined format defined with [FlexCelReport.SetUserFormat](#).
- **user format parameters:** This is a list of **optional** parameters and they are only used if *format name* is a User-defined format. It is a list of parameters that will be passed to the user-defined format function.

Description

Use format range to format a range of cells with a defined format. Note that the order on that the `<#format>` tags will be applied is:

1. format row / format col
2. format range
3. format cells

So, if you format range A1:B2 as "Red", and cell A1 as "Blue", the cell format will have priority over the range format and A1 will be Blue. All other cells on A1:B2 will be Red.

Example

`<#format range(a1:b2;blue)>` will format the range a1:b2 with the user-defined format "blue".

`<#format range(myrange;blue)>` will format the named range "myrange" with the user-defined format "blue".

Merge Range

Syntax

`<#merge range(range address)>`

Parameters

- **range address:** The range of the cells to merge. This might be a string like "A1:B5" or a named range like "myrange"

NOTE

Whenever possible, use named ranges instead of strings in the range definitions. If you define a named range "myrange" in cells A2:A3 and use it as a parameter for this tag, when you insert a row in A1 the range will move to A3:A4. If you had written the string "A2:A3" as the parameter for this tag, it would still point to A2:A3 after inserting the row.

Description

Use merge range to dynamically merge a range of cells when generating the report. The range will grow/shrink when copying the tag, depending on the count of records on the current band.

IMPORTANT

This tag is only for when you need to merge cells depending on a condition. For normal merged cells, just merge them in the template.

Example

`<#merge range(a1:a2)>` when written inside a band on A1:Z2 will merge the cells on column A once per band.

Row Page Break

Syntax

<#page break>

Description

<#page break> tags are useful for inserting page breaks on the report. "Normal" page breaks are fixed, they won't be copied each time a band is expanded. So you need to add this tag to get a page break for each value of the band. Note that manual page breaks on a sheet have a maximum of 1026, so any page break above this will not be inserted and will be silently ignored. There is a property [FlexCelReport.ErrorActions](#) that you can set to throw an exception instead on ignoring them when the limit is reached.

Example

<#page break> will insert a page break on the current row.

Column Page Break

Syntax

<#column page break>

Description

<#column page break> tags are useful for inserting page breaks on the report. "Normal" page breaks are fixed, they won't be copied each time a band is expanded. So you need to add this tag to get a page break for each value of the band. Note that manual page breaks on a sheet have a maximum of 1026, so any page break above this limit will not be inserted and will be silently ignored. There is a property [FlexCelReport.ErrorActions](#) that you can set to throw an exception instead on ignoring them when the limit is reached.

Example

<#column page break> will insert a page break on the current column.

Automatic Page Breaks

Syntax

1. <#auto page breaks>
2. <#auto page breaks(PercentOfPageUsed; PageScale)>

Parameters

- **PercentOfPageUsed:** This value must be between 0 and 100 and specifies the minimum percent of the sheet that can be empty when adding the page breaks.
- **PageScale:** This parameter must be between 50 and 100, and it specifies how smaller to consider the sheet when calculating the page break, in order to avoid rounding errors.

Calling this tag without parameters is equivalent to calling `<#auto page breaks(20;95)>`

Description

When you write an `<#auto page breaks>` tag in a sheet, FlexCel will try to keep together all named ranges starting with "keeprows_" and "keepcolumns_". For an in-depth explanation on how this works, take a look at the [Reports Designer Guide](#) and [API Developer Guide](#)

Example

`<#auto page breaks>` will tell FlexCel to add manual page breaks in all the sheet so all "keep..." ranges are kept together.

Row Height

Syntax

1. `<#Row Height(Value)>`
2. `<#Row Height(show)>`
3. `<#Row Height(hide)>`
4. `<#Row Height(autofit; Adjustment;AdjustmentFixed;MinHeight;MaxHeight)>`

Parameters

- **Value:** If it is a number, it means the height of the row. If it is "show" or "hide" means to show or hide the row. If it is "autofit", it means to autofit the row to the cell contents.
- **Adjustment:** This value is optional and only has meaning if "value" is autofit. It is a percent to make the row higher.
- **AdjustmentFixed:** This value is optional and only has meaning if "value" is autofit. It is a fixed amount to make the row bigger than the calculated value. The final height of the row might be calculated as: $FinalHeight = CalculatedHeight * Adjustment + AdjustmentFixed$
- **MinHeight:** This value is optional and only has meaning if **value** is autofit. It might be:
 - Dont Shrink: Means autofit, but never make the row smaller than the original size.
 - Dont Grow: Means autofit, but never make the row bigger than the original size.
 - A number: Specifies the minimum size of the row.
- **MaxHeight:** This value is optional and only has meaning if **value** is autofit. It might be:
 - Dont Shrink: Means autofit, but never make the row smaller than the original size.

- Dont Grow: Means autofit, but never make the row bigger than the original size.
- A number: Specifies the maximum size of the row.

Description

Use this tag to change the heights of rows. See the [Autofit](#) demo for more information.

Example

`<#Row Height(30)>` will set the row height to 30.

`<#Row Height(hide)>` will hide the row.

`<#Row Height(Autofit)>` will mark the row to be autofitted by FlexCel.

`<#Row Height(Autofit;100;0;dont shrink)>` will mark the row to be autofitted by FlexCel, with standard adjustment, and with a row size of at least the original row size.

Column Width

Syntax

1. `<#Column Width(Value)>`
2. `<#Column Width(show)>`
3. `<#Column Width(hide)>`
4. `<#Column Width(autofit; Adjustment;AdjustmentFixed;MinWidth;MaxWidth)>`

Parameters

- **Value:** If it is a number, it means the width of the column. If it is "show" or "hide" means to show or hide the column. If it is "autofit", it means to autofit the column to the cell contents.
- **Adjustment:** This value is optional and only has meaning if "value" is autofit. It is a percent to make the column wider.
- **AdjustmentFixed:** This value is optional and only has meaning if "value" is autofit. It is a fixed amount to make the column bigger than the calculated value. The final width of the column will be calculated as: $FinalWidth = CalculatedWidth * Adjustment + AdjustmentFixed$
- **MinWidth:** This value is optional and only has meaning if **value** is autofit. It might be:
 - Dont Shrink: Means autofit, but never make the column smaller than the original size.
 - Dont Grow: Means autofit, but never make the column bigger than the original size.
 - A number: Specifies the minimum size of the column.

- **MaxWidth**: This value is optional and only has meaning if **value** is autofit. It might be:
 - Dont Shrink: Means autofit, but never make the column smaller than the original size.
 - Dont Grow: Means autofit, but never make the column bigger than the original size.
 - A number: Specifies the maximum size of the column.

Description

Use this tag to change the widths of columns. See the [Autofit](#) demo for more information.

Example

`<#Column Width(Autofit)>` will mark the column to be autofitted by FlexCel.

Autofit Settings

Syntax

`<#Autofit Settings(Global; KeepAutofit; Adjustment; AdjustmentFixed; MergedCellsMode)>`

Parameters

- **Global**: It can be either the string **All** or **Selected** (without quotes). **All** means automatically autofit every row on the workbook, regardless of if the row has been marked for autofit (with `<#row height(autofit)>`) or not. **Selected** means only autofit rows that are marked. The default is **Selected** and we recommend this setting, since autofitting all the rows on the sheet could change heights of rows you do not want to change. If you want to use **All**, make sure you make rows that you do not want to change of fixed height on the Excel template.
- **KeepAutofit**: Might be **keep** or **fixed**. If **keep**, rows that were marked as autofit on the original template will be kept autofit. This means when you open the file in Excel it will recalculate the row heights and they might change a little, but you will never get cropped text. The default is true. If this setting is **fixed**, row height will be fixed at the size calculated by FlexCel, and Excel will not recalculate them. While this will make both Excel and FlexCel look the same, when seeing the file in Excel it might crop some text. If you want to use this option, we recommend you set **Adjustment** of about 150 to avoid text crop.
- **Adjustment**: It is a percent to make the columns wider or rows higher on all the sheets. The default is 100, but you can enter a bigger number here. Also, you can override global adjustments with the `<#row height(autofit, localadjustment)>` and `<column width(autofit, localadjustment)>`. If you do not specify localadjustment on those tags, the value specified here will be used.
- **AdjustmentFixed**: It is a fixed amount to make the columns wider or rows higher on all the sheets. The default is 0, but you can enter a bigger number here.

- **MergedCellsMode**: Sets how to fit a cell that spans over multiple rows when autofitting rows, or a cell that spans multiple columns when autofitting a column. The possible values for this parameter are **None**, **First**, **Last** and **Balanced** (without quotes).

If you omit this parameter, **Last** will be assumed.

- **None** means that the cell will be ignored when autofitting.
- **First** means that the first row/column of the merged cell will be changed to autofit the full cell.
- **Last** means that it will change the last row/column of the cell.
- **Balanced** means that it will increase the height of all the rows in the merged cell by the same amount.

First might be followed by a "+" and a number between 0 and 4. For example, "First+1" means use the second row of the merged cell to do the autofit. Similarly, "Last" might be followed by a "-" and a number between 0 and 4. For example "Last-2" means use the row that is 2 rows before the last.

Description

This tag commands the autofit settings on a sheet. You need to have only one of those tags in each sheet, and it will affect the autofit of all rows and columns. If you do not specify this tag on a sheet, the default used is `<#Autofit Settings(Selected;keep;100;0)>`

Example

`<#Autofit Settings(All, keep, 100)>` will autofit all non-fixed rows on the sheet, and you will not need to specify individual rows to autofit with `<#row height>` tag.

`<#Autofit Settings(Selected; Fixed; ; ;Last-1)>` will autofit only rows that have an autofit tag, and keep the size fixed. The adjustments are left to the default, and in case of merged cells, the row or column before the last in the merged cell will be used. Note that when autofitting merged cells, you might want to keep autofit "Fixed", because Excel doesn't autofit merged cells, so when you open the file, it will revert to a single line.

Comment

Syntax

`<#//(...)>`

Description

Everything inside a // tag will be ignored. You can use it to temporarily disable tags.

Example

`<#//(This is a comment)>` will not do anything and is equivalent to an empty string.

Image Size

Syntax

1. `<#imgsize>`
2. `<#imgsize(zoom; aspect ratio)>`

Parameters

When called with no parameters, this tag will resize the image to "Best Fit" inside the original image template rectangle maintaining the aspect ratio. That is, if the image in the template is 50px wide x 40px tall, the new image will be resized to be either 40 px tall or 50 px wide, in a way the aspect ratio is maintained and the new image is no bigger than the image in the template.

- **zoom**: Percent of zoom to resize the image. 0 means leave size untouched.
- **aspect ratio**: Aspect ratio of the image. 0 means "leave size untouched". Negative values mean "keep height fixed and resize the width to match", and positive values mean "keep width fixed and resize the height to match".

Description

This tag will only work when written on the name of an image. You shouldn't use both parameters at the same time, always leave zoom = 0 or aspect ratio = 0.

Example

The most common way to use this tag is just to name an image `<#Data><#imgsize>`. If you do so, the inserted image will be as big as possible without being bigger than the original, and maintaining the aspect ratio. If you name an image `<#Data><#imgsize(0;-1)>` the image will retain its designed height and resize its width so it is not distorted. You can see a lot of different uses of this tag on the [Images](#) demo.

Image Position

Syntax

`<#imgpos(RowAlign;ColAlign;RowOffset;ColOffset)>`

Parameters

All parameters are optional.

- **RowAlign**: It might be "Top", "Center" "Bottom" or omitted. If omitted vertical image position won't change.
- **ColAlign**: It might be "Left", "Center" or "Right" or omitted. If omitted the horizontal image position won't change.

- **RowOffset:** It is a number specifying how many pixels from the calculated position the image will be moved down. If negative, image will be moved up from the calculated position. If omitted it is assumed to be 0.
- **ColOffset:** It is a number specifying how many pixels from the calculated position the image will be moved right. If negative, image will be moved left from the calculated position. If omitted it is assumed to be 0.

Description

This tag must be written as part of an image name, not in a cell. Use this tag to dynamically move an image. You will normally need to use it when dealing with images of different sizes.

Example

If you name an image `<#Data><#imgpos(center:center;-10)>` the image will be centered in the column, and 10 pixels to the right of being centered in the row.

Image Fit

Syntax

`<#imgfit(FitInRows;FitInCols;RowMargin;ColMargin)>`

Parameters

All parameters are optional.

- **FitInRows:** It might be **"InRow"**, **"Dont Shrink"**, **"Dont Grow"** (all without quotes) or omitted. If omitted the row will not change. If you use **InRow**, the row size will always change to fit the image. **Dont Shrink** and **Dont Grow** work the same as **InRow**, but row size will only change if the new height is larger/smaller than the current size.
- **FitInCols:** It might be **"InColumn"**, **"Dont Shrink"**, **"Dont Grow"** or omitted. If omitted the column will not change. If you use **InCol**, the column size will always change to fit the image. **Dont Shrink** and **Dont Grow** work the same as **InColumn**, but column size will only change if the new width is larger/smaller than the current size.
- **RowMargin:** It is a number specifying how many pixels to add to the row as a margin around the image.
- **ColMargin:** It is a number specifying how many pixels to add to the column as a margin around the image.

Description

This tag must be written as part of an image name, not in a cell. Use this tag to resize a row or a column so they are big enough to hold an image. You will typically want to use this tag when image size changes dynamically, so the images fit inside their cells.

Example

If you name an image `<#Data><#imgfit(inrow;;-10)>` the row will be made as big as the image plus 10 pixels.

Image Delete

Syntax

`<#imgdelete>`

Description

This tag must be written as part of an image name, not in a cell. Use this tag to delete an image.

Example

If you name an image `<#Data><#if(<#Data=""><#imgdelete>;)>` image will be deleted when there is no data.

Lookup

Syntax

`<#lookup(table name; search key names; search key values ;result field)>`

Parameters

- **table name:** Master table where we will look for the value.
- **search key names:** A list of columns containing the search key on the master table. It will normally be just one column, but if you need to search by more than one, you can separate column names with a comma (",")
- **search key values:** A list of values containing the search values on the master table. The number of search key values should match the number of search key names. If you have more than one search key value, you need to use an `<#array>` tag.
- **result field:** the field of "Table name" you want to display.

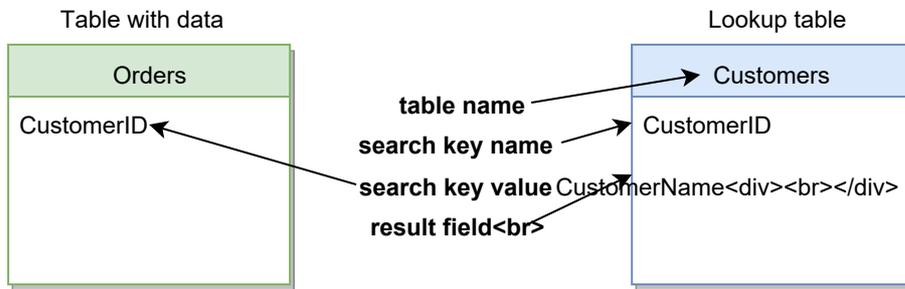
Description

Use `<#lookup>` to search for a field description on another table.

Example

If you keep an CustomerId on table Orders and the Customer data on a table Customers, to output the real customer name for an order you can use:

`<#lookup(Customers;CustomerId;<#Orders.CustomerId>;CustomerName)>`



For more examples on the use of lookup, see the [Lookups](#) demo.

Array

Syntax

`<#array(value_1; value_2;... ;value_n)>`

Description

Use `<#array>` to output an array of values. Currently, the only use of `<#array>` tag is to provide an array of search keys for the `<#lookup>` tag, but it could have more independent uses on the future.

Example

To lookup a field by two different keys, you should use:

`<#lookup(Table;Column1,Column2;<#array(value1;value2)>;result column)>`

Regular Expressions

Syntax

`<#regex(IgnoreCase; Expression; Match; [Replace])>`

Parameters

- **IgnoreCase:** 0 to do a case-sensitive search, 1 to do a case-insensitive search.
- **Expression:** Regular Expression we want to evaluate.
- **Match:** String where we will apply the regular expression.
- **Replace:** This is an optional parameter, and if present, it is the string that we will replace into the matching parts of Match.

Description

There are 2 ways to use this tag, depending if you include the **Replace** parameter or not. If you don't include it, this function will return the parts of the string **Match** that match the regular expression. When you specify a Replace parameter, this function will return the original "Match" string with the parts that match the expression replaced by the **Replace** parameter.

Examples

`<#regex(0;x.*e;flexcel)>` will return xce

`<#regex(0;x.*e;flexcel;***)>` will return fle***l

`<#regex(0;x*.e;flexcel;o)>` will return fool

`<#regex(0;x*.e;flexcel;)>` will return fl

Formula

Syntax

`<#Formula>`

Description

You can use this tag to make FlexCel enter the text on the cell as a formula instead of a string. Note that the text on the cell must be a valid formula, and start with an "=" sign. If the expression is not a valid Excel formula, an Exception will be raised. You only need to enter this tag once in a cell, generally at the beginning. Note that for entering cell references, you will need to use the `<#ref>` tag.

Example

If you enter on a cell:

```
B5: <#Formula>= <#ref(0;-1)> + <#Db.Field>
```

when the report is generated, on the cell you will have formulas like:

```
B5: "=A5 + 4"  
B6: "=A6 + 3"  
etc.
```

We used the `<#ref>` tag here to make the reference "A5" grow down when the cell is copied. Also, using `<#ref>` instead of writing the cell reference directly, allows you to insert for example a row at the beginning of the template, and not break the report.

WARNING

In the real world, there is little use for the `<#formula>` tag, to the point that it is difficult to find examples where it can be useful. Of course those cases exist, but if you are thinking about using this tag, first think if a simple Excel formula without any tags wouldn't be better.

In the example above, you could write `<#Db.Field>` in column C, and have a simple `= A5 + C5` formula in B5. Whenever possible, prefer simple Excel formulas instead of using this tag.

Ref

Syntax

1. `<#Ref(NamedRange)>`
2. `<#Ref(RowOffset; ColOffset)>`
3. `<#Ref(NamedRange; RowAbsolute; ColAbsolute)>`

Parameters

- **NamedRange:** The name of a named range with the cell address you want to use.
- **RowOffset:** How many rows below or above this cell is the reference. Use negative values to indicate rows above the cell.
- **ColOffset:** How many columns at the left or the right of this cell is the reference. Use negative values to indicate rows at the left of the cells.
- **RowAbsolute:** If true, the row will not move down when copying. This is analog to a `A$1` reference.
- **ColAbsolute:** If true, the column will not move to the right when copying. This is analog to a `$A1` reference.

Description

This tag will normally be used together with a `<#formula>` tag, in order to add relative references to a hand-written formula. Even if the values are absolute, it is a good idea to always use `<#ref>` tags on formulas, since if you don't, whenever you insert rows on the sheet the references will not be updated

Example

`<#ref(-1;-2)>` means the cell that is 1 row above and 2 columns to the left

`<#ref(Potatoes>true>true)>` means a reference to the name "potatoes" on the sheet that will not move when copying cells.

HTML

Syntax

<#HTML(Enable)>

Parameters

- **Enable:** Enable can be "TRUE" or "FALSE". When true, the text on the cell will be entered as HTML, when false it will be entered as normal text. For more information about HTML tags supported, see the [FlexCelReport.HtmlMode](#) property.

Description

This tag overrides the global property [FlexCelReport.HtmlMode](#) on a cell by cell basis. If you set [HtmlMode](#) = true on a report, you can exclude individual cells of being HTML formatted with the tag <#HTML(false)>. Similarly, when [HtmlMode](#) = false, you can make individual cells HTML formatted with the tag <#HTML(true)>. You only need to write one HTML tag into a cell, and its position does not matter.

Example

<#HTML(true)><#Text> will enter the value of <#Text> as an Html string when [HtmlMode](#) = false.

Preprocess

Syntax

<#Preprocess>

Description

The preprocess tag enters a "preprocessor" mode where you can modify the template before actually running the report. You only need to write one Preprocess tag into a cell, and its position does not matter.

When this tag is present in any cell of the template, FlexCel will make 2 passes on it. On the first pass, FlexCel will process all the cells with "Preprocess" tag, and in the second it will load the modified template. You can use the first pass to delete rows and columns, and customize the final template before the report.

You can get dynamic templates this way, that are customized depending on the data.

Example

`<#Preprocess> <#if(<#defined(customer.date)>);;<#delete column>>` will delete the column from the template before running the report when **customer.date** is no defined.

`<#Preprocess> <#if(<#includecustomer>);;<#delete column>>` will delete the column if the variable **includecustomer** is false

Defined

Syntax

1. `<#defined(field_or_variable)>`
2. `<#defined(field_or_variable;global)>`
3. `<#defined(field_or_variable;local)>`

Parameters

- **field_or_variable**: Field we want to find out if it is defined. "Defined" will return if the variable or database field exists in a global scope.
- **global**: This is the same as calling it with just 1 parameter.
- **local**: When the second parameter is the string "local", "defined" will return true only if the field is accessible to the current range. For example, if you had a master range `__master__` and included inside a detail range `__detail__`; `defined(detail.field;local)` would return true only if the cell was inside the `__detail__` range, but not if it was inside the `__master__` range. `defined(detail.field;global)` or simply `defined(detail.field)` will return true no matter the cell where the expression is in.

Description

Use this tag to know if a field variable is defined or not. This is normally useful when doing metatemplates (see [meta templates](#) demo) together with the Preprocess tag. This way you can have dynamic SQLs, and delete columns from the report if those columns were not selected in the SQL.

Example

`<#Preprocess> <#if(<#defined(customer.date)>);;<#delete column>>` will delete the column from the template if the field "date" does not exist in the table customers

NOTE

When using the "defined" tag, you will probably need to use default values in the database fields too. For example, the expression "**<#if(<#defined(db.field)>;<#db.field>;no data)>**" will raise an error if db.field does not exist. This is because FlexCel precompiles the whole expression before evaluating it, and it can't compile it if <#db.field> does not exist. The defined tag will be evaluated later, (many times, this is why FlexCel precompiles the expression), but at precompile time this expression will raise an error.

The correct expression in this case is "**<#if(<#defined(db.field)>;<#db.field>;no data)>;no data)>**", or more simple just "**<#db.field>;no data)>**"

In general, when using fields that might be defined or not, you should always specify a default value for them.

Defined Format

Syntax

<#defined format(expression)>

Parameters

- **expression**: An expression that should resolve to a string

Description

Use this tag to know if a custom format is defined in the config sheet.

Example

<#if(<#defined format(<#fmt>>;<#format cell(<#fmt>>);)> will format the cell with the style specified by the variable **<#fmt>** if it is defined, or do nothing otherwise.

Swap Series

Syntax

<#Swap Series>

Description

The Swap Series tag has no parameters and has only effect in two places:

- If you write it as a part of the name of a chart.
- If you write it as a part of the sheet name in a chart sheet.

In both cases you can write the tag anywhere, the position doesn't matter.

This tag allows you to swap the column and rows in a chart, allowing you to create charts that have a series for each row of data. This tag runs after all the report has been generated. See [Creating charts with dynamic series](#) for more information on how the tag works.

Example

If you name a chart sheet as "**Product Chart<#swap series>**" then the chart sheet will swap rows and columns after the report is generated. Same applies if you name a chart object "**Product Chart<#swap series>**". Note that the final name of the object or the chart sheet will be "Product Chart" as the <#swap series> part will be removed from the name.

FlexCel Performance Guide

Introduction

Performance is a complex but very important characteristic of any application. One of our design goals with FlexCel is to be as fast as possible, and we are always finding ways to improve a little the performance from version to version. The idea is that for most of the cases, FlexCel should be “fast enough” and you shouldn’t have to care about performance when using it, as we took care of it for you. But for cases where you need the absolute maximum of performance, you can help FlexCel perform better by writing code that uses it in an optimal way. To know how to code for maximum performance, you need to understand how FlexCel works from a performance viewpoint, and that’s what this document is about.

WARNING

Before doing anything else, let us make this clear: **Don’t over optimize**. In many cases code clarity and performance are not compatible goals and the more performing the code, the more difficult it is to maintain, change, or adapt. In most cases FlexCel is incredibly fast, and you shouldn’t have to depend on dirty tricks to create or read files instantly.

Memory

If there is one thing that can make your applications slower, that thing is using too much memory and starting paging to disk. And sadly, because of the way Excel files are created, and also because of how a spreadsheet works, FlexCel needs to keep the full spreadsheet loaded in memory.

Even when in many cases we use Excel files as databases they aren’t. It is impossible to randomly read or write cells from an xls/xlsx file, and due to the way formulas work, a single change in a cell might affect the full file. For example, imagine that we have a spreadsheet with cell A1 = 1, and in the second sheet, Sheet2!E4 has the formula “=Sheet1!A1 + 1” When we change the value at cell A1 from 1 to 2 we need to change the formula result in Sheet2!E4 from 2 to 3. Also, if we insert a row before A1, we will need to change the cell Sheet2!E4 from “= Sheet1!A1 + 1” to “=Sheet1!A2 + 1”. This makes it very difficult to work with just a part of the file in memory, and neither we nor Excel do it. We both load the full file into memory and do all of our work there.

The main issue with running out of memory is that performance degradation is not linear. That is, you might use 1 second to write a million cells, but 1 minute to write 2 million, and 1 hour to write 3 million. This is one of the reasons we don’t normally quote silly benchmark numbers like “n cells written per second”; they are meaningless. If you take n seconds to write m cells, it doesn’t mean that to write 2*m cells you will need 2*n seconds. Performance degrades exponentially when you run out of memory.

So we make a lot of effort to try to not use too much memory, for example, repeated strings in cells will be stored only once in memory, or cell styles will be shared between many cells. But no matter what, we are a fully managed .NET library, and we can’t escape the limitations of the

platform. .NET applications do use a lot of memory, and the Garbage Collector might not release everything you need either. This is not necessarily a bad thing, using the memory you have is good not bad, but for huge files it can become problematic.

There are 2 ways you can get around the memory problem: using FlexCel in "Virtual Mode", or going to 64-bits. We will expand on this in the sections below.

WARNING

Before continuing, we would like to give one last warning about memory.

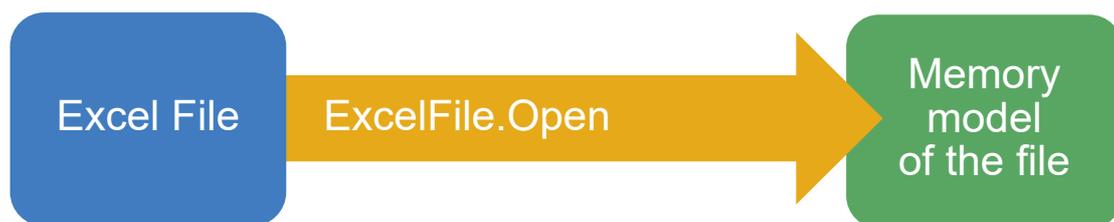
Measuring memory consumption can be very complex, especially in .NET. You need to measure how many objects you have in Generations 0, 1 and 2, you need to distinguish between private and public bytes, and you need a memory measurement tool. Task manager isn't such tool.

Virtual Mode

Sometimes you don't really need all the complexity that a spreadsheet allows; you just want to read values from cells or dump a big database into a file. In those cases, it might not make sense to have the full file into memory, you could read a cell value, load it in your application, and discard the value as the file is being loaded. There is no need to wait until the entire file has been read to discard all the cells.

For this, FlexCel provides what we know as "Virtual Mode", a special mode where cells are read on demand. Not all features are supported in this mode, but normally for the huge files where Virtual mode makes sense, those features aren't needed either.

The normal way you open a file is as follows:



You call `ExcelFile.Open`, and FlexCel reads the file and loads it into memory.

In virtual mode, it works as follows:



You call `ExcelFile.Open`, and the file is read. But for every cell read, the `VirtualCellRead` event is called, and the cell is not loaded into the memory model. You will end up with an `ExcelFile` object that has no cells (but has charts, drawings, comments, etc.). Cells can be used by your application as they are being read. You can get the rest of the things (comments, merged cells, etc.) from the memory model after the file has been opened.

There is a simple example on how to use virtual mode in the [ExcelFile.VirtualMode](#) documentation. A more complete example is at the [Virtual Mode](#) api demo.

64-bits

Sometimes, if you have a memory problem and the application runs server-side, the cheapest solution might be to install more memory in the server. But in 32-bit mode, no matter how much memory you have, FlexCel will never use more than 2 Gb.

If you want to use more memory than that you should use FlexCel in 64-bit mode. Note that FlexCel.dll is compiled as "Any CPU", and that means that it will work in 32-bit mode if your application is compiled as a 32-bit application and 64-bit mode if it is compiled as 64-bit. You don't need a different assembly.

But we should go into warning mode again:

WARNING

Going 64-bits might not only not improve performance, it might decrease it. In our tests in a 4Gb machine, 64-bit version is consistently slower than the 32-bit one (both versions running in a 64-bit Operating System).

It kind of makes sense, since by going 64-bits now every pointer is 8 bytes instead of 4, so there is a lot of memory used in those bigger pointers, that are mostly 0 anyway. For 64-bit to improve things, you need a lot of memory installed.

By the way, .NET 4.0 is much better than 3.5 in our tests for 64-bits, getting almost the speed of 32-bits, while in 3.5 64-bit is really bad. So if you are thinking about going 64-bit, consider at least .NET 4.0.

Server side

One of the most common uses for FlexCel is server-side. You install an app using FlexCel on a server, preferably with lots of cores, and people access it through a web page or web service.

If your CPU has 32 cores and life was perfect, opening 32 files in 32 parallel threads would take the same time as opening one file. But, life isn't perfect, and you might see noticeable performance degradation when opening many files in parallel.

FlexCel itself doesn't have any locks or shared memory when opening files, so all threads should be independent in theory. But still, you will see the degradation as the number of threads increases. There are multiple reasons for this: One is that the server might have 32 cores, but it still has one memory and one SSD, and all threads have to compete for memory. Also, the CPU might throttle the speed when using too many cores to keep it thermally safe.

But once again, the main problem is memory. Or, more precisely, memory allocations. The problem is worse when opening files because, in that stage, FlexCel allocates a lot of memory as it reads and creates an in-memory model of the file.

The garbage collector for .NET apps is optimized for client-side use. It tries to minimize latency, not the total time used. For client applications, this is what you want, but for a server, configuring the GC to be Server-GC can improve the throughput a lot.

You can get information on Server-GC here: <https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/workstation-server-gc> <https://devblogs.microsoft.com/premier-developer/understanding-different-gc-modes-with-concurrency-visualizer/>

To make your application use Server-GC, you can add the following lines to the csproj:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

then you can add the following line to your app to verify that you are really using Server-GC (it can be tricky to get it working):

```
Console.WriteLine(GCSettings.IsServerGC);
```

Server-GC can make a big difference in a server serving lots of requests in parallel.

NOTE

Another way to attack the multithreading problems is to use multiple processes, one per user, instead of multiple threads. If you chose this option, make sure to use workstation-GC, since there will be a single thread in each process.

Loops

Don't fear loops

From time to time we get emails from people migrating from OLE Automation solutions, asking us for a way to load all values into an array, and then write the full array into FlexCel in one single method call.

This is because in OLE Automation, one of the “performance tips” is to do exactly that. Set the cell values into an array, and then copy that array into Excel. But this is because **in OLE Automation method calls are very expensive**.

And this concept is worth expanding: Excel itself isn't slow; it is coded in optimized C/C++ by some of the best coders in the world. So how is it possible that third-party libraries written in high-level languages like FlexCel can be so much faster? This is in part because Excel is optimized for interactive use, but more important, because while Excel itself is fast, calling Excel from OLE Automation is not.

So any good OLE Automation programmer knows that to maximize the speed, he needs to minimize the Excel calls. The more you can do in a single call the better, and if you can set a thousand cells in one call that is much better than doing a thousand calls setting each value individually.

But **FlexCel is not OLE Automation and the rules that apply are different**. In our case, if you were to fill an array with values so you can pass them to FlexCel, it would actually be slower. You would use the double of memory since you need to have the cells both in the array and in FlexCel, you would lose time filling the array, and when you finally call the imaginary “ExcelFile.SetRangeOfValues(array)” method it would loop over all the array and cells to copy them from the Array to FlexCel since that is the only way to do it.

So just lopping and filling in FlexCel directly is faster; filling an array before setting the values only would add overhead.

Beware of the ending condition in “for” loops in C#

When you are coding in C#, you need to be aware that the ending condition of the loops is called for every iteration. So in the code:

```
for (int i = 0; i < VeryExpensiveFunction(); i++)
{
    DoSomething();
}
```

VeryExpensiveFunction() is called for every time DoSomething() is called.

This can be from a non-issue when VeryExpensiveFunction isn't too expensive (for example it is List.Count) to a disaster if VeryExpensiveFunction is actually expensive and the iteration runs for millions of times (as it can be the case when using “ExcelFile.ColCount”).

The solutions to this problem normally can be:

1. Cache the value of VeryExpensiveFunction before entering the loop:

```
int Count = VeryExpensiveFunction();
for (int i = 0; i < Count; i++)
{
    DoSomething();
}
```

1. If the order in which the loop executes doesn't matter, you might simply reverse the loop:

```
for (int i = VeryExpensiveFunction() - 1; i >= 0; i--)
{
    DoSomething();
}
```

This way `VeryExpensiveFunction` will be called only once at the start, and the comparison is against "`i >= 0`" which is very fast.

Avoid calling `ColCount`

`ExcelFile.ColCount` is a very slow method in FlexCel, and it is mostly useless. To find out how many columns there are in the whole sheet, FlexCel needs to loop over all rows and find which row has the biggest column. You normally just need to know the columns in the row you are working in, not the maximum column count in the whole sheet.

NOTE

Note that `RowCount` is very fast, the issue is with `ColCount`, because FlexCel stores cells in row collections.

Loop only over existing cells

A spreadsheet can have many empty cells in it, and normally you don't care about them. FlexCel offers methods to retrieve only the cells with some kind of data in them (be it actual values or formats), and ignore cells that have never been modified. Use those methods whenever possible. Look at the example below to see how this can be done.

Evil loop Example

Here we will show an innocent looking example of all the problems studied in this "Loops" section. Even when it might look harmless, it can bring the performance of your application to its knees.

```
XlsFile xls = new XlsFile("somefile.xls");
for (int row = 1; row <= xls.RowCount; row++)
{
    for (int col = 1; col <= xls.ColCount; col++)
    {
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

Let's study this by our rules. First of all, let's see the ending conditions of the loops. As explained, `xls.RowCount` and `xls.ColCount` are called for every iteration in the loop. This is not too bad for `RowCount` since `RowCount` is fast and it is the outer loop, but it is a huge problem for `Xls.ColCount`, which is not only slow as mentioned earlier, but also runs in the inner loop.

So if the file has 50,000 rows and the maximum used column is column 30, you are iterating $50,000 \times 30 = 1,500,000$ times. `xls.RowCount` is called 50,000 times (since it is in the outer loop) and `xls.ColCount` is called 1,500,000 times. And every time `ColCount` is called, FlexCel needs to loop over all the 50,000 rows to find out which row has the biggest column.

Using this example is the surest way to kill performance.

So, how do we improve it? The first way could be, as explained, to cache the results of `RowCount` and `ColCount`:

```
XlsFile xls = new XlsFile("somefile.xls");
int RowCount = xls.RowCount;

for (int row = 1; row <= RowCount; row++)
{
    int ColCount = xls.ColCount;
    for (int col = 1; col <= ColCount; col++)
    {
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

But while this was a huge improvement, it is not enough. By our second rule, `xls.ColCount` is slow, and we are calling it 50,000 times anyway. This is better than calling it 1,500,000 times, but still 50,000 times worse than what it could be. The column count is not going to change while we loop, so we can cache it outside the row loop:

```
XlsFile xls = new XlsFile("somefile.xls");
int RowCount = xls.RowCount;
int ColCount = xls.ColCount;

for (int row = 1; row <= RowCount; row++)
{
    for (int col = 1; col <= ColCount; col++)
    {
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

So, for now we have fixed this code to rules 1) and 2). We cache the end conditions in the loop, and we avoid calling `ColCount` much. (A single call to `ColCount` isn't a problem). But still, this code can be incredibly inefficient, and this is where rule 3) comes to play.

In this example, where our spreadsheet has 50,000 rows and 30 columns, we are looping 1,500,000 times. But do we really have 1,500,000 cells? In most cases, we don't. Remember that **ColCount returns the maximum used column in the sheet**. So, if we have 5 columns, but we also have some helper cells in column 30 (AD), `ColCount` will return 30:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	##	AD
1	Column1	Column2	Column3	Column4	Column5																							Some note!
2	65	389	718	312	144																							
3	203	849	94	614	894																							
4	826	724	640	833	354																							
5	363	791	31	522	117																							
6	971	599	278	456	940																							
7	941	36	327	89	874																							
8	398	197	493	422	386																							

But the loop will run as if every row had 30 columns, looping through millions of empty cells. If as in this example we only have 5 columns except for row 1, then we have $5 \cdot 50,000 + 1$ cells = 250,001 cells. But we are looping 1,500,000 times. Whenever you use ColCount for the limits of the loop, you are doing a square with all the rows multiplied by all the used columns, and this will include a lot of empty cells.

We can do much better:

```
XlsFile xls = new XlsFile("somefile.xls");

int RowCount = xls.RowCount;
for (int row = 1; row <= RowCount; row++)
{
    int ColCountInRow = xls.ColCountInRow(row);
    for (int cIndex = 1; cIndex <= ColCountInRow; cIndex++)
    {
        int col = xls.ColFromIndex(row, cIndex);
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

Note that even when ColCountInRow(r) is fast, we still cache it in the ColCountInRowVariable, as it doesn't make sense to call it in every column loop just because. And as the used columns in every row will be different, we need to call it inside the row loop.

This last version will run millions of ways faster than the first, naïve version.

NOTE

If you want to read a range of cells, not the full file, you could use this code:

```
// Loop at most until the last used row in the sheet.
// If LastRow is for example 1.000.000, but there are only
// 3 used rows, it makes no sense to loop over all the empty rows after row
3.
int LastUsedRow = Math.Min(LastRow, xls.RowCount);

for (int row = FirstRow; row <= LastUsedRow; row++)
{
    int LastCIndex = xls.ColToIndex(row, LastColumn);
    int LastColFromIndex = xls.ColFromIndex(row, LastCIndex);
    if (LastColFromIndex > LastColumn || LastColFromIndex == 0) //
LastColumn does not exist.
    {
        LastCIndex--;
    }
    if (LastCIndex == 0) continue; // This row is empty. Move to the next
row.

    int XF = -1;
    for (int cIndex = xls.ColToIndex(row, FirstColumn); cIndex <=
LastCIndex; cIndex++)
    {
        DoSomething(row, xls.ColFromIndex(row, cIndex), xls.GetCellValueInd
exed(row, cIndex, ref XF));
    }
}
```

Or, if you prefer a one-liner, just call [ExcelFile.LoopOverUsedRange](#) which will do the same as what is in the code above.

Reading Files

Ignore formula texts if possible

By default, when you are reading a cell with "[ExcelFile.GetCellValue](#)" and the cell contains a formula, FlexCel returns a [TFormula](#) class that includes the formula value and text. But returning the formula text isn't too fast, because FlexCel needs to convert the internal representation into text for each cell.

If you don't care about the formula texts ("=A1 + 5"), but only about the formula results ("7"), and you are reading huge files with thousands of formulas, setting:

```
ExcelFile.IgnoreFormulaText = true
```

before starting to read the file can improve the performance a little.

NOTE

This particular tip won't improve performance a lot, and it can make the things more complex if you ever start needing the formula texts. Make sure that the speed increase you get from it is worth it.

Reading CSV files

When reading a CSV (comma separated values) file, FlexCel needs to parse each cell and figure out if it is a number, a date, a boolean, a formula, or maybe just a plain string. If you have millions of cells, this can be a very time-consuming process.

Here we will see some tips to make FlexCel's job easier.

Import the full file as text.

If you don't care about converting strings, you can import everything as text. Importing everything as text is the **fastest way to read a CSV file**, and it might be a good option even if you care about some cell values, but not all of them. You can always read the full file as text and manually convert the cells you care about.

The code needed to import the full file is as follows:

```
ColumnImportType[] ColTypes = new ColumnImportType[FlxConsts.MaxColCount];
for (int i = 0; i < ColTypes.Length; i++)
{
    ColTypes[i] = ColumnImportType.Text;
}

ExcelFile xls = new XlsFile(true); //Create a new file.
xls.Open("csv.csv", TFileFormats.Text, ',', 1, 1, ColTypes, Encoding.UTF8, false); //Import the csv text.
```

And if you later need to convert some cell values to whatever they represent, you can use the code:

```
int XF = -1;
TRichString cellValue = xls.GetCellValue(1, 1) as TRichString;

//Remember that if you know the date formats, you can pass
//them as third parameter to ConvertString.
object value = xls.ConvertString(cellValue, ref XF);
```

NOTE

If you want every cell string converted to the corresponding value, don't use this method! It will be slower than just opening the file normally. You should import everything as plain text only if you don't care about the values being all strings, or care only about some cells.

Explicitly define which date formats your file uses.

One of the slowest parts of CSV parsing is figuring out dates. Dates (and times) can be written in many ways and combinations. So, for example, you might have "08/09/1972" in one cell, and you might have "1972-09-08" in another and maybe "08/09/1972 10:00" in a third. This is not a typical case, and you usually will use a single date format, max two, in a file. But FlexCel still has to check, for every cell it reads, if it is a date, and what date that particular cell it is.

So if you know in advance the date and times formats used in the file, passing those to FlexCel will speed up the process significantly. You can use code as below:

```
//File formats used in the file:
string[] dateFormats = { "yyyy-MM-dd", "yyyy-MM-dd hh:mm", "hh:mm" };

ExcelFile xls = new XlsFile(true); //Create a new file.
xls.Open("csv.csv", TFileFormats.Text, ',', 1, 1, null, dateFormats,
Encoding.UTF8, false); //Import the csv text.
```

Reports

Reports are by design a little slower than the API, because they must read a template before, parse it and compile before filling the cells. So if you need ultimate performance, don't use reports, use the API directly. But remember; this advice is similar to "If you need absolute performance use assembler and not C#". Yes, you will get the most performance in assembler or the API than in C# or Reports. But it might not be worth what you lose in flexibility and ease of use.

Avoid too many master-detail levels

In most cases, the [FlexCelReport](#) overhead is virtually zero, as the code in FlexCel was designed from the start to be very efficient when running reports. But there is a case where you can see visible performance degradation and that is when you have many nested master-detail reports.

The problem here is that FlexCel works by inserting ranges of cells. If you have a template with a single range, it will insert the rows needed for every record in the database at the beginning, and then fill those rows. A single insert of n rows, this is very fast.

But if you have a master-detail report, it will first insert all the rows needed for the master, and then, for every record in the master, it will insert all the rows needed for the corresponding detail. If that detail itself has other detail then the inserts grow a lot.

Normally, for more than 3 levels of master-detail, you shouldn't have very big reports. Most of the times this is ok, since when you have huge reports they are normally just the records of a database. Those will be used for analyzing the data because nobody is going to print or read a million row report anyway. Complex reports designed to be printed or read normally are smaller, and for small reports you don't really need to care about performance.

Decide whether to use LINQ or Datasets as data sources

FlexCel provides two different ways to access the data: `IQueryable<T>` objects and datasets. In theory, `IQueryable` objects can be a little faster because they don't load all the data in memory, fetching it as it is needed. But in practice, there is normally not a real performance difference unless you are really memory bounded. In fact, datasets can be faster than `IQueryable` objects because all the data is just fetched once from the database, and once that is done you don't hit the database anymore. And datasets are extremely efficient at fetching tables from databases.

As always, the best way to know what is faster in your cases might be testing. You might expect LINQ datatables to be a little faster with huge datasets with no master-details, and datasets to be faster in small reports with lots of nested master-detail levels.

Load tables on demand when using Datasets

When using `DataSets` in the common way, you need to know in advance what tables are needed for the report, and load them before the report is run. Something like this:

```
FlexCelReport.AddTable(LoadTable(table1));
FlexCelReport.AddTable(LoadTable(table2));
FlexCelReport.Run();
```

This is ok if you are going to need both `table1` and `table2` in the report, but in more complicated cases where you don't know in advance which tables might be used it can be a waste of resources. You might be loading `table2` from the database, and only using `table1` because the user checked a checkbox in the application for a "short" report that doesn't need `table2`.

If you are using LINQ this isn't a real problem; you can `AddTable()` as many tables as you think you might need, and the data will be fetched from the database only when that data is actually requested. But with datasets, all data is preloaded before running, so all tables will be queried from the database.

To solve this problem, FlexCel offers "on demand" loading of datasets through the `LoadTable` event. Instead of using `AddTable`, you can assign the "LoadTable" event in the report and only load the tables when they are actually requested by the report. For example you could call:

```
FlexCelReport fr = NewFlexCelReport();
fr.LoadTable += new LoadTableEventHandler(fr_LoadTable);
fr.Run(...);
```

and define the event "fr_LoadTable" as:

```
void fr_LoadTable(object sender, LoadTableEventArgs e)
{
    ((FlexCelReport)sender).AddTable(e.TableName,
        GetTable(e.TableName), TDisposeMode.DisposeAfterRun);
}
```

You could also use `DirectSQL` or `User Tables` to avoid preloading the datasets.

WARNING

Let us repeat: **This only applies to datasets.** When using LINQ, data is always fetched from the server when (and if) it is needed.

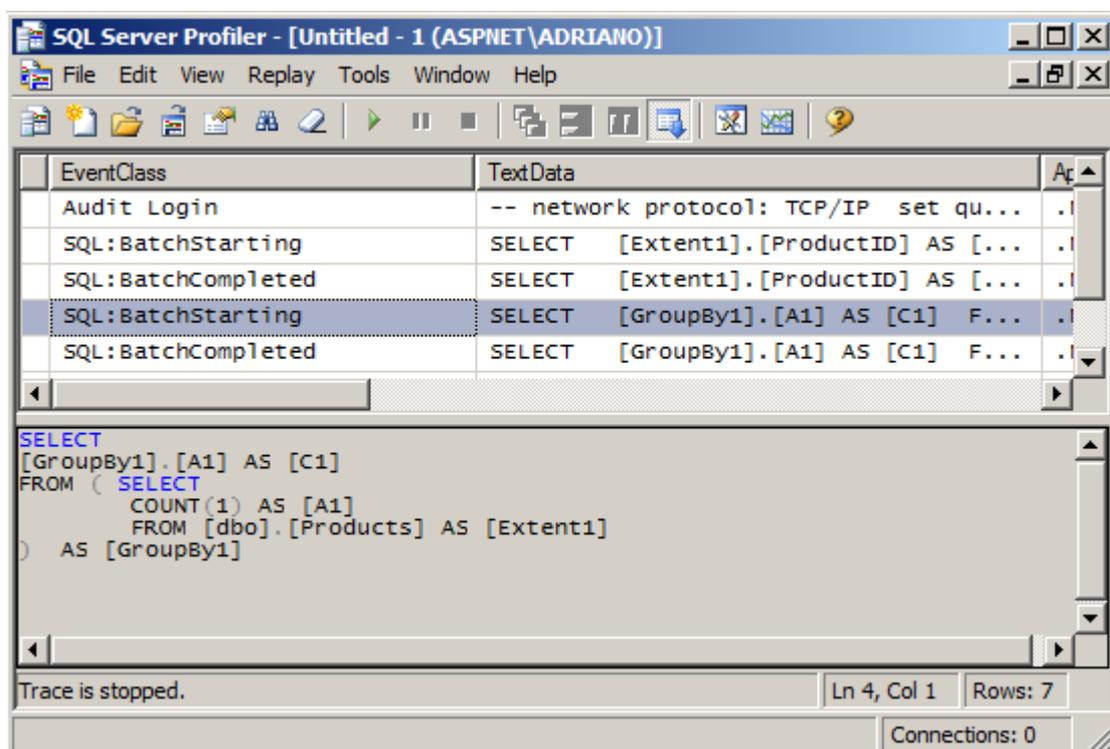
Profile the SQL sent to the server when using Entity Framework

When using `IQueryable<T>` datasets as datasources for the reports, SQL is generated on the fly and sent to the database. While this is good because you are not loading the datasets fully in memory, if you don't have the correct indexes in the database or if the queries are not correctly made, you might have a big performance impact. Sometimes the full table might be loaded from the database and then filtered locally. Make sure the SQLs sent have the correct "Where" part.

If you are using SQL Server, a simple way to check this is to run the SQL profiler in the server while running the report.

You should see two queries for every band, the first to query for the number of records in the band, and the second to retrieve the values. FlexCel needs the first query to know how many rows to insert in advance, since inserting rows in Excel is a costly operation.

The following example shows a typical log from a single table:



You can see the "Select count" statement highlighted in the screenshot above.

Make sure that the SQL sent to the server makes sense. For some tables that have the same data repeated over and over again it might make sense to cache the results.

Provide the Record Count if possible when using LINQ Datasets

FlexCel does its reports by inserting rows and columns. As inserting a row or column isn't a cheap operation, it tries to minimize the number of inserts. So, if you have a table with 1000 rows, FlexCel won't do 1000 inserts of one row each; it will ask for the row count to the database, and insert 1000 rows in a single pass.

For datasets this isn't an issue, as the dataset has all data in memory: the row count is a very fast operation just returning the length of the internal array that holds the data in memory. But when using LINQ datasets, this is not always the case. It is ok for List<...> collections since again "Count" will be called and that is fast. But in Entity Framework an SQL "Select count * from..." will be sent to the server, and in the worst case LINQ might need to iterate through all values in the IEnumerable to get the count.

If the collection that holds your objects is slow to compute the count, and you know the count anyway, you can tell FlexCel the record count by having a field "__FlexCelCount" in your business object. For example:

```
Class MyBusinessObject { ... public int __FlexCelCount{get;} }
```

If FlexCel finds a column named "__FlexCelCount" in your collection, it will assume this is the value of the row count, and it won't call "Count()" in the collection.

Please note that this is a last resort optimization. Count should be fast anyway for most cases, and by adding a "__FlexCelCount" field to your business objects, you will probably just making them more complex without any real gain. We provide this for really special situations, but this optimization isn't something that should normally be done.

Avoid Microsoft Distributed Transaction Coordinator (MSDTC) when using Entity Framework

When using Entity Framework as data sources, you normally need to run the report inside a "**Snapshot**" transaction to avoid data being modified while the report is running. If you don't open the database connection before running the report, transactions might be promoted to distributed transactions, as the connection will be opened and closed many times while running the report. When the connection is opened a second time inside the transaction, it will be promoted to distributed.

See

<http://www.digitallycreated.net/Blog/48/entity-framework-transactionscope-and-msdtc>

for more information.

Prefer snapshot transactions to serializable transactions

When reading the data for a report, there are two transaction levels that can be used to avoid inconsistent data: **snapshot** or **serializable**. While both will provide consistent data, “serializable” provides it by locking the rows that are being used, therefore blocking other applications to modify the data while the report is running. “Snapshot” transactions on the other side operate in a copy of the data, allowing the other applications to continue working in the data while the report is running.

If your database supports snapshot transactions, and your reports take a while to run, and other users might want to modify the data while the report is running, snapshot transactions are a better choice.

FlexCel PDF Exporting Guide

Introduction

FlexCel comes with a full PDF writer that allows you to natively export Excel files to PDF, without needing to have Acrobat or Excel installed. While the output will not be exactly the same as Excel, a lot of effort has been done to make it as similar as possible.

Creating PDF files

FlexCel provides two ways to create a PDF file. At a higher level, you can use [FlexCelPdfExport](#) component to convert an Excel file into a PDF file. At the lower level, you have the [PdfWriter](#) class, that provides a primitive API to create PDF files.

Using PdfWriter

[PdfWriter](#) is a lower level option, and it was really not designed to be used directly, but to provide the methods [FlexCelPdfExport](#) needs to do its job.

But it can be used standalone to create a PDF file for scratch, or most likely to **modify the output from FlexCelPdfExport using one of the [FlexCelPdfExport.BeforeGeneratePage](#) or [FlexCelPdfExport.AfterGeneratePage](#) events.**

We will not cover it in detail here since methods are documented in [PdfWriter](#), but it is worth mentioning that there is also an example to get you started in the [API Demos](#).

Using FlexCelPdfExport

This is a higher level option, and the one you would normally use. The simplest skeleton to export a file to pdf would be:

```
using (FlexCelPdfExport pdf = new FlexCelPdfExport(xls, true))
{
    pdf.Export("result.pdf");
}
```

While the above snippet of code should be enough in many cases, you can add to it in the following ways:

1. You can set the [FlexCelPdfExport](#) properties. You will find there are not a lot of properties (things like margins, printing gridlines or not, etc) and this is because all this information is read from the Excel file. If you need to change them, change the associated properties on the attached [XlsFile](#) object.

For example, to change the page margins you can use the [ExcelFile.SetPrintMargins](#) method.

TIP

As always, you can use [ApiMate](#) to find out how to change the printer settings in the `XlsFile` object.

But while all the document-specific settings are stored in the `XlsFile` object, that doesn't mean there aren't properties specific to the PDF output. You can change things like the `PdfType` or `Kerning` used in the PDF export. Make sure to take a look at the available properties in the documentation for `FlexCelPdfExport`

1. You can export multiple Excel files to the same PDF file. To do that, you use `FlexCelPdfExport.BeginExport`, then `FlexCelPdfExport.ExportSheet` or `FlexCelPdfExport.ExportAllVisibleSheets`, and finish the exporting by calling `FlexCelPdfExport.EndExport`

An example of multiple files would be:

```
using (FlexCelPdfExport pdf = new FlexCelPdfExport())
{
    pdf.AllowOverwritingFiles = true;

    using (FileStream pdfstream = new FileStream("result.pdf",
        FileMode.Create))
    {
        pdf.BeginExport(pdfstream);
        pdf.Workbook = xls1;
        pdf.ExportAllVisibleSheets(false, "First file");
        pdf.Workbook = xls2;
        pdf.ExportAllVisibleSheets(false, "Second file");
        pdf.EndExport();
    }
}
```

Font Management

The font handling when creating pdf files can be problematic, so here we discuss some concepts related to them.

Selecting which fonts to use

First of all, you need to know that there are two different kinds of fonts supported by FlexCel's PdfWriter.

1. **PDF internal fonts.** PDF defines 14 standard fonts that must be available to any PDF viewer, so you don't need to embed the fonts on the PDF document. They will always be supported by the PDF viewer.

Those fonts include a **Serif** (Times New Roman-like), a **Monospace** (Courier-like) and a **Sans Serif** (Arial-like) alternatives, on four variants each (regular, bold, italic and bold italic) and two Symbol fonts.

What those fonts **don't** include is Unicode characters outside the ASCII range, so if you are using those, you can't use internal fonts. **FlexCel will automatically change the fonts to be true type if your document contains characters outside the ASCII range.**

2. **True Type fonts.** Those are standard Windows fonts.

When exporting to PDF, you can choose between three different ways to handle fonts, depending on the value you set on the [FlexCelPdfExport.FontMapping](#) or [PdfWriter.FontMapping](#) property:

1. **ReplaceAllFonts.** This will replace all fonts on the xls file to the most similar ones on the 14 standard fonts. This way you get the minimum file size and the maximum portability, but the exported PDF file might not look exactly the same, as you will lose all fancy fonts.
2. **ReplaceStandardFonts.** This is a compromise solution. It will only replace Arial, Times New Roman and Courier for the standard fonts, and use True type for all the others. You will get a bigger PDF file (if you embed the true type fonts), but it will look the same as the Excel file.
3. **DoNotReplaceFonts.** This will only use true type fonts. It will be the one that better matches the xls file, but it will be a lot larger (if you embed the true type fonts) or might not look good when the user does not have the fonts you used installed (if you don't embed them)

NOTE

Besides choosing how the fonts will be replaced when creating the PDF, you can also choose whether to embed the True Type fonts or not. If you embed them, the file will be bigger, but also will render well when the user does not have the fonts on his machine. To avoid issues, it is normally recommended that you embed all fonts.

IMPORTANT

If you use Unicode characters, the fonts will always be embedded no matter which embed option you choose. This is needed to ensure that the Unicode mapping will remain correct.

NOTE

With the rise of **Android** and **iOS** devices you can't just assume that the final user will have any fonts (not even Arial) installed on the device where he is reading the file.

If you add the fact that the size increase by embedding the fonts isn't that much given the size of even a small webpage, **we can only recommend you that you just embed all fonts.**

FlexCel used to default to not embedding fonts, but since version 6.5 it defaults to embedding all fonts.

Font Subsetting

When embedding fonts you can choose if to embed the full font, or just the characters that were actually used in the document. You can control this with the property [FlexCelPdfExport.FontSubset](#)

If you embed just the subset of characters used, the file will be as you might expect smaller. For big fonts with thousands of characters the subsetting can make a big difference in size.

But on the other hand, if you embed only the subsets, the final users might not be able to edit the PDF document to add text, since the characters they might want to add might not be in the embedded subset.

You might see the issue that users won't be able to so easily edit the PDF documents as a feature, since PDF files are not normally generated for editing. And this is the reason FlexCel defaults to subsetting all fonts.

But if you care about the users being able to edit the PDF files you create, remember to set `FlexCelPdfExport.FontSubset = true`.

Accessing True Type data.

In order to embed fonts and get many font metrics that we need to create the pdf file, FlexCel needs to access the raw *.ttf files directly. FlexCel uses different ways to access the font data depending on the platform you are using (Windows, iOS, Android, etc):

1. On Linux (using SKIA, not Linux in Mono), iOS and macOS FlexCel asks the operating system for the font data directly, and there is nothing that you need to do.
2. On the rest of platforms (Windows , Windows Store and Android) FlexCel looks for the font folder, and reads the files directly from there.

When reading the ttf files from disk, FlexCel will try to locate the files in the following folders:

- **OS default font folders.**

- if the OS default font folder is empty, then we assume we are running on Linux (under Mono WinForms), and we will try `"/usr/X11R6/lib/X11/fonts/truetype"`. Note that when running in **Linux with SKIA** we will read the fonts directly from the OS, so we won't be looking for them in any folder. This applies only to Linux with Mono.
- if the folders above don't exist, FlexCel will search on `"<folder where FlexCel.dll is>/Fonts"`.
- If still failing to find the font folder, FlexCel provides a `FlexCelPdfExport.GetFontFolder` event that allows you to specify where fonts are stored on your system. Here you can tell FlexCel where the fonts are.

NOTE

Avoid using the `GetFontFolder` event if you can, since when you use it your code will not transparently run on different platforms. A Symbolic link from the FlexCel installation folder to the fonts should be a more elegant solution.

- Finally, if you have your true type fonts not available as files, but maybe as resources or entries in a database, you can use the `FlexCelPdfExport.GetFontData` event to provide the font data directly instead of a folder where FlexCel will look for the files.

NOTE

If you don't mind using p/invoke, you can use the [FlexCelPdfExport.GetFontData](#) event to call GetFontData on the GDI API, and return the font information to FlexCel. This way you will avoid scanning the font folder completely, and it can speed up a little PDF export.

While we do not recommend that you go unmanaged for this since scanning the font folder is fast anyway, you have all options. Do what you prefer.

For a demo on using the GetFontData event, see the [ExportPdf demo](#).

Fonts in Windows

Since Windows 10 version 1809, Windows has two different default font folders. The classic "c:\Windows\Fonts" (or similar) for all users, and a new folder for the current user only, located at %localappdata%\Microsoft\Windows\Fonts

FlexCel 7.6 or newer will search in both folders by default. If you are in an older FlexCel version, you might want to use the [FlexCelPdfExport.GetFontData](#) event to specify both folders. You will need to check if the %localappdata%\Microsoft\Windows\Fonts folder exists (because FlexCel versions older than 7.6 will raise an exception if any of the folders you pass to the GetFontFolder event don't exist). If the folder exists, you will need to pass both the global-font-folder and the user-font-folder to the event, separating them with a semicolon (;)

NOTE

Newer versions of Office 365 have [Cloud Fonts](#) which are fonts that are not available to Windows, only to Office.

Those fonts are installed in a private folder (which at the time of writing seems to be %localappdata%\Microsoft\FontCache\4\CloudFonts in Windows and ~/Library/Group Containers/UBF8T346G9.Office/FontCache/4/CloudFonts in macOS, but those folders might change)

FlexCel (and Windows itself) won't see those fonts, unless you add them to the Fonts search folder using the [FlexCelPdfExport.GetFontData](#) event. If you want to use those fonts in your document and with FlexCel, we would recommend that you install them in the Windows Fonts folder. But always check the font license to see if you are allowed to do so.

See also [this Cloud Fonts tip](#)

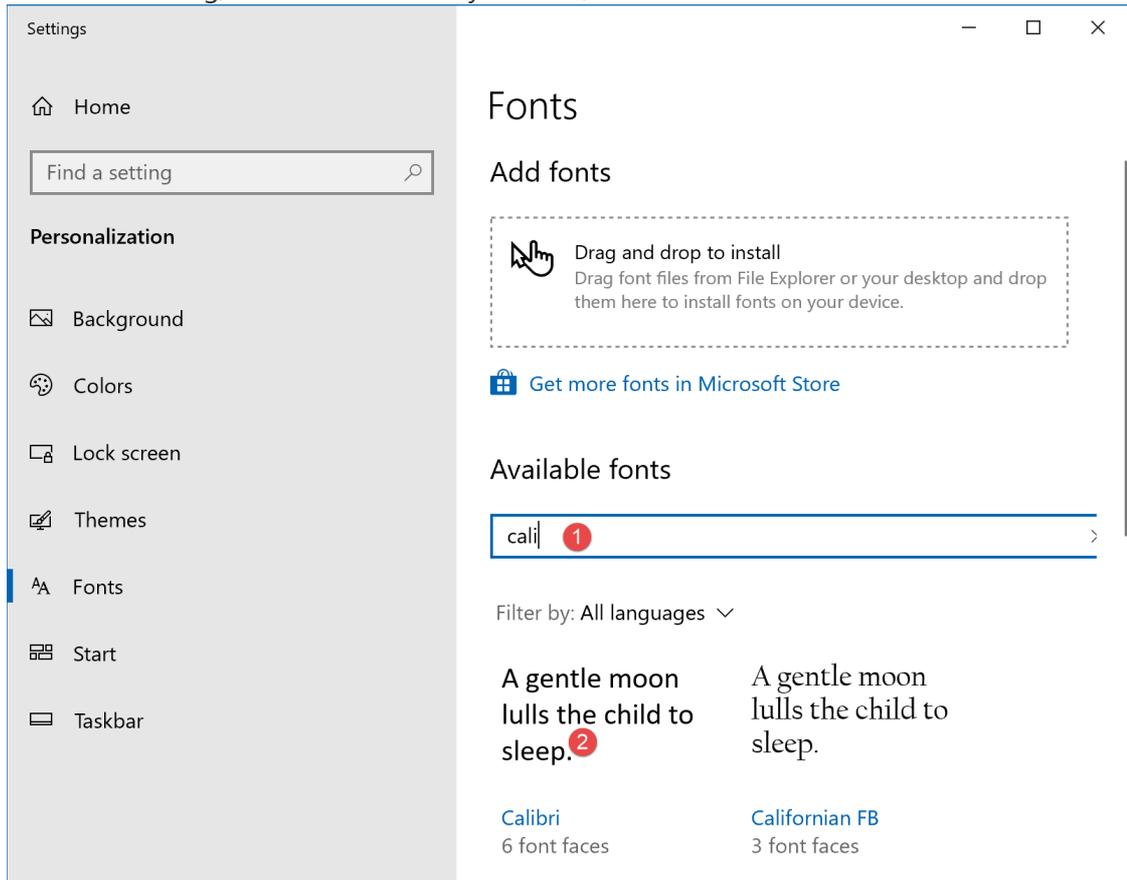
IMPORTANT

It is strongly recommended that you install the fonts in the operating system itself. If you just change the font path with the [FlexCelPdfExport.GetFontData](#) event, FlexCel will be able to find the font, but the OS itself will not. FlexCel uses OS functionality for example to measure the fonts, so if the OS can't see the fonts, it will likely report invalid data, unless there is a very similar substitute font.

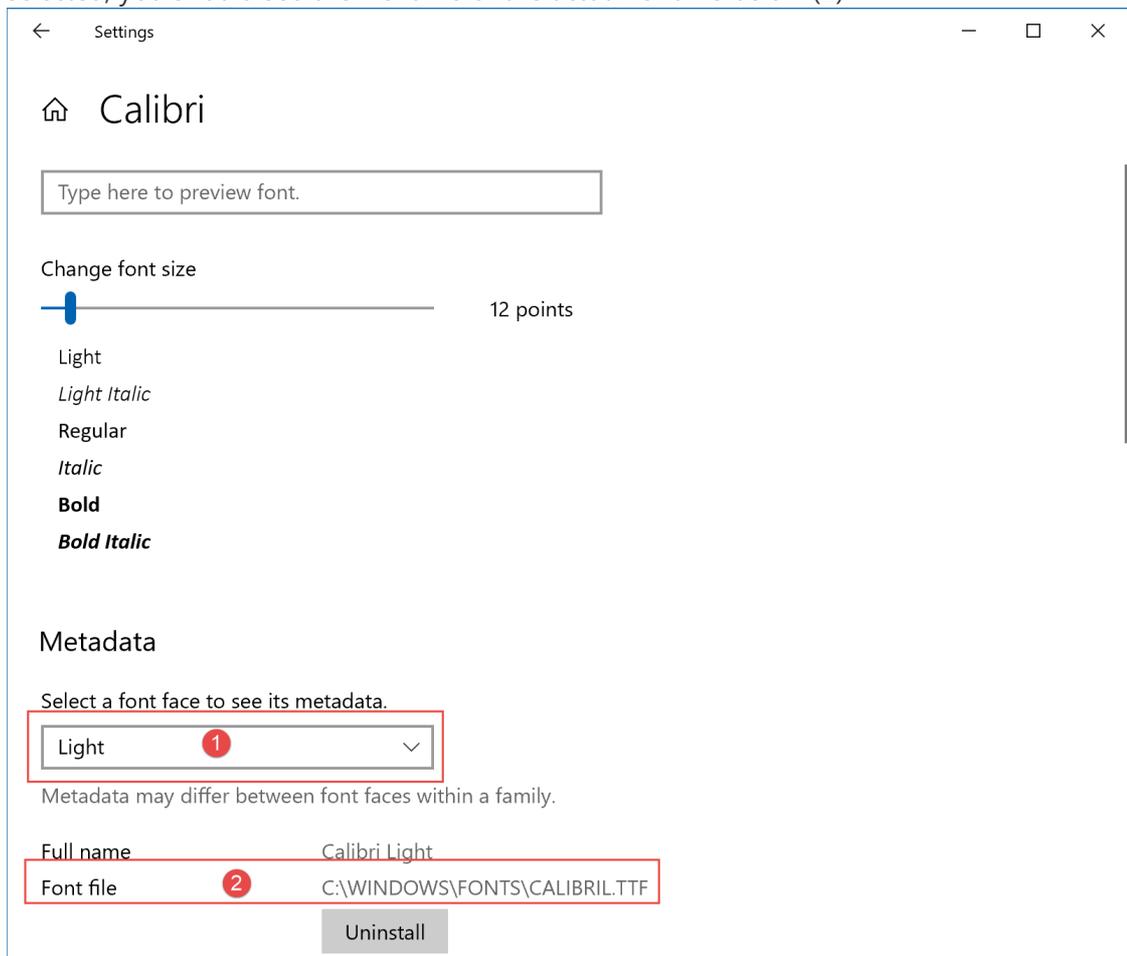
Finding where a font is installed

As explained above, a font might be installed in different folders, and new "default" font folders might appear in the future. If you see a font installed in Windows and want to know where the font actually is, you can follow the steps below:

1. In the start menu, search for "Font Settings"
2. In the font dialog, search for the font you want, and double-click it.



3. Once in the font, click in "Metadata" (1) and select the variant you want. Note that different font variants might be installed in different files, and even in different folders. Once selected, you should see the filename of the actual font file below (2)



Fonts in iOS and macOS

On iOS and macOS, FlexCel can access the true type directly from the Cocoa framework, so exporting to pdf should work without issues and without any extra step in those platforms. Note that in any case, the fonts available might be different from the fonts available in a Windows machine. You can get a list of fonts available in macOS here: http://en.wikipedia.org/wiki/List_of_typefaces_included_with_Mac_OS_X and in iOS here: <http://iosfonts.com>

Fonts in Android

At the time of this writing, in Android there are only 4 predefined fonts available for every app (Droid Sans, Droid Serif, Droid Mono and Roboto). This means that unless you want to use the internal pdf fonts, your application will have to provide its own fonts.

As in Android you will normally deploy your fonts as assets, FlexCel comes with prebuilt functionality for handling Assets. You can read more about it at [the Fonts section of the Android Guide](#)

Fonts in Excel 2007 and newer

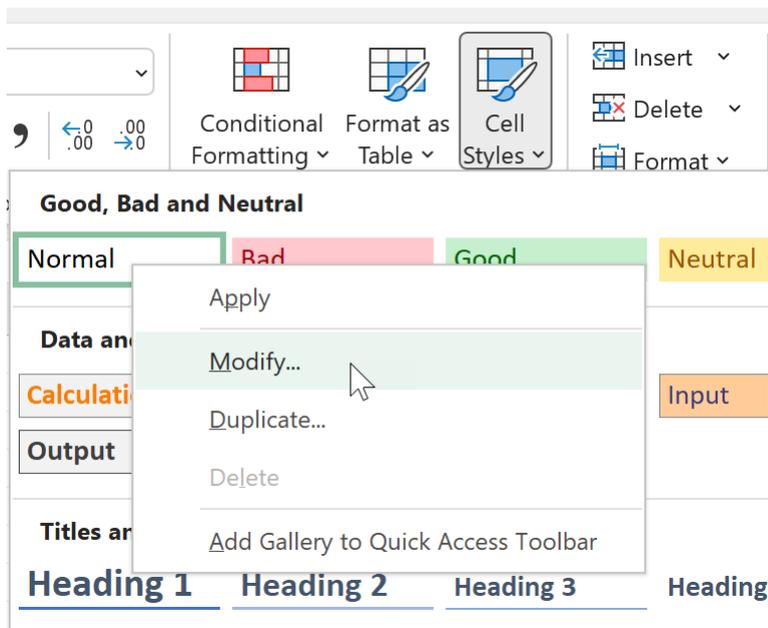
Excel 2007 changed the default font in a new file to be "Calibri" instead of "Arial". This might bring you problems if you develop in a machine that has Excel 2007 installed, but you deploy in a server that doesn't. There are two solutions to this:

1. You can **copy the Excel 2007 fonts to the server**, and make sure you **embed the fonts in the PDF file**. Note that if you do not embed the fonts, any user who does not have office installed might not be able to see your PDF file correctly.
2. If you want maximum portability, make sure you change all fonts to Arial or Times New Roman in your template before exporting.

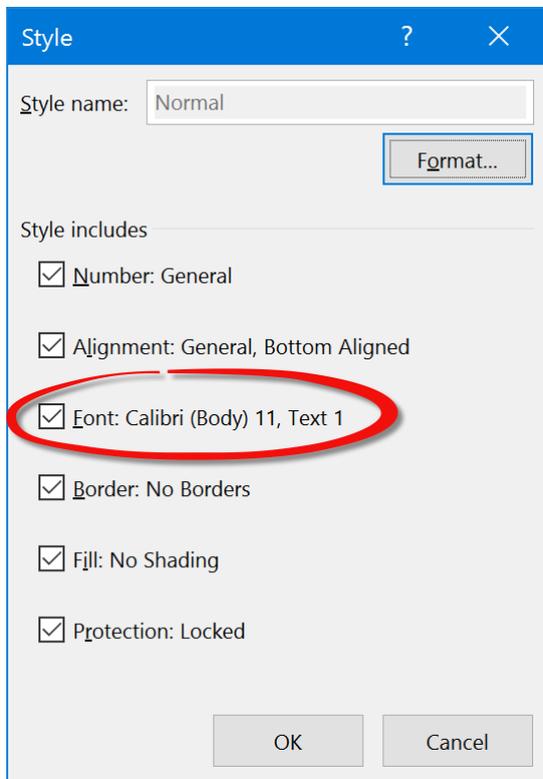
Setting the "Normal" font

Especially important when changing the fonts is to make sure the "Normal" format uses a known font. You can see/change the normal font this way:

1. In **Excel 97-2003**: Go to "Menu->Format->Style..."
2. In **Excel 2007 or newer**: In the home tab in the Ribbon, select **Cell Styles**, right click in "**Normal**" and choose "**Modify**". Note that depending on your screen resolution, "Cell Styles" might show as a button or just display the "Normal" box directly in the ribbon.



In both cases you should get a dialog similar to this:



Make sure the normal style uses a font you have in your server.

The "Normal" style is used not only for empty cells, but to set the column widths. For example, this is how an empty sheet looks with "Normal" style using **Calibri** with 11 points:

	A	B	C
1			
2			
3			

And this is how it looks using everybody's favorite font, **Comic Sans** with 28 points:

	A	B	C
1			
2			
3			

As you can see, the font used in the "Normal" style is used to draw the headings "A", "B", "1", etc., and even more important, it is used to calculate the column width. Column width is measured as a percentage of the "0" character width in the normal font. If you change the normal font, column widths will change.

If you do not have the "Normal" font installed in your server, Windows will replace it with a substitute, and it will probably have a different width for the "0" character, leading to a wrong column width. **So it is important that you have the normal font installed in your server.**

Dealing with missing fonts and glyphs

There are three main font-related problems you might find when converting an Excel file to PDF, and we are going to cover them in this section. The errors are non-fatal, and that means that the file will be generated anyway, but it will not look as good as it could.

You can control what to do when any of these errors happen by hooking an event to the [FlexCelTrace](#) static class. From this event, you could write to a log file when any of these errors happen, warn the user, or just raise an exception if you want to abort file generation.

Problem 1: Missing fonts

This is normally the easiest one to solve, and normally happens when deploying an application to a server. As explained in the section above, this often happens with "Calibri" font that gets installed by Excel, and probably will not be installed in the server. As FlexCel needs the font to be present in order to create the pdf file, it will substitute it with a "similar" font, normally Arial or Microsoft sans serif.

This might not be an issue if there are any fonts in the system that are similar to the one being replaced, but it can be a big issue with Calibri, since that font has very different metrics from the font it gets replaced (Arial). As an example, here you can see an Excel file using Calibri exported to PDF in a machine that has Calibri installed and in other that doesn't:

With Calibri installed in the fonts folder:

	A	B	C	D
1		This text is in Calibri	Balance of year	Balance Sheet
2		This text is in Arial	Balance of ye	Balance Sheet

Without Calibri installed (Replaced by Arial):

	A	B	C	D
1		This text is in Calit	Balance of ye	Balance Sheet
2		This text is in Arial	Balance of ye	Balance Sheet

As you can see in the images, Calibri is much narrower than Arial, so the text in cell B2 "This Text is in Calibri" is cut in the second screenshot. If you are seeing lots of cut text in the server while the files are exported fine in your development machines, this is probably the cause.

TIP

You can get a Calibri clone in Linux by installing the [Carlito](#) font:

```
sudo apt-get install fonts-crosextra-carlito
```

You can also get a Cambria substitute:

```
sudo apt-get install fonts-crosextra-caladea
```

And to get the "classic" fonts like Arial or Times New Roman, you can either use the [Google Crosscore fonts](#), the [Liberation fonts](#) or install the [Microsoft core fonts for the web](#):

```
sudo apt-get install ttf-mscorefonts-installer
```

The solution to this problem is easy; make sure you have all the fonts you use installed in your system. If you want to get notified whenever this automatic font replacement happens, you can catch the **FlexCelError.PdfFontNotFound** errors in [FlexCelTrace](#), and use it to notify the user he should install the missing fonts.

NOTE

To change all fonts of a given type to a different type in the Excel file instead of the pdf, take a look at [Replacing a font by another in an Excel file](#).

Problem 2: Missing Glyphs

This problem happens when you are using a font that doesn't contain the character you want to display. If you for example write

“日本に行きたい。”

inside a cell and keep the font “Arial”, you will see the correct characters in Excel, but when exporting you might see blank squares like this:

□□ □□□□□□

The reason for this is that “Arial” doesn't actually contain Japanese characters, and Excel is “under the hood” using another font (normally MS Mincho) to display the characters. To emulate this behavior, FlexCel provides a **FlexCelPdfExport.FallbackFonts** property, where you can enter a list of fonts to try if the font that was supposed to be used doesn't have the character. If no font in the FallbackFont chain contains the glyph, you will see a blank square.

The solution in this case is to use fonts that actually have the characters you want to display, or ensure that some fonts in the FallbackFonts properties have them. By default FlexCel uses “Arial Unicode MS;Segoe UI Symbol;Yu Mincho;Yu Gothic;Ms Mincho;Ms Gothic” as fallback fonts, but you can add as many others as you need.

TIP

Windows 10 changed which fonts are available in a default Windows installation. So for example while in Windows 8 or older Arial Unicode would be a font installed by default, in Windows 10 it isn't.

It also replaced "Ms Mincho" by "Yu Mincho" and "Ms Gothic" by "Yu gothic".

This is the reason the default Fallback fonts in FlexCel include so many fonts: We can't know in which OS you are going to run it, so we try both the default fonts for Windows 10 and for older.

TIP

Usually setting the fallback font is enough to deal with missing glyphs. But some fonts, especially for non-Latin characters, might not come with bold or italic variants. So when the fallback font is used, the bold or italics won't show, or show as faux-bold/faux-italics (See Problem 3 below).

As always, the best solution here is to use fallback fonts that have italics and bold variants. But if you can't find a font that has them, you might need to specify different fallbacks for bold or italics.

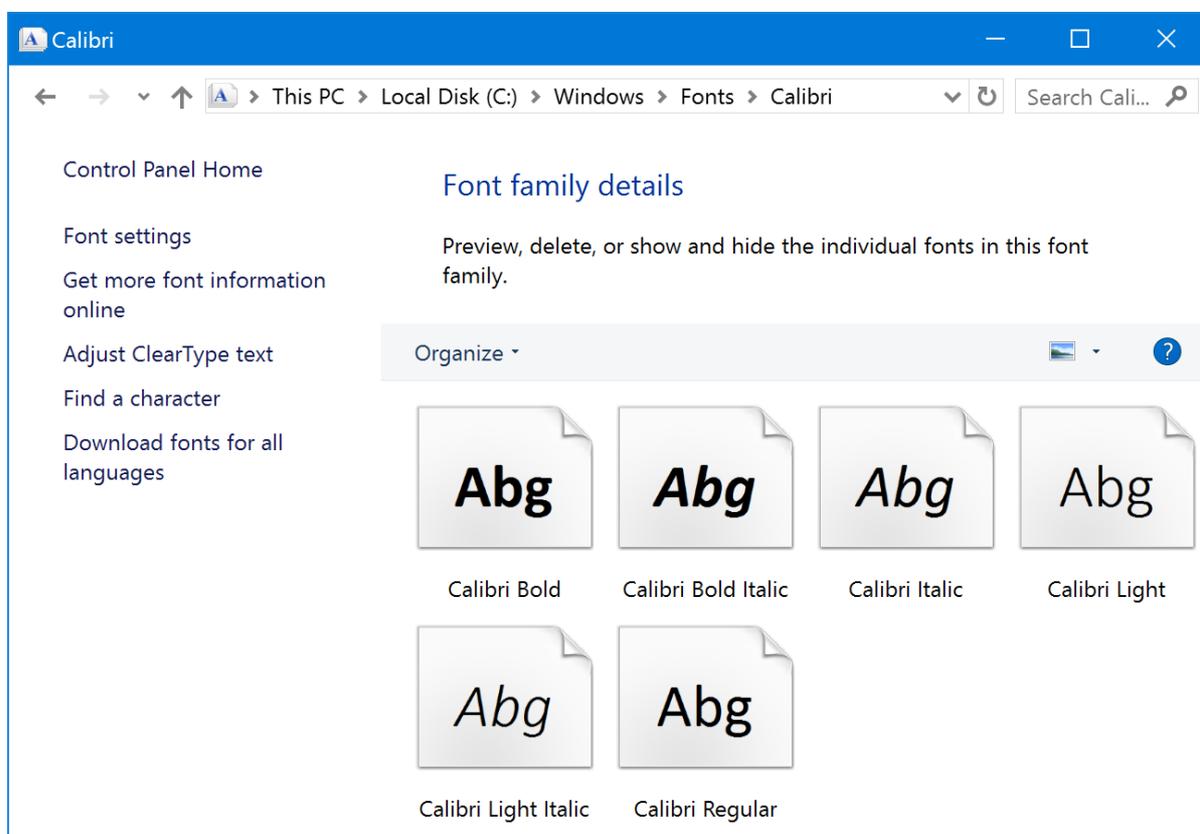
Let's see a little example: Imagine that you have Font1 which has only non-bold characters, and Font2 which only has bold. If you set the FallbackFonts to Font1, then FlexCel won't use Font2 for the bold, and if you set Font2 in the fallbacks, then FlexCel will use the bold font even for non-bold characters.

To deal with this, you can use the properties [FlexCelPdfExport.FallbackFontsBold](#), [FlexCelPdfExport.FallbackFontsItalic](#) and [FlexCelPdfExport.FallbackFontsBoldItalic](#) to specify different fallback fonts for bold, italics or bold and italics variants. So you can set Font1 as the main FallbackFont, but specify Font2 as FallbackFontsBold. When a font is bold, FlexCel will first look at FallbackFontsBold, and only if it can't find a suitable font there, then go and search for the default FallbackFonts.

If you want to get notified when a fallback font replacement happens so you can warn the user to change the fonts, you can catch the [FlexCelError.PdfGlyphNotInFont](#) and [FlexCelError.PdfUsedFallbackFont](#) errors in [FlexCelTrace](#).

Problem 3: Faux Italics and Bolds

The last problem is related to fonts that don't have a specific "Italic" or "Bold" variant. Normally, a quality font comes with at least four files including four more common variants of the font: Bold, Italic and BoldItalic. And they can include even more variants. If you look in your font folder, you will see things like this:



That is, a different file for each font variant. If the font comes with only one "Normal" file, the italics can be faked by the system by slanting the characters, and bold can be faked by adding some weight to the characters. But of course this leads to low-quality results, and should be avoided whenever possible.

The solution to this problem is to use fonts that include the variants you need.

Normally this is not a problem, since all quality fonts designed to be used with Italics and Bold already come with files for those variants.

NOTE

There is a property named [FlexCelPdfExport.UseFauxStyles](#) which you can set to false if you don't want FlexCel trying to simulate bolds or italics for the fonts that don't have those styles.

To be notified whenever FlexCel finds a "fake" style, you can use the [FlexCelError.PdfFauxBoldOrItalics](#) notifications in [FlexCelTrace](#).

Accessibility of the generated files

Setting a natural language

It is important to set a natural language for the document if you know what language the document is written in. This way a screen reader or a text-to-speech engine will be able to correctly read the text out loud.

To set up the language in FlexCel to for example Spanish, use code like this:

```
pdf.Properties.Language = "es-ES";
```

Tagging the files

FlexCel allows to create Tagged PDFs, which contain extra information about the document structure (like for example what are the cells in the table). This information allows a screen reader to know the correct order to read the text.

As it is an important accessibility feature, since FlexCel 6.5 files are tagged by default. You must explicitly turn tagging off with [FlexCelPdfExport.TagMode](#) in order to get untagged pdfs.

IMPORTANT

Tagged pdfs can be much bigger than normal ones, so this might be a reason to turn tagging off. If your files are big, try saving with and without tagging to compare the sizes; then decide if tagging is worth. For small documents, tagging should be kept on.

Creating PDF/A files

PDF/A files are files designed specifically for archiving. FlexCel has full support for the variations of the standard: **PDF/A1a**, **PDF/A1b**, **PDF/A2a**, **PDF/A2b**, **PDF/A3a** and **PDF/A3b**.

If you need to choose a version, we would recommend PDF/A2 or PDF/A3. PDF/A1 is a little too restrictive, and lacks some features that FlexCel could use to generate better files: It doesn't support transparency and it doesn't allow compressing the tags in the document. Due to the lack of transparency, if you have any transparent image in your file it might look wrong. Due to the lack of tag compression, files will be much bigger than PDF/A2.

In order to create PDF/A files, you need to set PdfWriter or FlexCelPdfExport property [FlexCelPdfExport.PdfType](#) to the correct version. For example:

```
pdf.PdfType = TPdfType.PDFA1
```

Then you need to choose if you want to generate "a" (PDF/A1a, PDF/A2a, PDF/A3a) or "b" (PDF/A1b, PDF/A2b, PDF/A3b) files. "a" files are the most complete, and they require you to tag the file. "b" files don't require tagging, so they can be smaller if the documents have a lot of pages.

When using FlexCelPdfExport, you would just set the correct option by changing the [FlexCelPdfExport.TagMode](#) property:

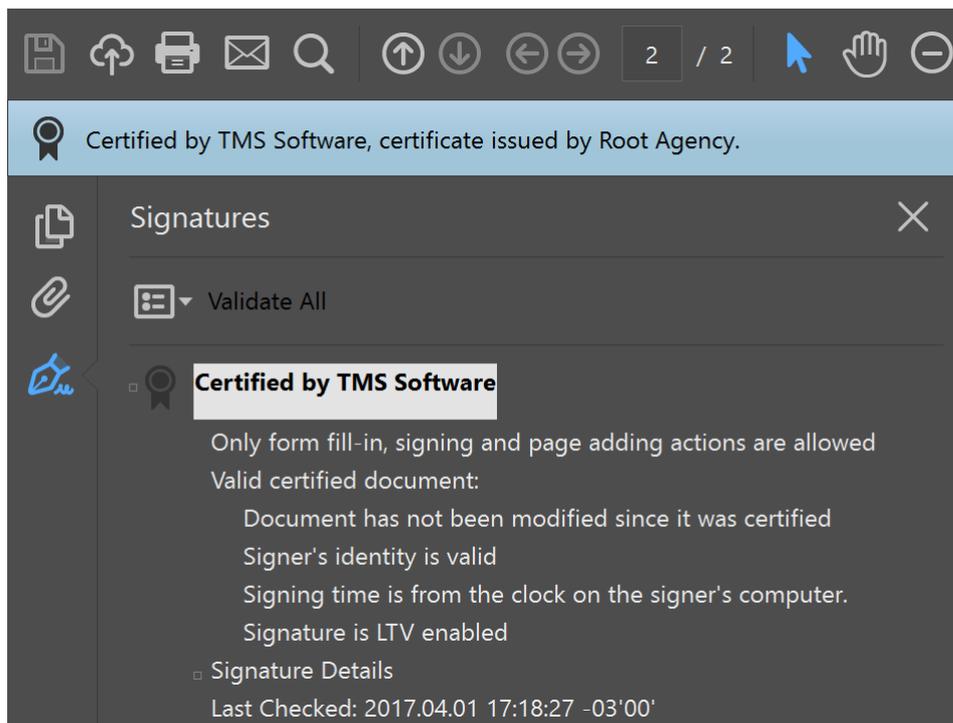
```
pdf.TagMode = TTagMode.None; //Generates "b" files
```

As the TagMode is Full by default, FlexCel by default generates "a" files.

When using PdfWriter, you need to manually tag the files, as FlexCel can't know the structure from the drawing commands. You need to use the methods [PdfWriter.TagContentBegin](#) and [PdfWriter.TagContentEnd](#) to specify the blocks of text you want to tag, and then set the TagActions property to specify how that tagged content relates to the structure of the file. Tagging in PdfWriter is an advanced topic outside the scope of this document. Due to the way PdfWriter is designed, it won't keep tags in memory and you need to write them directly to the file as you are creating it.

Signing PDF Files

FlexCel allows you to sign your PDF files with a certificate, so any change to the file will invalidate it. This is how a signature looks like in Acrobat DC:



IMPORTANT

FlexCel signing algorithm is supported only in Acrobat 7 or newer, it will not work in Acrobat 6 or 5 since those versions do not have the required support. Furthermore, SHA512 is only supported in Acrobat 8 or newer, and SHA1 is known to have vulnerabilities and it is not recommended to use. Because of these reasons, the header in the generated PDF file will automatically switch to say that the file is "Adobe 8 or newer" compatible when you include a signature in your files. If there are no signatures, the default header specifies the file is "Acrobat 5 or newer" compatible.

It is also worth noting that users will still be able to see the generated files in Acrobat 5, 6 or 7, but they will get a warning when opening them and the signature will not validate.

Concepts of signing are outside the scope of this document, but you can find a lot of information in signing in the Acrobat documentation or just in Internet. A good place to start might be:

<http://msdn.microsoft.com/msdnmag/issues/07/03/NETSecurity>

And you can look at the [Signing Pdfs](#) example to see how to sign a PDF with FlexCel.

Customizing the Signing Engine

FlexCel comes with a built-in signing implementation, but it allows you to change it by your own in case you have a better signer implementation. There is mainly one reason you might want to do that:

The built-in engine is not optimal. It uses the standard .NET PKCS classes, and those classes do not allow for incremental signing. This means that the whole PDF document must be kept in memory so we can pass all the contents as a byte array to the signing method in .NET. If you have a library that supports incremental signing, that is, each time a small group of bytes is written you recalculate the hash instead of calculating the whole hash at the end, you can save memory by creating your own engine. **Just make sure you need the extra performance** before doing so, because normally the built-in implementation is good enough, and has the advantage of being 100% managed and built-in in the framework.

If you decide to create your own Signer class, you need to implement two simple abstract classes:

1. TPdfSigner

This is the class that implements the signing. You need to override three methods:

TPdfSigner.Write, **TPdfSigner.GetSignature** and **TPdfSigner.EstimateLength**. The first method is called each time some bytes are written to the file. You should use them to calculate a PKCS7 signature with them. If you can't calculate the signature incrementally you will need to buffer them and calculate it when GetSignature is called. The second method, GetSignature, is called only once at the end of the pdf file and it should return the PKCS encoded signature as an array of bytes. The third method must return the length of the byte array that will be returned by GetSignature or a bigger number, but never smaller. Note that this third method will be called before the signature is computed so you might need to estimate the length.

2. TPdfSignerFactory

This class is really simple, and it just should return an instance of the particular [TPdfSigner](#) child you created in 1).

When creating your own classes, it might be helpful to look at the [TBuiltInSigner](#) implementation on file Signatures.cs. Also look at [ExportPdf](#) demo.

Export to PDF and FlexCel recalculation

When you create a report with FlexCel most formulas are recalculated, but some are not. This is not an issue when opening the file with Excel, as Excel will recalculate the whole file again, but will be an issue when exporting the Excel file directly to PDF.

FlexCel implements over [300 Excel functions](#), and most used formulas are there so you should not experience big issues. But you need to make sure you do not use any not implemented function to get a properly recalculated sheet.

Other reason why recalculation might now work is because you are using [User defined functions](#) and you haven't defined those functions in FlexCel.

And the last common cause why FlexCel could fail to recalculate is if you have [linked files](#) and you haven't set a [TWorkspace](#) object to calculate those links.

FlexCel comes with a little utility, the demo "[Validate Recalc](#)" that will allow you to check if all the formulas on an Excel file are ok to use. And of course you can use the code on this demo inside your own application to tell your users when they use a not supported formula.

FlexCel HTML Exporting guide

Introduction

One of the things FlexCel allows you to do is to export your xls/x files to HTML. We do our best to create the HTML files as closely as possible to the original xls/x file, but there are some restrictions in HTML that make it not as good as a PDF export. HTML is a "Flow" format, as opposed to PDF (which is a "Fixed Page" format), and this means the browser can resize and reposition elements depending on the output medium.

With this limitation in mind, we will study now how this is done.

Creating HTML files

There are two ways to create an HTML file from inside FlexCel:

1. Using the **FlexCelHtmlExport** component.

This is the most flexible way to create an HTML file.

2. Using the **FlexCelAspViewer** Component.

This is a component that allows for "drag and drop" display on a web page of an Excel file, using ASP.NET Webforms. Internally it uses FlexCelHtmlExport, and provides some extra functionality to avoid repetitive tasks.

Creating a file using FlexCelHtmlExport

FlexCelHtmlExport is a straightforward class. You would normally set its **FlexCelHtmlExport.Workbook** property to the Excel file you want to export, and then call **Export**, **ExportAllVisibleSheetsAsTabs** or **ExportAllVisibleSheetsAsOneHtmlFile** depending on what you want to export.

As always, we will not cover those methods or the properties on FlexCelHtmlExport in detail, since they are described in the reference and it makes no sense to repeat the information here. This document is about the conceptual part of creating HTML files.

Naming the created files

The first thing we need to note is that creating an HTML file is different from creating a PDF or other types of files in the sense that you actually create many files for a single xls/x file.

Exporting "workbook.xlsx" to PDF will return in just one file: "workbook.pdf", but it might result in 3 files when exporting to HTML: "workbook.htm", "workbook_image1.png" and "workbook.css"

So we need a way to name the different generated files. By default **FlexCelHtmlExport** will take the name of the main HTML file as a parameter to the export, and generate the name of the images with the following pattern:

<ImagesFolder>\<htmlfilenamewithoutextension>_image<imagenumber>.<imageext>

You can change this pattern, but all in all it gives a good default for images being created. You might use a GUID too as name for the generated images, by changing the [FlexCelHtmlExport.ImageNaming](#) property.

Now, you might want or need more control than this over the filenames created, and FlexCelHtmlExport gives you that with the [GetImageInformation](#) event, where you can specify exactly which filename you want for each image or even a stream for each one if you are saving for example the file to a database and not to a file system.

Remember that [GetImageInformation](#) will be called for every image in the Excel file, even if they are not real images. For example, a chart will be rendered as an image, and so it will throw an OnGetImageInformation event. Also, if you have vertical text and the [FlexCelHtmlExport.VerticalTextAsImages](#) property is true, then this event will also be called when creating the images with the vertical text.

TIP

When exporting to HTML5, you might choose to embed the images inside the files so there are no extra files generated.

This is not a good idea for big images, but it can help to avoid the naming issues if your images are light.

You can export to HTML5 by setting the [FlexCelHtmlExport.HtmlVersion](#) property to [THtmlVersion.Html_5](#) and you can control which images to embed with the [FlexCelHtmlExport.EmbedImages](#) property.

TIP

If you prefer not to use HTML 5 but still would like to embed the images, for example to send the file by email, FlexCel can also export to MHTML which is a format that supports embedding the images.

CSS files are another place where you might create extra files. The HTML standard allows embedding the CSS files inside the generated HTML or using an external stylesheet, and so does FlexCelHtmlExport. If you decide to go for an external stylesheet, you need to tell FlexCelHtmlExport where to place it, by providing a [TCssInformation](#) class or a CSS filename to the export methods.

Embedding the generated HTML inside your own pages

Another thing you that is different in generating HTML from other file formats, is that you might want only a part of the html file and not all of it. For example, imagine you want to embed an Excel file inside your existing company page. You will not want the whole html file (since embedding <html> tags inside other <html> tags is not valid HTML), but only the part between the <body> tags. And you will want to have the <head> part of the file in a separated place, so you can merge it with the <head> tags in your site.

With FlexCelHtmlExport, you can use the [PartialExportAdd](#) method and the [TPartialExportState](#) class for this.

You start by creating a new `TPartialExportState` instance, where FlexCel will store all the information it needs to create your files.

Then, you call `FlexCelHtmlExport.PartialExportAdd` for each sheet or file you want to add to the HTML file, using the same `TPartialExportState` instance as parameter.

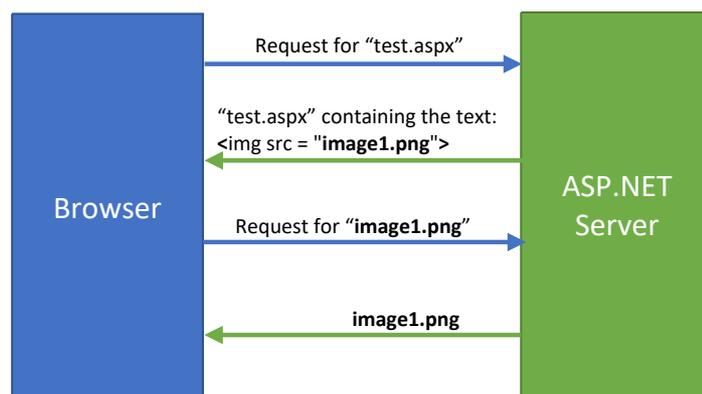
Once you have added all the sheets and files you want, you can use the methods in `TPartialExportState` to get the individual parts of the HTML file.

Creating a file using FlexCelAspViewer component.

The `FlexCelAspViewer` component is a component for WebForms, that allows a smooth integration of the xls/x file with your ASP.NET site. Internally it uses `FlexCelHtmlExport`, and as explained in the section above, it uses the **PartialExportAdd** method and **TPartialExportState** class to inject the different parts of the HTML file inside the ASP.NET file you are designing.

While normally really straightforward to use (just drop it and set its properties), there is one thing to take into account, and this is images.

`FlexCelAspViewer` needs a way to feed the images back to the browser. While it can send the HTML text back to the main stream when the browser requests an ASP.NET page, it has no way to send the images to the browser, when the browser requests them after reading the HTML file. Remember the browser will make many requests, the first for an aspx file, and then for the individual png/jpg/gif files, as shown below:



In this example we need a way to provide "image1.png" from the ASP.NET server. On the request for the page (test.aspx) we can create the report and send back the HTML, but we need to persist all images here, so when the browser asks for them we know how to get them.

We provide three operating modes for images so they can be sent to the browser, and they are controlled by the `ImageExportMode` property:

TImageExportMode.TemporaryFiles

This is the simplest mode, and when using it, FlexCel will output all the images to a temporary folder when the browser asks for the page. The image links in the main HTML file will link to this folder, and so the browser will be able to get them when it needs them.

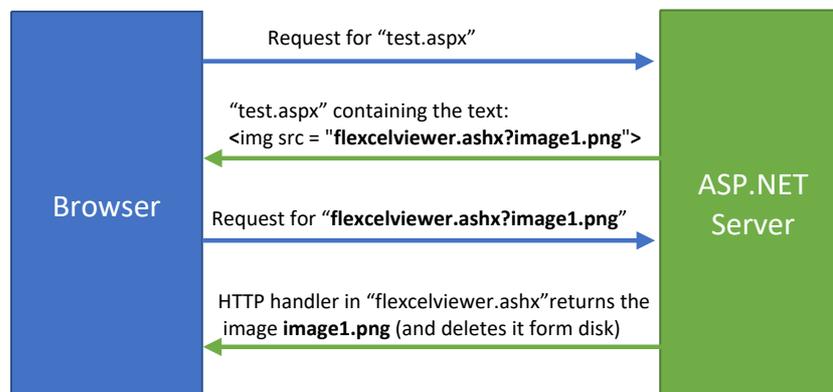
Images will be named with a GUID, so even if two different browsers ask for the same report at the same time, they will get different images.

When working in this mode you will need to implement a “Garbage collection” of images older than a given timespan, in order to avoid infinite grow of temporary images. You can do this with some scripts on the server, or use the [ImageTimeout](#) property in FlexCelAspViewer.

TImageExportMode.UniqueTemporaryFiles

This mode is not actually suited for wide use, but can be used in controlled environments, where you can test that the browsers work fine. **Not all the browsers will work in this mode.** Older browsers might ask for the image twice, and will not get it the second time

On this mode, as in the first one, images will be saved to a temporary folder when the main page is requested by the browser. But, instead of links to the image, the generated HTML will contain links to HTTPHandlers, as shown below:



The advantage over the first method is that images will be **deleted by the HTTP handler once they are served**. This provides a more scalable solution, since you do not rely so heavily on the garbage collection, but on the other hand if a browser re-asks for the same image it will not find it.

IMPORTANT

It is critical to realize that while this method minimizes the need for garbage collection on temporary images, it does not eliminate it. You will still need to periodically delete the images from the images folder!

The reason why there could still be images left even when they are deleted by the HTTP handler is simple, and it is that the HTTP handler might never be called. Let's imagine the following “broken flow”:

As in the picture above, the browser requests an html file from the ASP.NET server. And as above, the ASP.NET server returns the html file with links to the handlers for the images.

But before the browser actually requests the handlers in the HTML file, the user loads another web page. In this case, the "flexcelviewer.ashx" will never be called, and the image will never be deleted.

Now let's imagine a malicious user, trying a "denial of service" attack to your server. He could make a script that asks for a page in your server, but never calls the handlers, creating lots of temporary files that will never be deleted. In fact, just refreshing his browser manually hundreds of times will get this effect. In each refresh a new html page is generated (along with all the corresponding temporary images), but as he presses refresh again before the page is loaded, the image handlers will not be called and the images will not be deleted.

So, while this method provides a better mode for regular use (where the user requests the page and waits for the images), it is no better to prevent malicious attacks. FlexCelAspViewer provides a property, [MaxTemporaryImages](#), that can be used to mitigate this effect, by deleting older files once you have too much temporary files in the disk. It is recommended that you set MaxTemporaryImages to a reasonable value.

TImageExportMode.CustomStorage

This is a more advanced method that allows you complete control over how the images will be delivered to the browser. It works by creating links to ASP.NET HttpHandlers and allowing you to customize those HttpHandlers to your needs.

It is in some ways similar to the UniqueTemporaryFiles method in that the generated HTML file will have links to HTTP Handlers instead of images, but it is your job now to implement it.

There are two things you need to implement here:

1. You need to specify the image links for the HTML file, providing the parameters you need in order to get the images when the http handler is called.
2. You need to implement the HTTP handler that will return the images to the browser. It could use temporary files as the second mode, or it could cache them in memory, a database or wherever you want.

In order to specify the image links you need to assign the event [ImageLink](#) in the FlexCelAspViewer component.

You have complete freedom in how to implement your HttpHandler as long as you implement the IHttpHandler interface, but you can derive your class from FlexCelHtmlImageHandler to avoid writing the basic code. The code in FlexCelHtmlImageHandler has been adapted from the article:

<http://www.hanselman.com/blog/PermaLink,guid,5c59d662-b250-4eb2-96e4-f274295bd52e.aspx>

and it provides the basic boilerplate common to all Http Handlers. We would like to thank Scott Hanselman for it.

You might also take a look at the implementation of

UniqueTemporaryFilesImageHandler class in the code in order to get an idea of how a real implementation might look like.

When implementing the handler, you will need to use the [SaveImage](#) event in FlexCelAspViewer in order to get the images you will need to serve later.

IMPORTANT

Even when it is mentioned in the FlexCel reference, we think this is important enough to repeat here: Take a lot of care when creating your own custom handler. A naive implementation might provide a malicious user access to any file in the server, if you fail to validate the parameters correctly.

Improving the image handling in TemporaryImages mode

While TemporaryFiles is probably the best mode for general use, it has the drawback that can create too many images in heavy loaded sites. Each time a user requests a file, there will be an image created for each of the images in the xls file, and named with a unique name so it doesn't crash with the other users. This means that if you have 7 users asking for the same document at the same time, and the document has 8 images, this mode will generate $7 \times 8 = 56$ different images. If of all those images only one is dynamic (for example a chart), and the other 7 are static (for example the company logo) you can minimize the number of images created by supplying a [FlexCelHtmlExport.GetImageInformation](#) event, and setting the ImageFile parameter to null (so no image is created) and the ImageLink pointing to a place where you have the static images.

For example, you could have this event handler:

```
private void htm_GetImageInformationOverwrite2(object sender, ImageInformationEventArgs e)
{
    string ShapeName = e.ShapeProps.ShapeName;
    if (ShapeName == "logo")
    {
        e.ImageFile = null;
        e.ImageLink = "images/logo.gif";
    }
}
```

And manually place logo.gif in the images folder, so it does not have to be generated each time.

You can do even more things. If you set the ImageFile parameter to an absolute value (for example "c:\www\images\logo.png"), and the link to point there (for example "/images/logo.png") each time you create a file those images will be recreated so you do not have to care about manually updating them.

The only issue here is that when two users ask for the same image at the same time, one will get a sharing violation because the image is being created by the other thread.

FlexCelHtmlExport has a property "IgnoreSharingViolations" that is true by default and fixes this issue. When a file is locked by another thread, FlexCelHtmlExport will assume this is because there was another thread writing the same image, so it will simply not write it.

In the example, if two threads try to write to "c:\www\images\logo.png" at the same time, the first one will succeed and the other will silently fail and continue with the rest of the file. When the browsers then ask for "/images/logo.png" both browsers will be served with the same image that was written by the first thread.

Customizing the generated HTML

General Customization

Customization is a very important part of creating HTML files, since differently from xls/x or pdf files, HTML files usually have to be integrated with an existing site. This means the style of the generated pages must match the general look and feel of the whole website, and we tried not to cut any corners in letting you do so.

Customizing the raw HTML

There is one property, [FlexCelHtmlExport.ExtraInfo](#) where you can add extra HTML in different parts of the document. For example, you can use this property to add META tags to your file including the keywords for searching.

Customizing the Sheet Selector

When exporting a workbook to different HTML files we offer three ways in which you can customize the tabs that represent each sheet:

1. You can get basic customization of colors and properties by changing the [TStandardSheetSelector.CssTags](#) property in the [TStandardSheetSelector](#) class.
[TStandardSheetSelector.CssTags](#) is a collection of macros that will be replaced in the style definition of the Selector, allowing to change the width of the selector (when placed at the right or at the top, the background color, etc). There is detailed information on the available properties you can change in the [TStandardSheetSelector.CssTags](#) reference.
2. If the basic customization is not enough, you can completely redefine the css properties in a [TStandardSheetSelector](#) class. There is a general property for the styles that apply to all the position in the selector ([TStandardSheetSelector.CssGeneral](#)) and then customized properties for all the other positions.

For example, you could set border black for all [TStandardSheetSelector.CssTags](#) with [CssGeneral](#) style, and then redefine the border as blue when the tab is on the left by using the [TStandardSheetSelector.CssWhenLeft](#) property.

You can use the built-in macros when defining your CSS properties and you can even define your own. For example, you could define:

```
TStandardSheetSelector.CssWhenLeft.Main = "float: left; width: <#width>";
```

And then set

```
TStandardSheetSelector.CssTags ["width"] = "100px"
```

As explained, you can even **define your own macros**. For example you could use

```
TStandardSheetSelector.CssWhenLeft.Main = "float: <#floatpos>";
```

And then add "floatpos" to the [CssTags](#) array. This way you will later be able to change your own styles by replacing the macro values. The syntax for using a macro in a CSS definition is "<#macroname>", and then you need to add it to the [CssTags](#) array.

3. Normally methods 1. and 2. should be enough to handle most needs, but if you need a completely new way to display your tags, you can just define your own [SheetSelector](#) by deriving it from the class [TSheetSelector](#). You will need to override the abstract methods in this class to provide the behavior you need. In fact, this is the way [TStandardSheetSelector](#) is defined, and you can look at its code when creating your own [SheetSelector](#) class.

Customizing the Fonts

Fonts used in the HTML file are the same that the ones used in the xls/x file, and this might bring some compatibility issues if you expect your file to be shown in a website where it can be opened by people all over the world.

So it is recommended that you stick to standard fonts, like Arial or Times New Roman on the xls/x files you want to export. But if the files already exist and have non standard fonts, you can use the [FlexCelHtmlExport.HtmlFont](#) event to convert the fonts to standard typefaces. You can search for an example on this in the [Export HTML](#) demo.

IMPORTANT

The quote character FlexCel uses in style tags is the single quote ('). So, if you need to quote your font names because they have spaces, use a double quote (") so the style declaration is not affected.

For example, a style declaration in FlexCel might be: style = 'font-decoration: "my font"; '

You need to use double quotes around "my font" so it does not end the style declaration.

Export to HTML and FlexCel recalculation

Look at the notes in the [PdfExportingGuide](#) on the recalculating issues, this applies to HTML too.

Using Relative Hyperlinks

You can create links in Excel, both inside cells or on images, and they will be exported to html. But there is a small issue in that in Excel you can only enter absolute URLs. For example, you can enter:

"<https://www.tmssoftware.com/site/flexcelnet.asp>", but you cannot enter just [site/flexcelnet.asp](#). If you want your websites to be "site independent", you need to replace those URLs with relative ones.

You can do this using the [FlexCelHtmlExport.BaseUrl](#) property. If for example you define "BaseUrl" to be "<https://www.tmssoftware.com/>", whenever a link starts with that text, the text will be removed. In our example, the link:

"<https://www.tmssoftware.com/site/flexcelnet.asp>" will be converted to

"site/flexcelnet.asp"

Exporting the images as SVG

When using HTML5, FlexCel offers the option to export the images in the file as SVG instead of png/jpeg. This won't change much for standard images, but other objects like charts or autoshapes can be exported in a vectorial format instead of rasterizing them to pngs. SVG objects can also be smaller if embedded in the page.

SVG is well understood by all modern browsers, so it might be a good idea to turn SVG exporting of images on. To do it, change the [FlexCelHtmlExport.SavedImagesFormat](#) property to [THtmlImageFormat.Svg](#).

FlexCel iOS Guide

Introduction

The FlexCel code you have to write for iOS itself is very similar to normal FlexCel for Windows code.

This is for example the code needed to create a file in Windows:

```
static void CreateFile()
{
    XlsFile xls = new XlsFile(1, true);
    xls.SetCellValue(1, 1, "FlexCel says hello!");
    xls.Save("result.xlsx");
}
```

And this is the code needed to create the same file in iOS:

```
static void CreateFile()
{
    XlsFile xls = new XlsFile(1, true);
    xls.SetCellValue(1, 1, "FlexCel says hello!");
    xls.Save("result.xlsx");
}
```

There are really not many differences, and this applies to most FlexCel code you can write: It is exactly the same. So why are we covering iOS in a separate document? What else can we say that is not covered in the other documents?

Well, while most FlexCel code will be the same, there is a fundamental difference between iOS and Windows: **Files in iOS are in a sandbox**. You can't just open a file in "My Documents" and save it in another place in the disk. Actually, you can't access *any* file outside the folder where your application is installed. How to deal with the file sandbox is what we are going to cover on the rest of this file.

The document Sandbox

When working in Windows, applications can access almost any file in the hard drive. Which is a nice thing from a usability point of view, but a complete nightmare from a security point of view. Imagine you download an application from the internet, how do you prevent it from encrypting all the documents in your hard drive and then asking for some ransom money in order to decrypt them again?

For this reason, in iOS your application can only read and write to the folder where it is installed or its subfolders. This gives you the added advantage that when you uninstall the app it is gone completely, as it can't leave garbage all over your hard disk. But on the other side, how do you work with a restriction like this? How do you create a file in Excel, open it with FlexCel, modify its values and give it back to Excel, if FlexCel and Excel can't see each other at all?

The first way to share things is for special files: Apps can access certain other files such as address book data and photos, but only through APIs specifically designed for that purpose. But this isn't a general solution, and while it might work for images, it won't work for xls/x or PDF files.

The solution for more general files comes in 2 parts:

1. In order to do anything useful with the xls/x, pdf, or html files FlexCel can generate, you need to **Export** them to other apps.
2. To be able to read files from other apps like Dropbox or the email, you need to **Import** the files from the other apps.

With this Import/Export system, your application can't open any file that wasn't given to it. In order to open a file, the user needs to export it to your app.

A look at some of the available folders for your application

Before we continue, and having established that you can't write to any folder in the device, let's look at the folders where you can read or write:

- **<Application_Home>/Documents/** This is where you normally will put your files. Backed up by iTunes.
- **<Application_Home>/Documents/Inbox** This is where other apps will put the files they want to share when exporting to your app. **Read only**. Backed up by iTunes.
- **<Application_Home>/Library/** This is for the files that ship with your app, but not for user files. You could for example put xls/x templates here.
- **<Application_Home>/tmp/** The files you write here might be deleted when your app is not running. Not backed up by iTunes.

Those are at a glance the most important folders you need to know about. You can get a more complete description of the available folders in the [Apple documentation](#).

Importing files from other apps

Registering your app with iOS

In order to be able to interact with files from other applications, you need to register your application as a program that can handle the required file extensions. To do so, you need to modify the file Info.plist in your app bundle.

For handling xls or xlsx files, you would need to add the following to your Info.plist:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>CFBundleDocumentTypes</key>
<array>
  <dict>
    <key>CFBundleTypeName</key>
    <string>Excel document</string>
    <key>CFBundleTypeRole</key>
    <string>Editor</string>
    <key>LSHandlerRank</key>
    <string>Owner</string>
    <key>LSItemContentTypes</key>
    <array>
      <string>com.microsoft.excel.xls</string>
      <string>com.tms.flexcel.xlsx</string>
      <string>org.openxmlformats.spreadsheetml.sheet</string>
    </array>
  </dict>
</array>

<key>UTExportedTypeDeclarations</key>
<array>
<dict>
  <key>UTTypeDescription</key>
  <string>Excel xlsx document</string>
  <key>UTTypeTagSpecification</key>
  <dict>
    <key>public.filename-extension</key>
    <string>xlsx</string>
    <key>public.mime-type</key>
    <string>application/vnd.openxmlformats-officedocument.spreadsheetml.sheet<
/string>
  </dict>
  <key>UTTypeConformsTo</key>
  <array>
    <string>public.data</string>
  </array>
  <key>UTTypeIdentifier</key>
  <string>com.tms.flexcel.xlsx</string>
</dict>
</array>
</dict>
</plist>

```

You can either do this manually by entering the xml in a code editor (info.plist is just an xml file), or add this from Visual Studio, by clicking in info.plist:

Toolbox

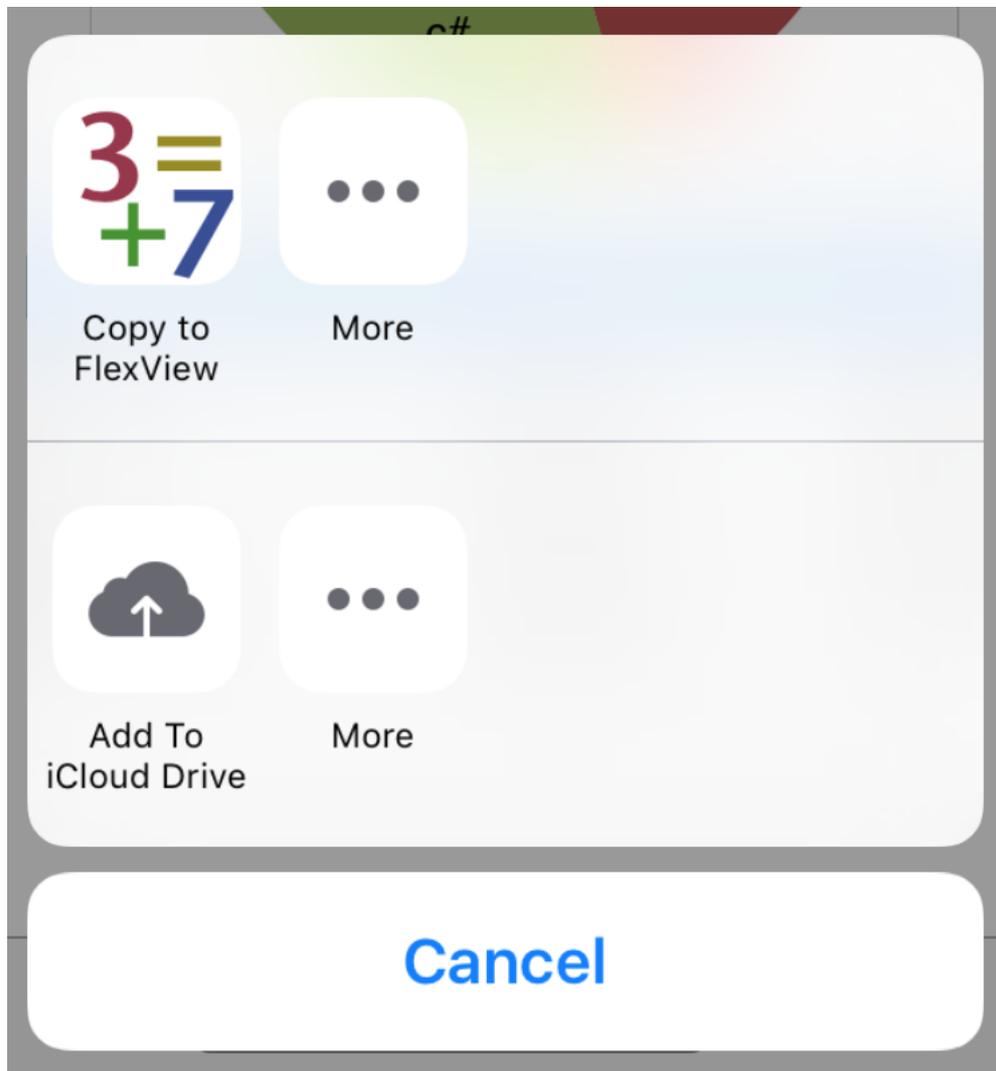
Info.plist [X]

iOS Settings Bundle

Property	Type	Value
Document types	Array	(1 item)
	Dictionary	(4 items)
CFBundleTypeName	String	Excel document
CFBundleTypeRole	String	Editor
LSHandlerRank	String	Owner
Document content type UTIs	Array	(3 items)
	String	com.microsoft.excel.xls
	String	com.tms.flexcel.xlsx
	String	org.openxmlformats.spreadsheetml.sheet
Exported Type UTIs	Array	(1 item)
	Dictionary	(4 items)
UTTypeDescription	String	Excel xlsx document
UTTypeTagSpecification	Dictionary	(2 items)
public.filename-extension	String	xlsx
public.mime-type	String	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet

You can find step-by-step information on how to register your app in the [iOS tutorial](#).

When you configure everything, your app will appear in the "Open in" dialog from other applications on your phone:



Answering to an “Open in” event

Once you’ve registered your application as an app that can handle xls/x files, it will appear in the other application’s “Open in” dialogs. When the user clicks in your app icon, iOS will copy the file to <Your app folder>/Documents/Inbox (See [“A look at some of the available folders for your application”](#) above).

After that iOS will start your app, and send it an `application:OpenUrl:` message so you can actually open the file. So in order to do something useful, you will need to listen to `application:OpenUrl:` event.

Luckily this is straightforward; just open the file `AppDelegate.cs` in your project and add the following lines:

```
public override bool OpenUrl(UIApplication application, NSURL url, string sourceA
application, NSObject annotation)
{
    DoSomething(url);
}
```

NOTE

OpenURL would get the URL of the file (something like "file://localhost/folder...") instead of a filename. The [FlexCelView example](#) shows how to convert the URL into a path, and how to handle an OpenUrl event.

Exporting files to other apps

Exporting files to other apps is the reverse of what we've seen in the previous section: Now we want to show a dialog where we show the user all the applications that can open the file we generated.

You can export a file with the following code:

```
partial void ShareClick(MonoTouch.UIKit.UIBarButtonItem sender)
{
    if (PdfPath == null) return;
    UIDocumentInteractionController docController =
        new UIDocumentInteractionController();
    docController.Url = NSURL.FromFilename(PdfPath);
    docController.PresentOptionsMenu(ShareButton, true);
}
```

Printing from iOS

To print an xls or xlsx file created by FlexCel, you need to export it to PDF first (using [FlexCelPdfExport](#)). Once the file is in pdf format and saved in your hard drive, just export it to the user as in the sample above.

And the "Print" button will appear among the other options inside the share sheet.

Backing up files

A note about the files you use with FlexCel. Not all of them might need to be backed up, and Apple considers it **a reason for App Store rejection** if your application is backing up static files (as this will increase backup times and sizes for all users).

If you are using xls or xlsx files as templates for your app, but they aren't actual data and shouldn't be backed up, you should use the `NSURLsExcludedFromBackupKey` or `KCFURLsExcludedFromBackupKey` properties to exclude them from backup.

You can find more information about this topic at: https://developer.apple.com/library/ios/#qa/qa1719/_index.html

Other ways to share files

Besides exporting and importing files, there are two other ways in how you can get files from and to your application:

iTunes file sharing

Your application can offer “Share in iTunes” functionality. To allow it, you need to add the key:

```
<key>UIFileSharingEnabled</key>  
<true/>
```

To your Info.plist file. Once you add this entry, your app will appear in iTunes and the user will be able to read and write documents from the “Documents” folder of it. The interface is kind of primitive, but it gets the work done.

NOTE

Again as in the case of registering the application to consume some file types, the Delphi IDE doesn't allow you to do it directly. You need to add a Boolean key, and Delphi will only add string keys. So, again you will have to create a different Info.plist and merge it, as we did in [“Registering your app with iOS”](#). If you are already doing so to registering files to import, then you can use that same file to add this entry.

WARNING

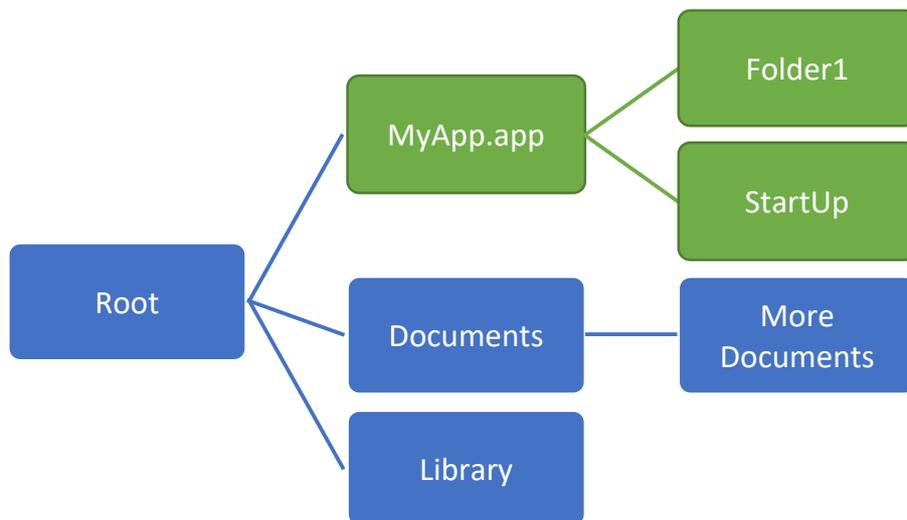
If you decide to enable iTunes sharing for your app, make sure that the documents in the “Documents” folder are actual documents the user cares about, and put the others somewhere else, like in “Library”. Failing to do so can result in a rejection from the App store. (as you can see here: <http://stackoverflow.com/questions/10767517/rejected-by-uifilesharingenabled-key-set-to-true>)

Copying files on startup

The last way to put files in your app is to copy them from your app bundle to the “Documents” folder when your app starts

Note that you can only put files **inside** your app bundle, you can't put a file directly in the “Documents” folder. Because you will not be uploading the Documents folder to the app store; you will upload only the app bundle.

If your application is called “MyApp” and the folders look something like this:



Then you can only put files in the green folders in the diagram. This is because "MyApp.app" is what will be distributed to the App store, and what your users will download when they download your app.

So, how do we put a file on the blue folders in our distribution package? The usual technique is iOS is to copy them to some folder inside MyApp.app, and on startup of your application, copy those files to "/Documents" or "/Library"

A note about Encodings

When you create a Xamarin iOS or Android application, by default it will come with a limited number of encodings. This is to keep application size small. Those encodings include for example ASCII and Unicode, but no Win1252 (encoding used in western Windows machines) or IBM 437 (encoding used in zip files).

FlexCel will work in most cases with the reduced number of encodings, but there are some rare cases where we need the full list of encodings. An example is when reading an Excel 95 file, and there are a couple of other cases more.

So in order to not have problems with non-existing encodings, it might be a good idea to click in your project properties and add "west" encodings.

- In iOS: Go to Build->iOS Build and select the "Advanced" tab. There in the "Internationalization" section choose "west".
- In Android: Go to Build->Android Build and select the "Linker" tab. There in the "Internationalization" section choose "west".

If you are worried about the extra size you can skip this step. FlexCel will still work in most of the cases without the extra encodings.

FlexCel Android Guide

Introduction

As it is the case with [iOS](#) and most of the platforms we support, the FlexCel code you have to write for Android itself is very similar to the code for normal "Full-Framework" .NET apps in Windows. We've reused most of the code from FlexCel in our Android implementation, and you can still use APIMate, the Demos, everything that you could in Windows when coding for Android.

Fonts

The biggest difference with Windows apps is that Android by default has a single font: "**Roboto**". When exporting documents to PDF the lack of extra fonts can be a problem.

There are multiple ways to work around this:

1. Replacing all fonts in the document by internal fonts.

If you are **only using ANSI Windows-1252 characters** and a single font, and you don't care about the exact font used, you can use the code:

```
FlexCelPdfExport.FontMapping = TFontMapping.ReplaceAllFonts;
```

before exporting to PDF.

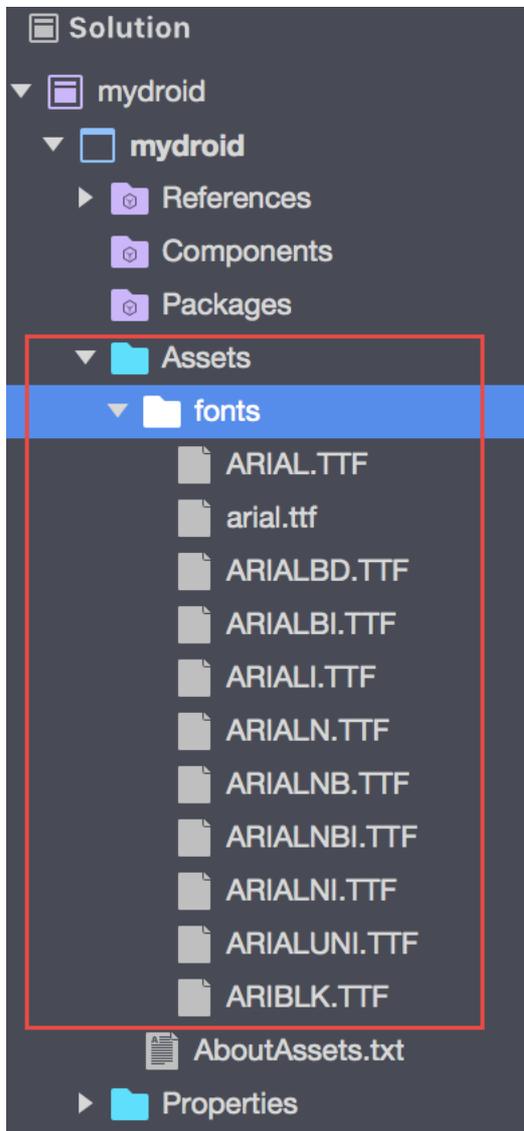
A PDF viewer is required to understand 3 basic fonts: A Sans-Serif (Arial or Helvetica like), a Serif (Times or Times New Roman like), and a proportional font (Courier or Courier New like).

So if you don't embed any font and are fine with using the internal fonts, you can get a smaller file and forget about other issues by replacing the fonts by internal ones. **However, note that this will only work if you are not using characters outside of the ANSI Windows-1252 range:** The internal fonts don't support Unicode, so if you have Unicode characters, you need to provide a font that has the needed characters.

2. Deploying the fonts with your app.

You can provide FlexCel the fonts you need in the place where FlexCel expects them. The simplest way to do that is to deploy the font files needed as internal assets in your application.

To do it, in the project manager of Visual Studio, right click in the Assets folder and create a "**fonts**" folder behind it:



WARNING

The path is case sensitive: "Fonts" instead of "fonts" will not work.

IMPORTANT

At the time of this writing there is a bug in Visual Studio 2017 which will raise an error * **MSB3025**: The source file Assets/fonts is actually a directory* when you add a folder to the assets.

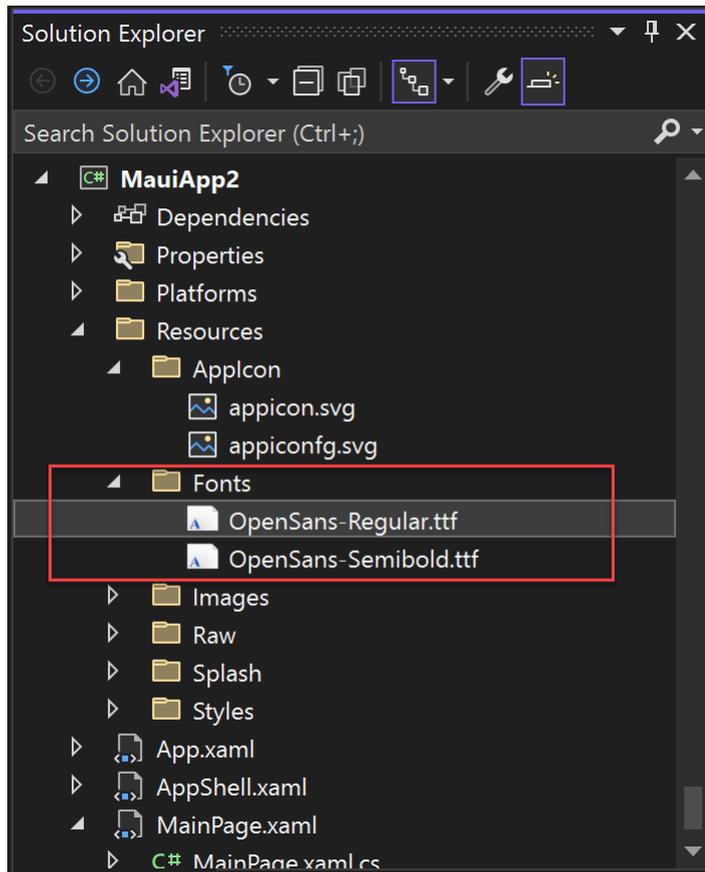
If you are getting this error, you need to manually edit the csproj after adding the fonts folder to the Assets, and remove the line:

```
<AndroidAsset Include="Assets\fonts\" />
```

For more information take a look at <https://forums.xamarin.com/discussion/87052/compiling-error-when-assets-folder-containing-subfolder>

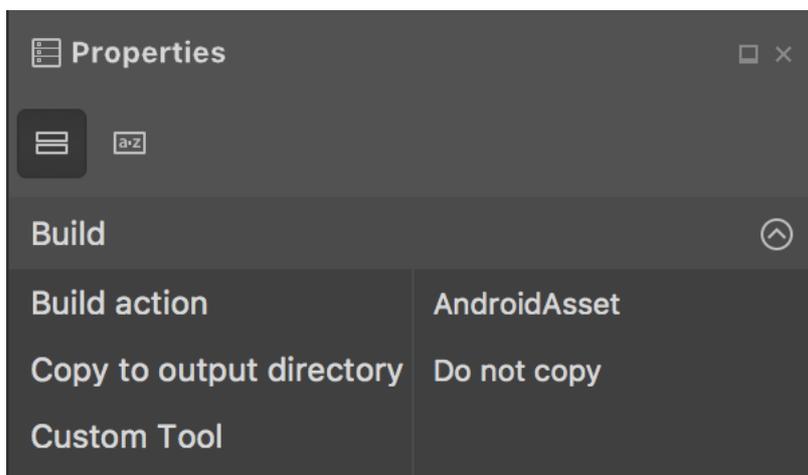
NOTE

If you are using .NET Maui, there is no need to setup the Assets. You can copy the fonts directly in Resources/Fonts:



And FlexCel will pick them from there. This way you can use the fonts in both FlexCel and your app.

Also note that all assets should have their "Build action" set to "AndroidAsset":



IMPORTANT

Make sure you have the rights to distribute the fonts that you deploy.

3. Providing the font data via an event.

If you deploy your fonts to a "fonts" folder inside your app assets as explained in the point above, then you have nothing more to do.

But if your app already deploys the fonts to some other folder, you can use the **FlexCelPdfExport.GetFontFolder** event to change where FlexCel will look for the fonts. For example, if your fonts happen to be inside a folder in your assets named Shared Fonts, you could use the following code:

```
using (FlexCelPdfExport pdf = new FlexCelPdfExport(xls, true))
{
    pdf.GetFontFolder += (sender, e) =>
    {
        e.FontPath = "@Shared Fonts";
    };

    pdf.Export("result.pdf");
}
```

Take a look at the "@" sign the code above. To use fonts that you included as assets, you need to return a string starting with a "@" sign in the GetFontFolder event.

So for example to specify that the fonts are in the "fonts" folder asset, you would return "@fonts" on the OnGetFontFolder event. ("@fonts" is the default used by FlexCel and covered in point 2 of this document).

TIP

Of course you can specify any folder in the Android device, not only assets. If the fonts are not in assets, just specify the full folder without a starting @.

A note about Encodings

When you create a Xamarin iOS or Android application, by default it will come with a limited number of encodings. This is to keep application size small. Those encodings include for example ASCII and Unicode, but no Win1252 (encoding used in western Windows machines) or IBM 437 (encoding used in zip files).

FlexCel will work in most cases with the reduced number of encodings, but there are some rare cases where we need the full list of encodings. An example is when reading an Excel 95 file, and there are a couple of other cases more.

So in order to not have problems with non-existing encodings, it might be a good idea to click in your project properties and add "west" encodings.

- In iOS: Go to Build->iOS Build and select the "Advanced" tab. There in the "Internationalization" section choose "west".
- In Android: Go to Build->Android Build and select the "Linker" tab. There in the "Internationalization" section choose "west".

If you are worried about the extra size you can skip this step. FlexCel will still work in most of the cases without the additional encodings.

Saving files

Android has 2 types of storage: "Internal storage" and "External storage". While historically internal storage might have been in the device and external outside it, today there is normally no such difference. What makes internal different from external is the permissions: Your app has full permissions to the internal storage, and those files belong to your app. If you uninstall the app, the files on its internal storage will be deleted with it. On the other side, you need permissions for external storage, and those files are available for other apps too.

Internal storage

For internal storage there is not much more to say: You can create, modify and delete those files as you like. To get the path to the internal storage from your code, you can use

[System.Environment.GetFolderPath\(System.Environment.SpecialFolder.Personal\)](#)

If using [Xamarin Essentials](#), you can also use [Xamarin.Essentials.FileSystem.AppDataDirectory](#)

NOTE

[Xamarin Essentials](#) and the .NET API shown above are cross-platform, so they can also be used in for example iOS and they will point to similar folders there. On the other hand, there are Android-specific APIs like `Application.Context.FilesDir` or `Android.Content.Context.GetExternalFilesDir(string type)` which will only work in Android.

External storage

External storage can on its own be divided into 2 different types: **Private** and **Public** external storage. Both are publicly available for the rest of the apps, but conceptually private files are expected to be used by your app, while public files are to be freely shared.

To save a file in external storage, you can use

- For private external storage: [Android.Content.Context.GetExternalFilesDir\(string type\)](#)
- For public external storage:
[Android.OS.Environment.GetExternalStoragePublicDirectory\(string directoryType\)](#)

NOTE

The APIs for external storage are not cross-platform, likely due to the fact that external storage is an Android-only concept.

You can find a good description of external storage here : <https://docs.microsoft.com/en-us/xamarin/android/platform/files/external-storage>

Also, you can check <https://developer.android.com/training/data-storage/files> for more information about the kind of folders where you can save the data.

Getting permissions to external storage

In order to read or save a file into external storage, you first need to get permissions from Android. In older Android versions, adding those permissions to the manifest was enough. But if you target Android 6 or newer, you also need to dynamically ask for the permissions when you want to use them.

You can find the code needed to ask for the permissions here:

<https://docs.microsoft.com/en-us/xamarin/android/app-fundamentals/permissions>

Sharing files

A tale of two APIs

A long, long time ago, the way to share files in Android was to store the file in external storage and then call an intent to share the file. This had multiple problems, one of those being that both the app sharing the file and the app consuming it should have permissions to read/write external storage.

To fix this, Android introduced the [FileProvider](#) class which was made available via a [support library](#) to Android 4 (Ice Cream Sandwich) and later.

But there was little reason to change the FileProvider API, since the old API was still working despite its problems. That changed with Android 7 (Nougat), which deprecated the old way to share files: <https://developer.android.com/about/versions/nougat/android-7.0-changes#perm>

And still, there was no need to change the old ways: you could still target an older Android version and have it run in Android 7 without issues. The new behavior only happened if you explicitly targeted Android 7.

NOTE

The versions in Android can be a little confusing, and it makes sense that we clarify them a little before continuing.

The **target** version in Android is **not the minimum Android version your app will run**. The minimum version is given by **minSdkVersion**. The target version instead is the version your app is optimized for. You can target Android 7 and still have a minimum version of Android 4, but if you target Android 7, the Android 7 rules will apply to your code so the old way to share files won't work. If you target Android 4 (and have a minimum version of Android 4 too), then the Android 4 rules will apply, even if you are running in Android 7.

This all means that if you target Android 4 and have a minimum SDK of Android 4, you don't have to care about the new Android 7 rules, and your app will still work in Android 7. If you target Android 7, even if your minimum supported version is still 4, you have to use the new way to share files.

You can read more about Android versioning here: <https://developer.android.com/guide/topics/manifest/uses-sdk-element#ApiLevels>

Now, going back to our story: in August 2018 this all changed again. Google started to [ask for a target version of 8.0 or newer](#) to be able to publish in the Play store. When you set a target version of 8.0 (no matter if your minimum version is lower), then the old-style way to share files won't work anymore in devices running Nougat or later. Old tricks won't work anymore, and it is time to change to the new system.

IMPORTANT

Since Android 9, the [support library](#) is also deprecated, and now FileProvider is at <https://developer.android.com/reference/androidx/core/content/FileProvider>

So you need to decide what to use: If the not-deprecated androidx which only supports Android 9 or newer, or the old support library which isn't updated anymore but supports older devices.

In the examples bundled with FlexCel we will show AndroidX, but note the differences with Support Library. If you want a working example with Support Library, you can find it in GitHub: <https://github.com/tmssoftware/TMS-FlexCel.NET-demos/releases/tag/v7.1.2.1>

Converting to AndroidX

To convert your application to AndroidX, you need to install the [AndroidX nuget package](#)

You can find more information here: <https://docs.microsoft.com/en-us/xamarin/android/platform/androidx>

Once you've done that, you need to adapt the FileProvider from `android.support.v4.content.FileProvider` to `androidx.core.content.FileProvider` as shown below.

Using FileProvider to share files

To share files in the new way, you need to define a [FileProvider](#).

You can read about the concepts here: <https://developer.android.com/training/secure-file-sharing/>

And in the rest of this section we will discuss how to share a simple file created by FlexCel. You can find a finished working example here: [LangWars](#)

Step 1. Install Xamarin Essentials.

In order to have access to the functionality we need, you will have to install [Xamarin Essentials](#) from [nuget](#).

Step 2. Declare the file provider.

You can do this in two different ways:

- You can manually edit your `properties/AndroidManifest.xml` file, and add the following `<provider>` below the `<application>` tag:

```

<application ...>
...
<provider android:name="androidx.core.content.FileProvider"
    android:grantUriPermissions="true"
    android:exported="false"
    android:authorities="${applicationId}.fileprovider">
    <meta-data android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/file_provider_paths" />
</provider>
</application>

```

NOTE

To use support library instead of AndroidX, set the provider to be:

```
<provider android:name="android.support.v4.content.FileProvider"
```

Also note that it is always "android.support.FILE_PROVIDER_PATHS", even in AndroidX

- If you don't want to manually edit AndroidManifest.xml, you can use the [ContentProviderAttribute](#) instead. You need to define a class that inherits from `AndroidX.Core.Content.FileProvider` and add an attribute like:

```

[ContentProvider(
    new[] { FileProviderAuthority },
    Name = "langwars.custom.FileProvider",
    GrantUriPermissions = true,
    Exported = false
),
MetaData(
    name: "android.support.FILE_PROVIDER_PATHS",
    Resource = "@xml/file_provider_paths"
)
]
public class LangWarsCustomFileProvider: AndroidX.Core.Content.FileProvider
{
}

```

> [!Note]

> To use support library instead of AndroidX, set the class to derive from:

```
> public class LangWarsCustomFileProvider: Android.Support.V4.Content.FileProvider
```

Where you would have to replace "FileProviderAuthority" with a constant that uniquely identifies your provider among *all* installed applications. Normally you use "your_app_id.fileprovider" here, and whatever name you use, it is the name you will have to use below in step 4. In method 2.1, we used "\${applicationId}.fileprovider" as value for the authority.

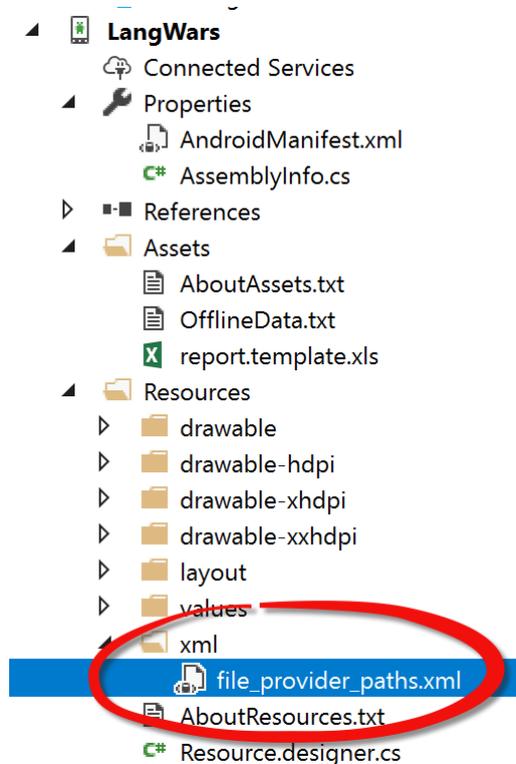
The second approach can be cleaner in that it doesn't require to manually modify the app manifest, but on the other hand it needs to define a new custom provider when you could just use the built-in FileProvider. In our demo, we choose to simply modify the manifest.

Step 3. Declare the paths that the FileProvider can use.

In step 2 we defined a resource:

```
Resource = "@xml/file_provider_paths"
```

Now we need to create this resource, where we will tell Android which are the paths that the provider will use. To do so, we create an "xml" folder under the "Resources" folder, and inside that folder, we create a file named **file_provider_paths.xml**



For our example, we are going to store the file in the root data folder in the internal memory, so we will add this to the provider paths. The file **file_provider_paths.xml** will be:

```
<?xml version="1.0" encoding="utf-8" ?>
<paths>
  <files-path name="files" path="." />
</paths>
```

If you need to share a file in external storage or on a different path, you should change this file accordingly. For more information about what you can write in this file, please read <https://developer.android.com/reference/androidx/core/content/FileProvider#SpecifyFiles>

Step 4. Write the code.

Once the provider is correctly declared, the last step is to write the code to share the file. The code we used in the example is as follows:

```
Intent Sender = new Intent(Intent.ActionSend);
Sender.setType(StandardMimeType.xls);
Java.IO.File xlsFile = new Java.IO.File(TempXlsPath);
var contentUri = AndroidX.Core.Content.FileProvider.GetUriForFile(
    this,
    ApplicationContext.PackageName + ".fileprovider",
    xlsFile);

Sender.putExtra(Intent.ExtraStream, contentUri);
Sender.setFlags(ActivityFlags.GrantReadUriPermission);
StartActivity(Intent.CreateChooser(Sender, "Select application"));
```

The authority we used for `GetUriForFile` (`ApplicationContext.PackageName + ".fileprovider"`) is the same as the authority we defined in step 2.

NOTE

If you get a `Java.Lang.NullPointerException` error when trying your app, you most likely didn't define the provider right. Make sure that the name of the provider in the xml definition is the same as the name you use in your code, and that the provider is correctly embedded in the manifest.

Using FlexCel with Windows Phone, Store and Universal

There are currently two ways to use FlexCel with Universal apps:

1. Using the nuget package or the library at lib\uwp10. This requires Windows 10, and doesn't support the Universal style in windows 8.
2. Using .NET Core.

Same as with iOS and Android, the code needed to use FlexCel in Windows Phone or Universal Windows Platform (UWP) apps is very similar to the code needed to use FlexCel in Desktop .NET.

1. FlexCel for UWP 10

FlexCel for UWP 10 requires Windows 10 and won't run in Windows 8.1.

There isn't much to add about FlexCel for UWP 10. It works the same as always and you can reuse the code you wrote for other platforms.

2. FlexCel for .NET Core

.NET Core is another platform which is positioning itself as a better option for cross-platform development. The main advantage (and disadvantage) of using FlexCel for .NET core over FlexCel for UWP is that FlexCel for .NET Core uses the SKIA library for graphics instead of the built-in graphics. This is an advantage because SKIA is more complete than the native graphics available in UWP, and it is a disadvantage because you need to ship SKIA together with your app, and this will add tens of megabytes to the distribution.

About the Assembly to reference

The assemblies you need to reference are available in different places depending on which platform you choose to use:

1. FlexCel for UWP 10 is available in the folder lib\uwp10, or available in the FlexCel NuGet package.
2. FlexCel for .NET Core only available in the FlexCel NuGet package.

Encodings

Depending on the exact platform you are using, you might get an error of missing encoding when calling FlexCel.

With the **FlexCel NuGet package** encodings are automatically added through NuGet, so you should see no issues.

If you are manually referencing the portable dll you should do the following to avoid Encoding issues:

1. Add a reference to the System.Text.Encoding.CodePages NuGet package in your app at: <https://www.nuget.org/packages/System.Text.Encoding.CodePages/>
2. Somewhere in the initialization of your app, register the encodings by calling:

```
Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);
```

Using FlexCel with Mono

FlexCel is fully managed code without any external dependencies, and this makes it possible to run in different platforms besides Windows and .NET. It can be used via Mono or .NET Core in a Linux server, or on macOS, iOS and Android via Xamarin. In general, code should be 100% compatible between .NET and Mono, but there are some points worth noting.

Installing

For developing for Linux under MonoDevelop, you will have to manually copy FlexCel.dll to your development machine and add a reference to the assembly. Or if you prefer to develop in Visual Studio, then you don't need to do anything. Just compile your app in Windows under Visual Studio, and copy the files to the Linux server. There is no need to recompile under Mono, as Mono can read the dlls that Visual Studio generates.

Fonts

Probably one of the biggest problems you might get when using FlexCel in anything but Windows or macOS is the lack of fonts. Most of the default fonts in Excel are only present in Windows, and when you want to create a PDF file with them you will get a file that looks wrong. FlexCel needs to read the fonts to know the metrics, where the next character will go, where the line breaks are, etc.

We discuss how to deal with fonts when exporting in the [Font Management section of the Pdf exporting guide](#); make sure you read it if you are thinking in deploying to Mono.

Bugs

In a perfect world we wouldn't have to write this section. But this is not a perfect world, and while Mono is a high quality piece of work, it has bugs, as does .Net or FlexCel itself. The problem being, that while .NET has bugs, you have probably tested in .NET, and workarounded them. But Mono has a different set of bugs, and the only way to know which, if any, affect you, is to test in Mono. We do regular testing in Mono, and most tests pass ok. But you might find issues, so test heavily.

.NET Core

[.NET Core](#) is positioning more and more as the best way to move your applications to Linux. But right now it lacks some features you might need in a full featured app, and it is not as mature.

Depending on your needs, mono might still be a better choice today.

Using FlexCel with .NET Core and .NET 5 or newer

.NET Core is the next iteration of the .NET framework, and the latest release dropped the "Core" part, to become ".NET5". In this document, we will refer to ".NET5" and newer versions of the framework as ".NET Core" too, since they have the same root. FlexCel comes with full support for .NET Core. Being cross-platform, you can use FlexCel for .NET Core instead of the individual versions of FlexCel for iOS, FlexCel for Windows Universal, etc.

Selecting the graphics engine

By default FlexCel for .NET Core uses the Native OS graphics framework when available, or [SkiaSharp](#) otherwise. But you can change the behavior by changing the property [FlexCelConfig.GraphicFramework](#)

When deciding to use Native or SkiaSharp frameworks, you need to consider the following points. When using SkiaSharp:

1. Your app will have a dependency with the SKIA dll in SkiaSharp. This will make your application larger.
2. The rendered files will be a little different from the files generated using the OS.
3. For Android this is quite similar since the native graphics engine in Android is SKIA, but still, you won't be using the engine bundled with android. You will use the one in the SKIA dll from SkiaSharp.
4. For iOS and macOS, it is probably better to use the iOS/macOS FlexCel dlls directly, since they use CoreGraphics which is a very capable framework and there is no need to use SKIA.
5. For Windows, it is hard to say. The current implementation of FlexCel using System.Graphics (which uses GDI+) has better support for some things, like right to left languages, or EMF/WMF images. But on the other side, GDI+ [is not officially supported in a server](#) even when it works. So the choice is yours.
6. For Universal Windows Apps, it is probably better to use SKIA, even if it will make your applications larger. The graphics stack in UWA lacks a lot of functionality, and the SKIA implementation is more complete.
7. Your application won't be fully managed since it will call the native skia dll. This might have security implications.
8. Your application will work the same everywhere since it will use the exact graphics framework everywhere. The native components might have tiny differences in how they render the text or graphics depending on the underlying graphics engine.
9. Linux support can be problematic due to all the different distributions. See [Installing in Linux](#)
10. The native graphics engine of the OS will likely be always up to date, while if you ship SKIA with your app, it won't get updates until you update your app. As you can see, it is not a clear winner between using SKIA or the native framework in each platform. SKIA has the advantage of producing the same results anywhere, but the native engines might be more complete and you don't have to ship the SKIA framework with your application.

You can choose what you prefer.

Migrating from .NET 4 to .NET 5 or newer

While it is just a version number of difference, .NET 4 is based in the old .NET framework, and .NET 5 is based in .NET Core. This means that to move to .NET 5 from .NET 4, you need to use the NuGet package with .NET 5. By default, FlexCel will keep using the native OS graphic engines. But if you decide to use SkiaSharp, it can have some unexpected consequences in your app. Especially if you were using WMF/EMF images, which the SKIA framework can't render. So, if you want to keep your application running as similar as possible as the way it worked in .NET 4, the most important thing is to keep `FlexCelConfig.GraphicFramework` using the native framework. That should make the migration to .NET 5 seamless. But there is another big change in .NET 5 which affected the FlexCel codebase, and it might make sense to review that it doesn't affect yours. This is the fact that .NET moved from using NLS in Windows to use ICU (see <https://docs.microsoft.com/en-us/dotnet/standard/globalization-localization/globalization-icu>) This change can lead to subtle bugs, like this one: <https://github.com/dotnet/runtime/issues/43736> In our particular case, our main issue is that now this code:

```
static void Main(string[] args)
{
    int cmp = String.Compare("\u0007", "\u0008",
StringComparison.CurrentCultureIgnoreCase);
    Console.WriteLine(cmp);
}
```

returns 0, when in .NET 4 it would return -1. Now, all "invisible" characters are considered the same when comparing strings using linguistic order.

Installing in Linux

To run in Linux, with SKIA, you need to provide a native `libSkiaSharp.so` that works in your Linux distribution. For most usual cases, you just need to add a nuget dependency to <https://www.nuget.org/packages/SkiaSharp.NativeAssets.Linux.NoDependencies/> to your project. See also <https://github.com/mono/SkiaSharp/issues/312>

If your distribution isn't supported, then you need to manually compile Skia. The instructions to do so are here: <https://github.com/mono/SkiaSharp/wiki/Building-on-Linux>

Installing in a Docker container

Besides the section [Installing in Linux](#) above, make sure to read [Running FlexCel inside Docker containers](#)

Tutorials

In this section there is a list of tutorials to help you get started.

In this section:

[Creating an iOS application](#)

iOS Tutorial

Overview

In this tutorial we will create a small xls/x document viewer. The app isn't particularly useful or flashy, but it is designed to show most concepts you need to know in order to work with files in iOS

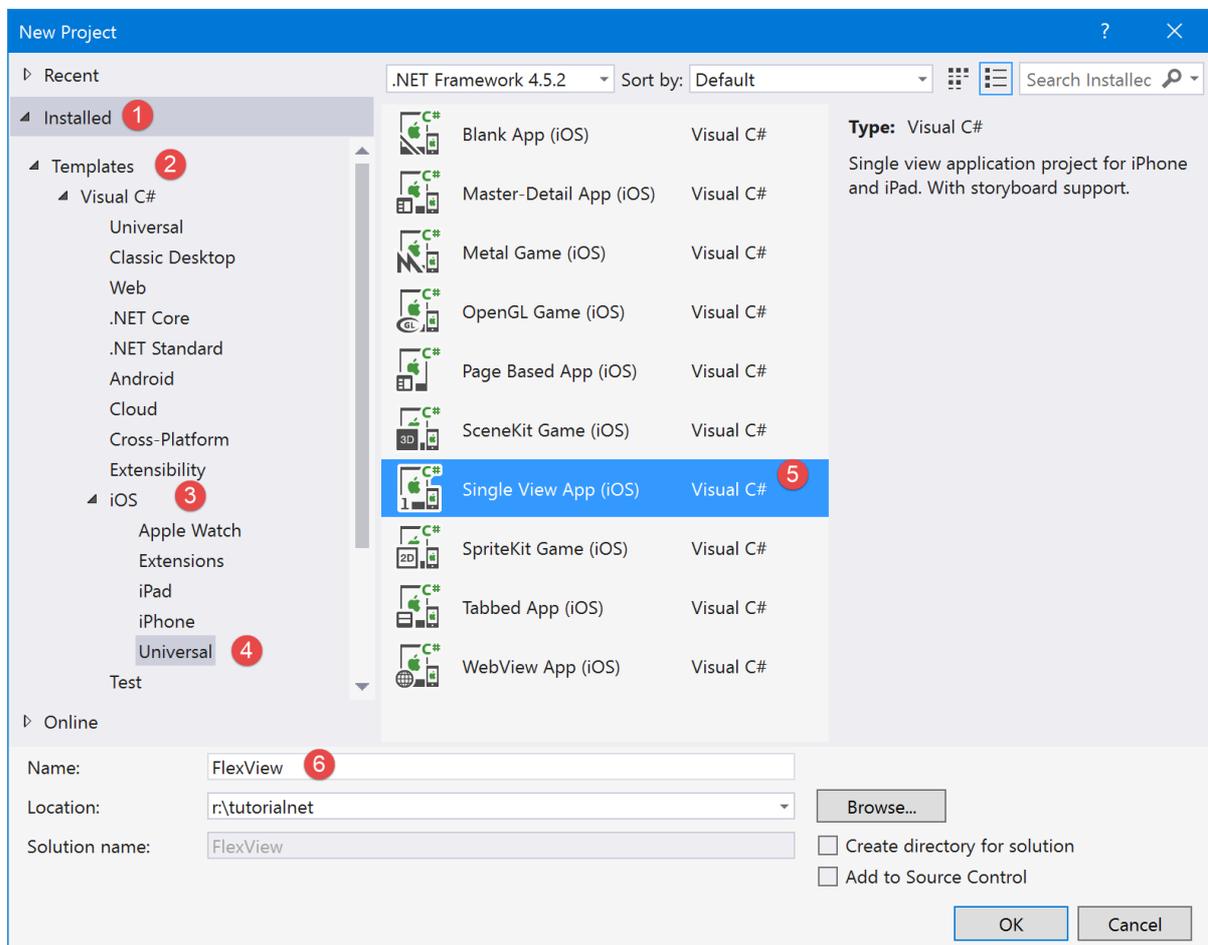
NOTE

The complete source for this tutorial is available as [FlexView demo](#) in the FlexCel distribution.

Step 1. Setting up the Application

Lets start by creating a new iOS Universal app:





- Select **Installed** (1), **Templates** (2), **iOS** (3) and then **Universal** (4)
- Select **Single View Application** (5)
- Give it a name (6). In this tutorial we will use **FlexView**.

Then you can go to the application properties, and set icons and the application name.

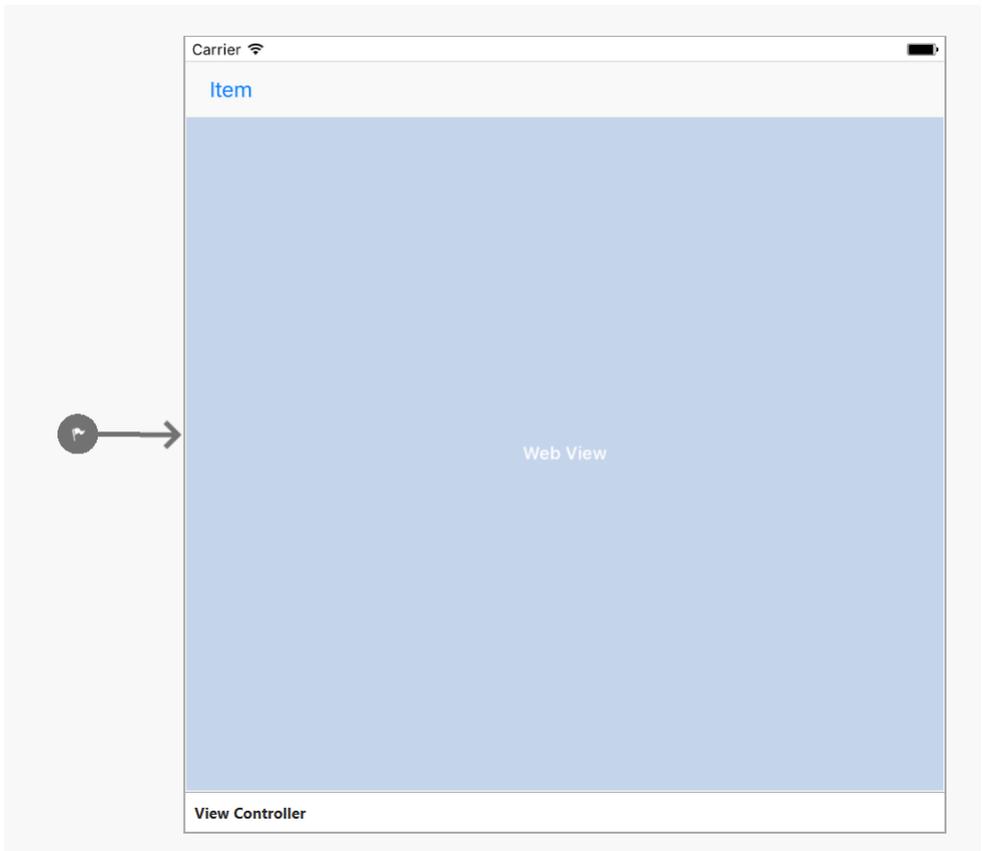
You might now try running the application, it should show as an empty form in the simulator or the device.

Step 2. Creating the User Interface

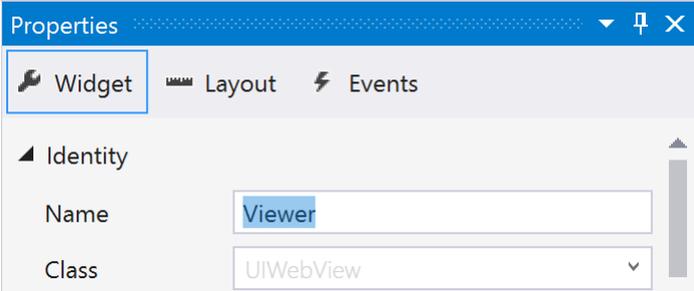
We want to display an Excel file into our application. For this, we will convert the file to pdf using FlexCel, and show the pdf output in a web browser.

Double click in the file Main.storyboard. Depending on your version of Xamarin or Visual Studio and on your preferences, the file might open in XCode or the Xamarin designer. We'll be showing the Xamarin Designer here, but the XCode steps are similar.

In the designer, drop a WebView and a Toolbar. Adjust the anchors so the WebView resizes with the window:

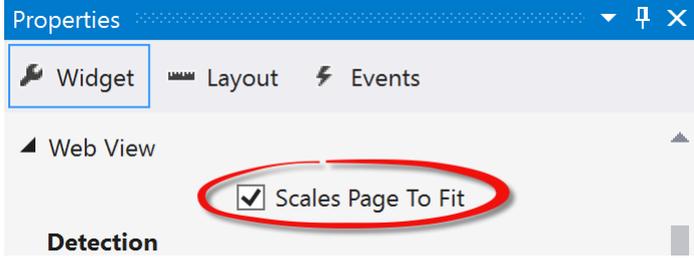


Name the WebView Viewer:



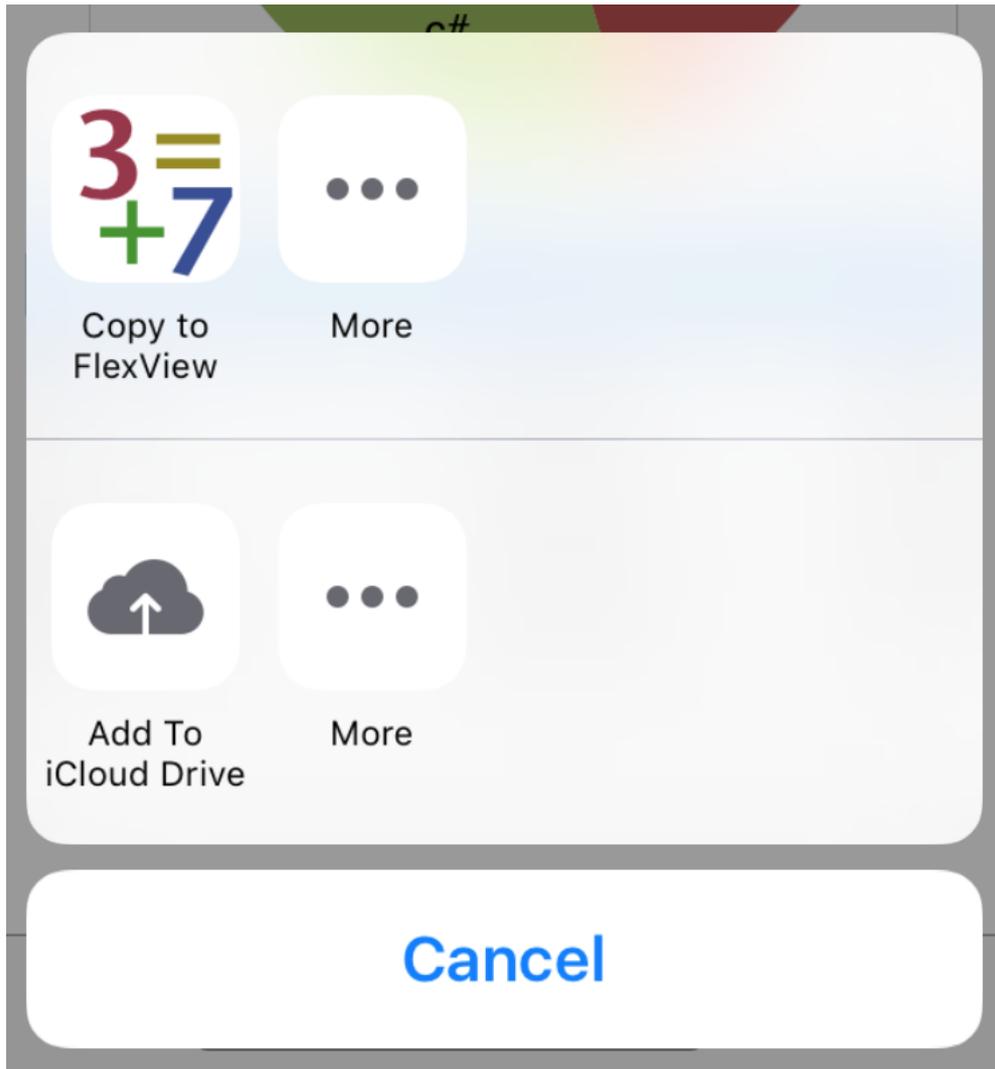
NOTE
If using XCode, create an Outlet for the webview named "Viewer" by ctrl-dragging the webview to the assistant view.

And finally, set **Scales Page To Fit** = true in the web properties to enable pinch to zoom:



Step 3. Registering the application

The next step is to tell iOS that our application can handle xls and xlsx files. This way, when another app like for example mail wants to share an xls or xlsx file, our application will show in the list of available options:



To register our app, we need to change the file Info.plist.

This can be done directly from the Xamarin Studio Info.plist editor, but for this example we'll just open the new generated Info.plist with a text editor, and paste the following text before the last `</dict>` entry:

```

<key>CFBundleDocumentTypes</key>
<array>
  <dict>
    <key>CFBundleTypeName</key>
    <string>Excel document</string>
    <key>CFBundleTypeRole</key>
    <string>Editor</string>
    <key>LSHandlerRank</key>
    <string>Owner</string>
    <key>LSItemContentTypes</key>
    <array>
      <string>com.microsoft.excel.xls</string>
      <string>com.tms.flexcel.xlsx</string>
      <string>org.openxmlformats.spreadsheetml.sheet</string>
    </array>
  </dict>
</array>

<key>UTExportedTypeDeclarations</key>
<array>
  <dict>
    <key>UTTypeDescription</key>
    <string>Excel xlsx document</string>
    <key>UTTypeTagSpecification</key>
    <dict>
      <key>public.filename-extension</key>
      <string>xlsx</string>
      <key>public.mime-type</key>
      <string>application/vnd.openxmlformats-officedocument.spreadsheetml.sheet<
/string>
    </dict>
    <key>UTTypeConformsTo</key>
    <array>
      <string>public.data</string>
    </array>
    <key>UTTypeIdentifier</key>
    <string>com.tms.flexcel.xlsx</string>
  </dict>
</array>

```

Once you have done this, if you run the application and have for example an email with an xls or xlsx file, you should see "FlexView" in the list of possible applications where to send the file when you press "Share".

Step 4. Reading the file sent by another application

If you tried the application after the last step, and pressed the “Open in FlexView” button, you will notice that FlexView starts, but the previewer is still empty. It won’t show the file that the other application sent.

What happens when you press the “Open in FlexView” button is that iOS will copy the file in the “Documents/Inbox” private folder of FlexView, and send an `OpenURL` event to our app. We need to handle this event, and use it to load the file in the preview.

Open `AppDelegate.cs` in Visual Studio, and write the following code:

```
public override bool OpenUrl(UIApplication application, NSURL url,
    string sourceApplication, NSObject annotation)
{
    return FlexViewViewController.Open(url);
}
```

TIP

You might just write “override” inside the `AppDelegate` class, and Visual Studio will show you a list of possible methods to override.

NOTE

In iOS, we are going to get the URL of the file, not the filename. For example, the URL could be: `'file://localhost/private/var/mobile/Applications/9D16227A-CB01-465D-B8F4-AC43D70C8461/Documents/Inbox/test.xlsx'`

And the actual filename would be: `'/private/var/mobile/Applications/9D16227A-CB01-465D-B8F4-AC43D70C8461/Documents/Inbox/test.xlsx'`

But while iOS methods can normally use an URL or a path, C# `FileStream` expects a path. This is why we need to convert the URL to a path, using the `url.Path`.

So now it is the time to do the actual work. We need to write the **`FlexCelViewViewController.Open(url)`** method, that will convert the file to pdf and display it on the browser.

For this, open the file `FlexCelViewViewController.cs`, and write the following uses at the start:

```
using System;
using System.Drawing;
using MonoTouch.Foundation;
using MonoTouch.UIKit;
using FlexCel.Render;
using FlexCel.XlsAdapter;
using System.IO;
```

And then type the following code inside the class:

```

NSURL XlsUrl;
string PdfPath;

public bool Open(NSUrl url)
{
    XlsUrl = url;
    return Refresh();
}
private bool Refresh()
{
    try
    {
        XlsFile xls = new XlsFile(XlsUrl.Path);
        PdfPath = Path.Combine(
Environment.GetFolderPath(Environment.SpecialFolder.InternetCache),
Path.ChangeExtension(Path.GetFileName(XlsUrl.Path), ".pdf"));

        using (FlexCelPdfExport pdf
                = new FlexCelPdfExport(xls, true))
        {
            using (FileStream fs = new FileStream(PdfPath,
                FileMode.Create))
            {
                pdf.Export(fs);
            }
        }
        Viewer.LoadRequest(new
            NSUrlRequest(NSUrl.FromFilename(PdfPath)));
    }
    catch (Exception ex)
    {
        Viewer.LoadHtmlString("<html>Error opening " +
            System.Security.SecurityElement.Escape(
                Path.GetFileName(XlsUrl.Path))
            + "<br><br>"
            + System.Security.SecurityElement.Escape(
                ex.Message)
            + "</html>", null);

        return false;
    }
    return true;
}

```

If you run the application now and press "Open in FlexView" from another application, FlexView should start and display the file. You should be able to scroll and pinch to zoom.

Notes on temporary files and memory usage

Normally when coding for a PC, you want to avoid temporary files. In the code above, we could have directly exported the xls file to a memory stream, and then use that memory stream to create an NSData object and load it directly to the web browser.

But when coding for a phone, things are different. In this case, the memory of the device is limited, and the flash storage is fast and huge (it has to be able to store music and videos).

So, if memory is our constraint, it makes more sense to create a temporary file and read from there. While FlexCel will keep the spreadsheet in memory when opening, it won't keep the pdf file, which is generated on the fly. So saving this pdf file to a temp place can reduce the memory usage a lot.

Another thing to notice is that we should not write this temporary file to the /Documents folder, because this isn't user data, and it shouldn't be backed up by iTunes. That's why we write it to Library/Caches

And the last thing we are missing here, is to remove the file once the WebView loaded it. While this isn't strictly necessary (iOS will remove files in this cache when it needs space), it is a good practice.

We could remove the file in the Viewer.LoadFinished event, but since we plan to share the file with other apps in the next steps, we will keep it longer. So we will delete the old file before writing a new one.

In FlexCelViewViewController.cs, add the following method:

```
private void RemoveOldPdf()
{
    if (PdfPath != null)
    {
        try
        {
            File.Delete(PdfPath);
        }
        catch
        {
            //do nothing, this was just a
            //cache that will get deleted anyway.
        }
        PdfPath = null;
    }
}
```

And call RemoveOldPdf() as the first line in the Refresh() method we wrote above. Note that we could have also saved always to the same filename, so we wouldn't need to worry about deleting files at all. But when sharing the file, the filename would be lost.

NOTE

While we won't cover this here, another advantage of using a temporary file is that if your app gets killed by the OS, you can restore the state from the temporary file when restarted.

Step 5. Modifying the file

For this step, we will be replacing all numbers in the file with random numbers and recalculating the file. While this doesn't make a lot of sense, it shows how you can modify a file.

In this particular case, the only difficulty is that we can't overwrite the original file at Documents/Inbox. It is read-only. So we will save it to the Caches folder as tmpFlexCel.xls or tmpFlexCel.xlsx depending on the file format.

Saving it with the same name allows us to not care about deleting the temporary file because it will always be the same. But on the other hand, we need to store the original name of the file so the generated pdf file has the right filename.

So we'll introduce a new variable:

```
string XlsPath;
```

And we'll replace the old **XlsUrl** for **XlsPath** in all places where it isn't dealing with the pdf file.

Finally, we'll add a button and write this on the event handler:

```
partial void RandomizeClick(MonoTouch.UIKit.UIBarButtonItem sender)
{
    if (XlsUrl == null) return;

    XlsFile xls = new XlsFile(XlsPath, true);
    //We'll go through all the numeric cells and make them random
    Random rnd = new Random();

    for (int row = 1; row <= xls.RowCount; row++)
    {
        for (int colIndex = 1;
            colIndex < xls.ColCountInRow(row);
            colIndex++)
        {
            int XF = -1;
            object val = xls.GetCellValueIndexed(row, colIndex, ref XF);
            if (val is double) xls.SetCellValue(row,
                xls.ColFromIndex(row, colIndex), rnd.Next());
        }
    }

    //We can't save to the original file, we don't have permissions.
    XlsPath = Path.Combine(
        Environment.GetFolderPath(
            Environment.SpecialFolder.InternetCache),
        "tmpFlexCel" + Path.GetExtension(XlsUrl.Path));

    xls.Save(XlsPath);
    Refresh();
}
```

This code should do the replacement.

Step 6. Sending the file to other applications

In [step 4](#) we saw how to import a file from another application. In this step we are going to see how to do the opposite: How to export the file and make it available to other applications that handle xls or xlsx files. We will also see how to print the file.

Luckily, this isn't complex to do.

Go to the UI designer, locate the "item" button in the toolbar, rename it "Share", and add the following code in the button event handler:

```
partial void ShareClick(MonoTouch.UIKit.UIBarButtonItem sender)
{
    if (PdfPath == null) return;
    UIDocumentInteractionController docController =
        new UIDocumentInteractionController();
    docController.Url = NSURL.FromFilename(PdfPath);
    docController.PresentOpenInMenu(ShareButton, true);
}
```

This should show a dialog in your app to share the file with other apps. And it might be what we want in many cases. But this dialog doesn't include the options to "Print", or "Mail", which might be interesting to show too.

To show the extended options, change the last line in the code above from `PresentOpenInMenu` to `PresentOptionsMenu`:

```
docController.PresentOptionsMenu(ShareButton, true);
```

NOTE

Exporting to pdf will show a "Print" option when sharing the file, allowing us to print it.

Step 7. Final touches

In this small tutorial we've gone from zero to a fully working Excel preview / pdf converter application. But for simplicity, we've conveniently "forgotten" about an interesting fact: Excel files can have more than one sheet.

Modifying the app so it allows you to change sheets isn't complex, on the FlexCel side you just need to use [ExcelFile.SheetCount](#) and [ExcelFile.GetSheetName\(sheetIndex\)](#) to get an array of the sheets, then use [ExcelFile.ActiveSheetByName](#) to set the sheet you want to display.

But while not complex, there is a lot of plumbing needed for that: we need to define a new view in the storyboard, we need to populate a table view with the items, and select the correct one when the user selects a new sheet. Sadly this is a lot of code, and would mean making this tutorial twice as big with little FlexCel code and a lot of UI code that you can get tutorials everywhere, and so we won't be showing how to do it here. It would make the tutorial much more complex and add very little to it.

Tips and Tricks

An assorted list of small and interesting stuff to help you get the most out of FlexCel.

We will be adding new tips regularly here, so make sure to check this page from time to time.

In this section:

Localized month names

Why you are not getting the same date strings as Excel. And while we are at it, we tell you what's a "genitive month name"

Cloud fonts

Excel is moving more and more to cloud fonts, and why that is bad news for the rest of us.

Painting a full sheet black

Paint the columns, not the cells.

Running FlexCel inside Docker containers

A Docker container needs fonts!

Text rotation in shapes inside xls and xlsx files

Do we rotate the text first and then align it, or do we align it first?

SVG files inside xlsx files

Modern Excel versions allow SVG as a file format for images inside. In this tip we explore the FlexCel support for the feature.

The maximum used column on a sheet

It all depends on what we define by "used column"

Finding the actual fonts used when exporting to PDF

When exporting to PDF, you need to find out which fonts are actually used.

Scalable images in your documentation

Did you know that you can use FlexCel's "Export as SVG" feature to get scalable images of your spreadsheets?

Using Tokens to get information from formulas

Let's imagine that you want to manually modify all formulas that refer to A1 to refer to A2. You could try some manual regex to search and replace all occurrences of "A1" by "A2" into the formula string. But of course this would replace also A11 by A21 and also do replacements in things that aren't references. Luckily FlexCel's Tokens provide a much more elegant and reliable solution to do it.

Semi-absolute references

You know that a cell reference like "A4" is relative, while "\$A\$4" is absolute. But is there a way to have a reference be absolute when it is outside the range copied, and relative if it is inside?

References in conditional formats and data validations

Formulas inside conditional formats or data validations apply to a range of cells instead of a single cell. This makes them behave differently from regular cell formulas in ways that might not be obvious.

Replacing a font by another in an Excel file

If you want to replace for example all Calibri fonts with a different font in a file, you can use the code in this tip.

Finding out how many pages will be exported

Sometimes you want to know the number of pages needed for a report, before actually exporting or printing it.

Fine-tuning row autofitting

When autofitting rows, you might want to use `ExcelFile.CellMarginFactor` besides adjustment.

Understanding CSV files

There is no such thing as a "Universal CSV" which everyone can understand. To create a CSV file, you need to know who will be reading it.

Embedding Excel files in your application

Don't use APIMate for this.

Internal numeric formats

Excel has some numeric formats that vary from locale to locale. Let's get a more in depth look under the covers.

How to change the FlexCel locale

Some numeric formats in Excel change depending on your machine locale. FlexCel will by default pick your locale from the machine configuration too, but you can change those without modifying the machine settings.

Using barcodes

Here we discuss the different ways to embed a barcode in your file using FlexCel.

How to get the hyperlink in a given cell?

Hyperlinks in a cell aren't as straightforward as other cell properties. Here we discuss how they work and how to get the link in some specific cell.

Why are xlsx files generated by FlexCel smaller than the same files generated by Excel?

You open an existing file with FlexCel and save it. Now the file is some kilobytes smaller. What is happening?

Automatically opening generated Excel files

Sometimes you might want to open the generated files directly in Excel without asking the user where to save them first. While this is not technically possible, in this tip we show a way to simulate it.

Using strict xlsx files

FlexCel supports both normal and strict xlsx files. Which one should you use?

Finding out which FlexCel version you are using

A simple code snippet to find out the version of FlexCel that you are using from inside your application.

Sign your PDF files

You are creating documents. How do you ensure that nobody alters the data on them?

Conditionally format all things!

Whenever you need change the format of a cell depending on its value, conditional format is here to help.

Expanding formulas in consecutive cells

If there only was a simple way to enter "=B1" in A1 and "=B2" in A2...

We also tell you about a simple way to check if all the formulas in a range have consistent values (so the formula in A1 refers to B1, and the formula in A2 refers to B2 and so on)

Changing the font name

Excel 2007 made changing the font name in a cell a little more complex than what it should be.

Entering multiple lines of text inside a cell

You can enter multiple lines of text inside a single cell, but there are a couple of things to be aware of.

Understanding Excel measurement units

What does a column width of 8.44 actually mean?

Dumping a Dataset into Excel

I have my data in this dataset. How do I export it to Excel?

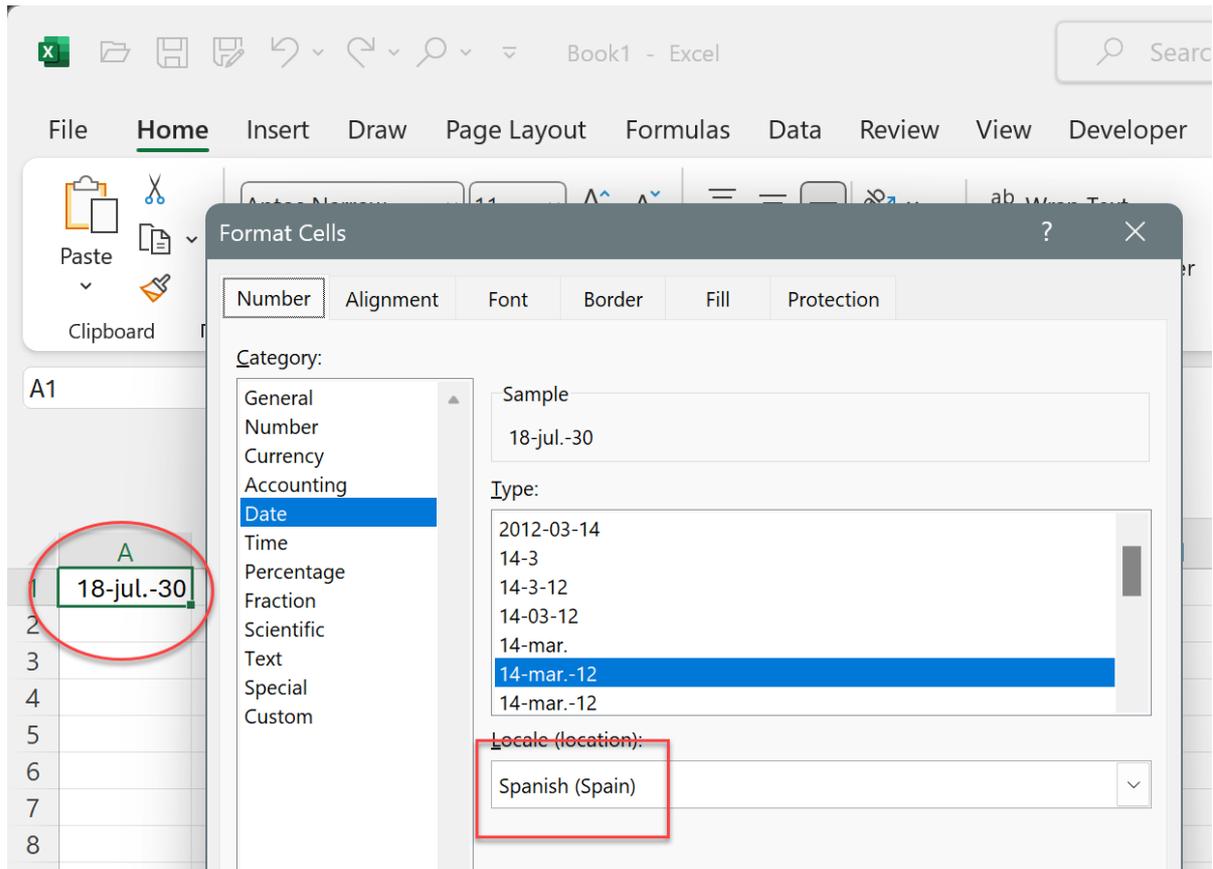
Reading only the first row of a file

If you have lots of big files to analyze, you might not want to load the full files into memory just to check if you need to process them.

Localized Month Names

Month names can be more complex than what they look like.

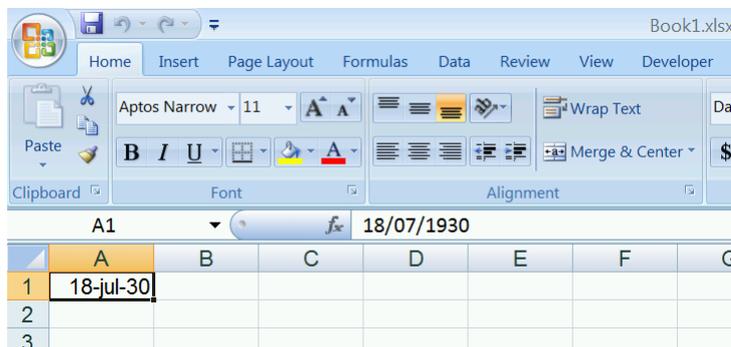
A simple example: In Excel, let's write the date "1930-07-18" and let's format it in a Spanish "dd-MMM-yyyy" format:



As you can see, it uses a dot after the month abbreviation ("jul." instead of "jul"). This is because [Abbreviations must always be followed by a full stop](#)

It makes sense. We native Spanish speakers have always known that abbreviations end in dots.

But just for fun, let's open this same file we just created in Excel 2007, Windows 7:



It makes sense, too. While we write dots after abbreviations, it is not common to write dots after abbreviated month names. Maybe because we usually write the [Código tríltero](#), not the abbreviation. So, we can conclude that, initially, the people coding Excel used the "Código tríltero," but then they realized it was wrong and switched to the abbreviation.

But now, let's look at the .NET world. Let's try this program:

```
using System;
using System.Globalization;

public class Program
{
    public static void Main()
    {
        var Culture = CultureInfo.CreateSpecificCulture("es-ES");
        Console.WriteLine(new DateTime(1930, 7, 18).ToString("dd/MMM/yy"));
    }
}
```

If we compile it with .NET 4.8, we will get `18/Ju1/30`. No dot, and the "J" in "Julio" is uppercase. What can I say? The whole thing is starting to look crazy. But we shouldn't surrender so soon. Let's change our framework to .NET 8 on the same machine and try again. And now we get "18/jul./30". So they went the reverse route as Excel/Windows, they started with the [Código tríltero](#) and then switched to abbreviations.

And Delphi has its own set of cases where it doesn't show the same as Excel (or .NET). Everyone does as they please here.

I could keep writing for hours: we've just made an example with the fourth most spoken language in the world and a language with a [Royal Academy](#) that ensures it has clear rules. If we use languages with no-so-clear rules, differences start to grow bigger. And we only tried Microsoft Windows here; if we add other OSs, it gets even worse.

There are combinations where it uses "July" instead of "Jul" as the "July abbreviation," but only for "en-AU," not "en-EN." Don't Australians abbreviate July? I don't know, and it looks like nobody does.

There are also [Genitive Month Names](#) and even [Abbreviated Genitive Month Names](#). .NET 4 will never use Genitive Month Names, .NET 8 will use them if you have a format string like "dd/MMM/yyyy", but not if you have "MMM" alone. Excel has its own rules (that also changed over the years) on whether to use Genitive month or standard month names. In .NET Core 5, Microsoft [switched from using the built-in Windows functions to ICU](#) even in Windows.

This is all to say that it is possible that you won't get the same abbreviated month names that you see in your Excel when you export from FlexCel. And if you see them the same, those might change when you update Windows, .NET, Delphi, or Excel.

It usually won't matter, but in some cases it might. For example, you might see "Jul" in Excel but "July" in FlexCel, and the text won't fit in the cell anymore. I advise not to use abbreviated month names and always leave some extra space in the cell so longer text can fit.

But if you need to get the same results as Excel with FlexCel, you can.

The first thing is to set the month names (and genitive month names) in the `CultureInfo` of your app, so they look as you want them to:

```
CultureInfo Culture = CultureInfo.CreateSpecificCulture("es-ES");
//In this case RemoveDots() just removes the dots from the abbreviated names.
You might just specify the array manually.
Culture.DateTimeFormat.AbbreviatedMonthNames = RemoveDots(Culture.DateTimeFormat.AbbreviatedMonthNames);
Culture.DateTimeFormat.AbbreviatedMonthGenitiveNames = RemoveDots(Culture.DateTimeFormat.AbbreviatedMonthGenitiveNames);
```

But this won't cover all cases. As we saw in the first screenshot, you can specify a different language for the format of a cell, even if Excel itself is using a different locale. In this case FlexCel will create that locale based in the locale code, and you need to override that too. You can use the code below to override the locale creation:

```
TFlxNumberFormat.CultureCreating += args =>
{
    switch (args.LanguageCode)
    {
        // see https://learn.microsoft.com/en-us/openspecs/office_standards/ms-oe376/6c085406-a698-4e12-9d4d-c3b0ee3dbc4a
        //for the language code
        case 1034: //Spanish - Spain (Traditional Sort)
            args.Culture = CultureInfo.CreateSpecificCulture("es-ES");
            args.Culture.DateTimeFormat.AbbreviatedMonthNames = RemoveDots(args.Culture.DateTimeFormat.AbbreviatedMonthNames);
            args.Culture.DateTimeFormat.AbbreviatedMonthGenitiveNames = RemoveDots(args.Culture.DateTimeFormat.AbbreviatedMonthGenitiveNames);
            break;
    }
};
```

Cloud fonts

As we write this tip, Microsoft has just released a [new default font](#) for all the Office applications, including Excel. And we've just released support for it, but there are some issues that we just can't fix.

A new default font is usually a headache, as the font is not likely to be available on platforms other than Windows. For that reason alone, we suggest using Calibri until the new Aptos font becomes as ubiquitous. But this time, we have another reason to recommend not switching just yet:

Unlike Calibri, Arial, or all the "classic" fonts, Aptos was delivered as a [Cloud font](#). As explained in the link above, cloud fonts are available only to the Office apps, and not to Windows itself. This means that your application (and any other application in your machine) can't see them, even if you can see them in Office.

If you are creating a PDF, a workaround for FlexCel in Windows would be to add the folder "%LocalAppData%\Microsoft\FontCache\4\CloudFonts" in the [FlexCelPdfExport.GetFontFolder](#) event.

But this isn't a great workaround because:

1. It will only work for PDF files. When printing, or previewing, FlexCel won't find the fonts, and they will be poorly substituted by another, probably breaking the layout of the page.
2. Even when exporting to PDF, FlexCel might sometimes ask the operating system for some font data, and the operating system will provide the wrong font. See also the [FlexCel PDF Exporting Guide](#)
3. The place of local fonts is not documented and might change: Today, it is "FontCache\4\CloudFonts," but tomorrow, it might change to "FontCache\5\CloudFonts," and your app will break.

Given all the issues, if you really want to use Aptos (or any other cloud font) in your files, we recommend installing them locally so that all Windows apps can see them. You can find a nice step-by-step guide in how to do so at <https://office-watch.com/2023/cloud-fonts-available-all-programs/>

IMPORTANT

At the time of this writing, the Aptos font is only available as a Cloud font, and it is not installed in Windows. It might happen that some Windows update in the future installs the font also in Windows, which would fix most of the issues of this tip. So we recommend you to open the Font Viewer in Windows, and see if the Aptos font isn't available.

Painting a full sheet black.

Let's imagine we want to paint all cells on a sheet with a black background and a white font. Unfortunately, Excel doesn't have a way to set the format for a full sheet, so that's not an option.

The first idea is that we could loop over each of the 17 billion cells in the spreadsheet and paint each cell in black. And that would work, but it would not only be very slow (even computers take their time to count to 17 billion), but it would also use a lot of memory. Because we need to create 17 billion empty cells to format them in black. Even when xlsx files are compressed, the file size would be huge, too, as it would be the time it takes for Excel to open the file.

So, no way to set the entire sheet to black, and if we try to format every cell, it won't work either. But there surely is a way, isn't it? If you create a file in Excel, select all cells, and paint them black, the file will not be much bigger than an empty file and will not take vast amounts of memory. Excel somehow manages to do it, and we know it is not setting the format for the entire sheet or for every cell.

Then, how is Excel doing it? As usual, the simplest way to find out is with APIMate. So, let's create an empty file in Excel, paint all the cells black, then save the file and open it with APIMate. And this is the code we get:

```
TFlxFormat ColFmt;  
ColFmt = xls.GetFormat(xls.GetColFormat(1));  
ColFmt.Font.Color = TExcelColor.FromTheme(TThemeColor.Light1);  
ColFmt.FillPattern.Pattern = TFlxPatternStyle.Solid;  
ColFmt.FillPattern.FgColor = TExcelColor.FromTheme(TThemeColor.Dark1);  
ColFmt.FillPattern.BgColor = TExcelColorAutomatic;  
xls.SetColFormat(1, 16384, xls.AddFormat(ColFmt));
```

While Excel doesn't have a way to format the whole sheet, it has [Cell, Row, and Column formats](#). A sheet has about 17 billion cells, nearly 1 million rows, and around 16 thousand columns. To format every cell, we would need to set the format in 17 billion cells. To format every row, we would need to set the format in 1 million rows. To format every column, we need to set the format in 16 thousand columns.

The option requiring fewer formats is formatting every column, which is what Excel does, and what you should do, too, if you want to change the format for a whole sheet.

Running FlexCel inside Docker containers

Docker containers are an excellent alternative for distributing applications that work the same anywhere. Conceptually they are similar to deploying to a Linux machine (and they share the same issues), but there is an extra problem.

Fonts strike once again

When running FlexCel in a different Operating System, the first thing to check is that the fonts FlexCel needs to work on are available. Specifically, when inside a docker container, it might be the case that no fonts are available at all, causing SKIA to return an invalid font with no name, zero width, and zero height. FlexCel will, in most cases, catch those errors and tell you to install the fonts. But in other cases, it might not. So be aware to always check for fonts.

There are two issues with fonts:

The "No font available at all" problem

It happens when there are zero fonts available in the OS, and the problem is common in containers. This is where SKIA returns a font with no name, and you will be informed of the issue. The problem goes away simply by having a single font installed. But having a single font installed will lead us to the next point:

The "Some fonts aren't available" problem

FlexCel uses the font files to calculate cell widths and heights during exporting and in other operations like autofitting. If you have a single font available to the OS, that font will be used as a substitute for any "real" font used in the Excel file. So metrics will likely be wrong.

To fix this, ** you need to have not just one generic font installed but all the required fonts instead. **

Installing fonts in a Docker container

To install some fonts into a docker container, you must first copy them to the folder where "Dockerfile" is. Once you have them there, you can zip them and install them with the commands:

Ubuntu or Debian Container:

```
WORKDIR /app
RUN apt-get update
RUN apt-get -y install fontconfig
RUN mkdir -p /usr/share/fonts/truetype/
COPY ./fonts/* /usr/share/fonts/truetype/
RUN fc-cache -f
```

Alpine-Linux Container:

```
WORKDIR /app
RUN apk add --no-cache fontconfig
RUN mkdir -p /usr/share/fonts/truetype/
COPY ./fonts/* /usr/share/fonts/truetype/
RUN fc-cache -f
```

Others Linux distributions will likely be similar, but might vary in the package manager they use (apk, apt-get, pacman, etc)

NOTE

Remember that you might need to copy multiple font files to install a single font. For example to install Arial you need to install:

- arial.ttf // Normal font
- arialbd.ttf // Bold variant
- arialbi.ttf // Bold-Italic variant
- ariali.ttf // Italic variant.

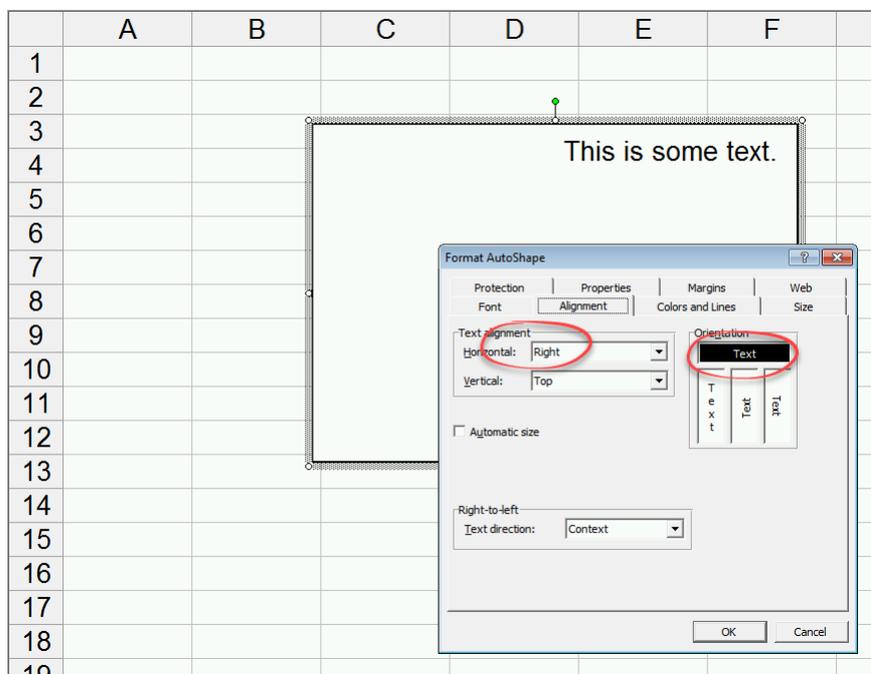
Text rotation in shapes inside xls and xlsx files

The problem with shape rotation

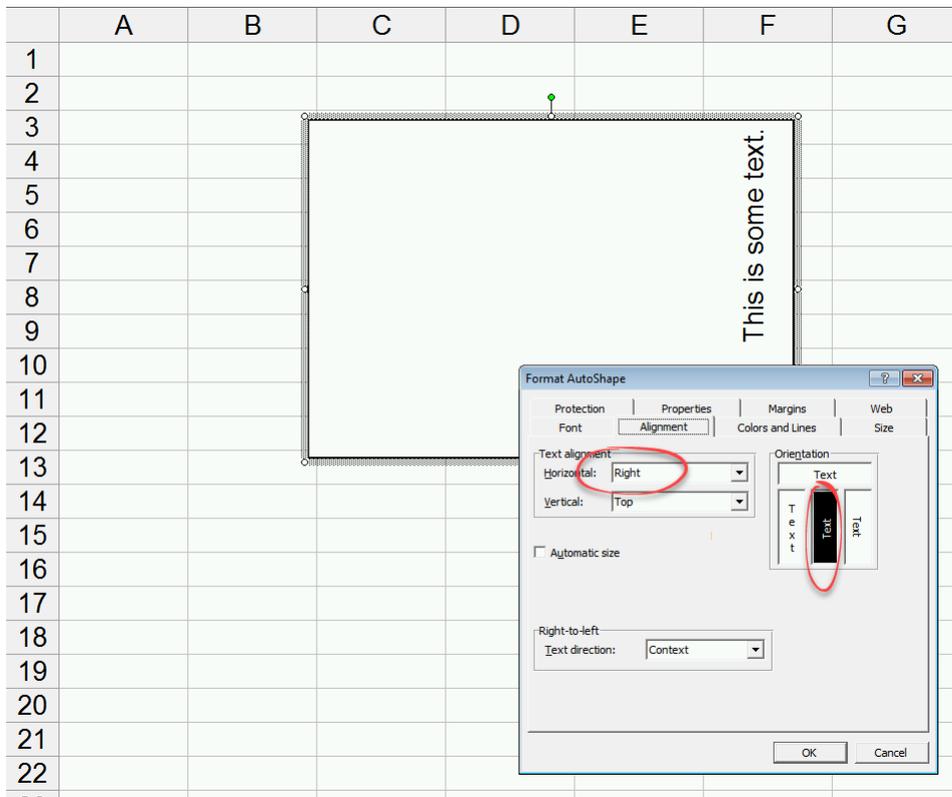
When Excel changed the file format from xls to xlsx back in 2007, they rewrote the entire graphics engine. And while the functionality was more or less the same, the base was completely different, and this created subtle differences. We usually keep those differences abstracted from you, but in this tip, we wanted to review a small thing that can be confusing.

Back in Excel 2003 times (xls), you could change the rotation of the text inside an autoshape to be rotated 90 degrees, -90 degrees, vertical or horizontal. Besides that, you could align the text to the right, left, top, or bottom.

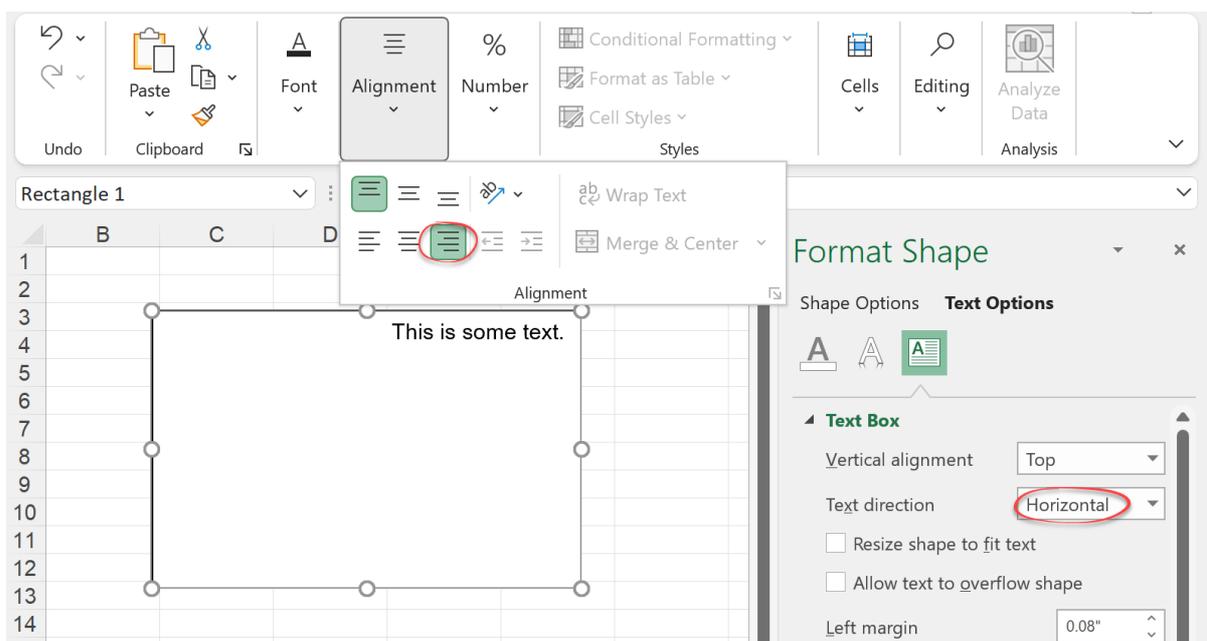
So let's imagine we have some text aligned to the right:



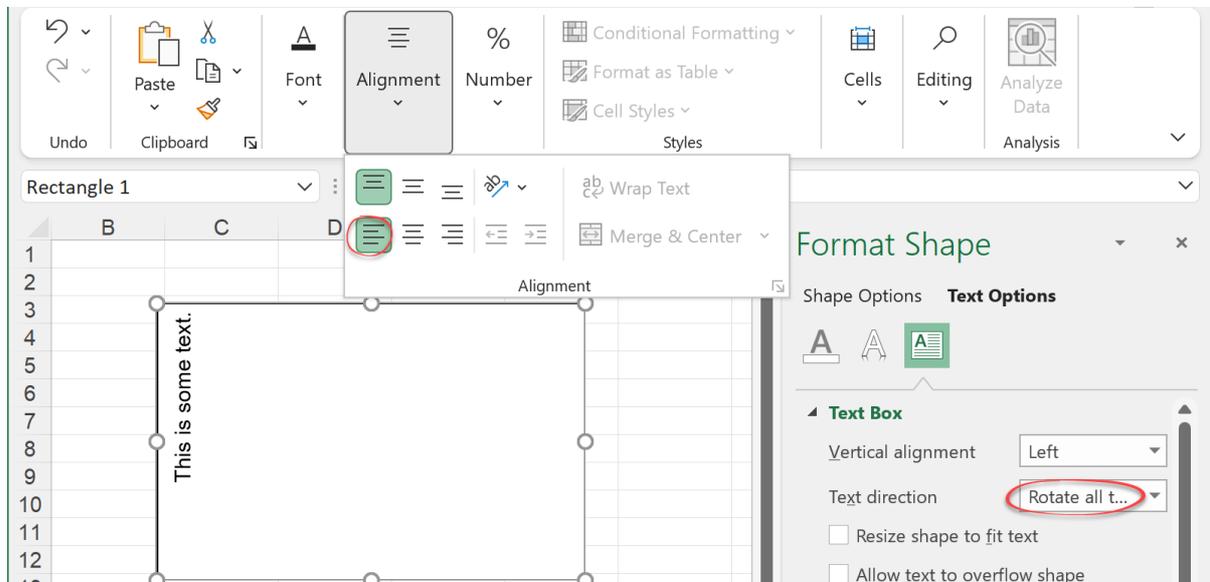
And then we rotate it 90 degrees counter-clockwise:



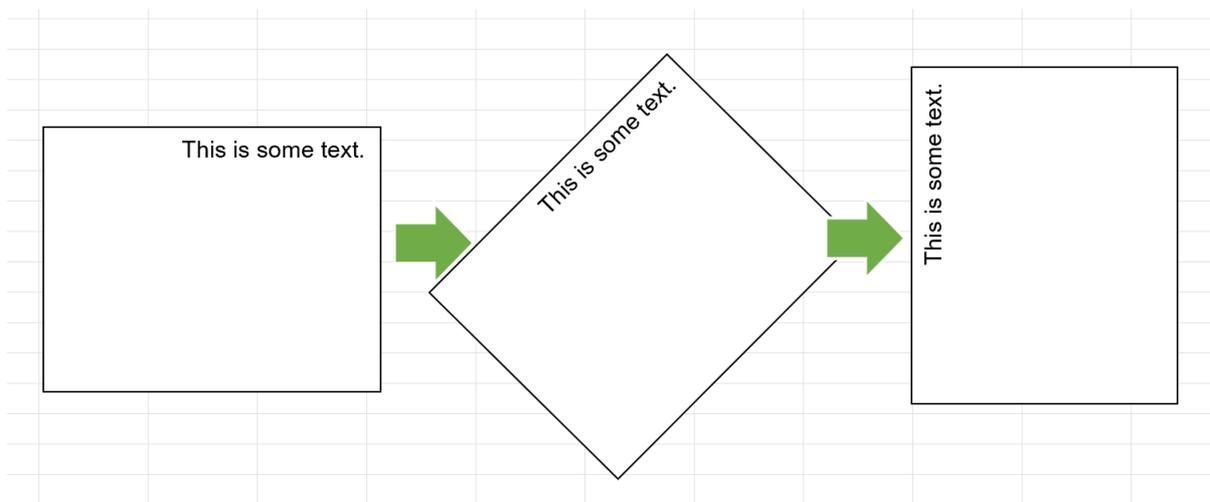
As you can see, Excel first applies the rotation, then the alignment. That means the text is still aligned to the right after rotating it. On the other hand, if you have a similar file in a newer Excel:



And rotate it the same:

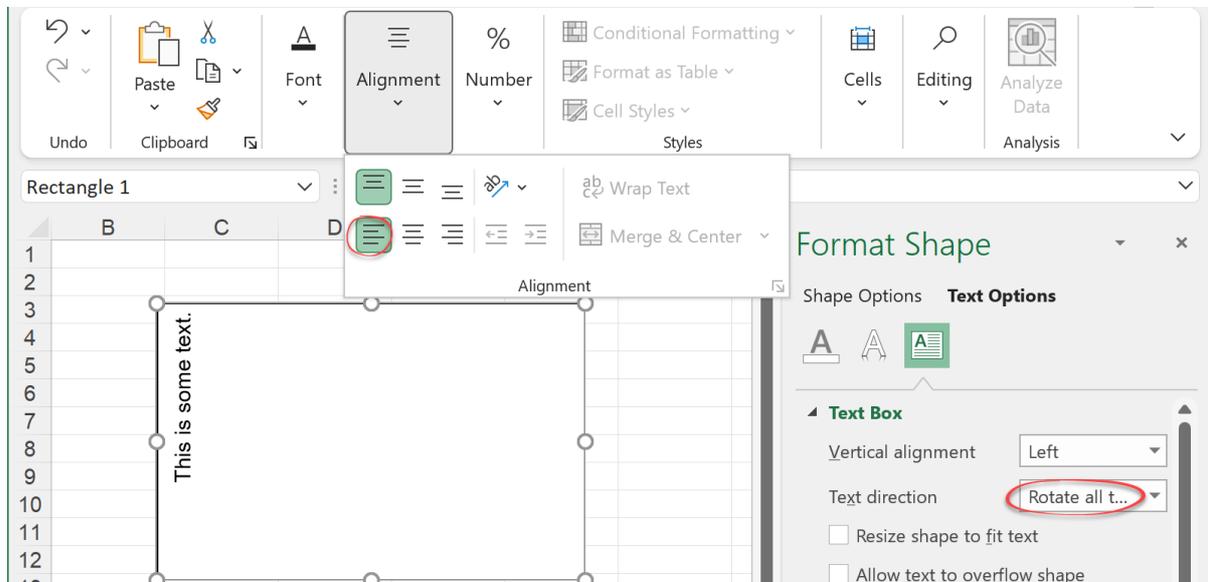


The text will now be aligned to the left, not to the right. This happens because, in newer Exrels, the text is first aligned to the right, then rotated. The alignment is part of the text, not of the shape. It might be easier to see with an image:



In older Exrels, the shape would be rotated first, and the text aligned to the right no matter the rotation.

So, we've seen the process is different under the hood. However, Excel tries to keep the illusion that nothing changed, and so it displays the text as "Top-Left aligned" in this screenshot:



But internally, in the xlsx file, the text will be saved as "Top-Bottom" aligned, with a rotation of -90 degrees clockwise.

The problem with comments and controls rotation

Probably because of how huge the task was, comments and controls weren't really migrated to xlsx in the big Excel 2007 rewrite. Xlsx files used an older, Excel-2003 xml to store those. And that means that comments and controls kept working as in Excel 2003. They rotate first, then align.

Newer revisions of xlsx introduced newer ways to write comments and controls inside the xlsx file, and that made everything worse. Now you have a control saved twice in the file, and one part has xls behavior while the other has xlsx.

How it affects FlexCel

When writing code, you need to know the order in which operations are applied. Let's imagine we are trying to get the text rotated 90 degrees and aligned to the top-right as in the second screenshot.

Inside an xls file, we need to write:

```
Shape.VerticalJustification = Top
Shape.HorizontalJustification = Right
Shape.Rotation = Rotation90DegressCounterClockwise.
```

But for xlsx, we need:

```
Shape.VerticalJustification = Bottom
Shape.HorizontalJustification = Right
Shape.Rotation = Rotation90DegressCounterClockwise.
```

We have a single API for both xls and xlsx files, so properties have to have some defined behavior. It can't be that you set the text to be top-right, then rotate the text, and when you save as xls or as xlsx you get different results.

So we've decided to have all properties to behave like xls. **To get the text top right, you need to set the justification to Top-Right.** Then FlexCel will save in the xlsx file the alignment as Bottom-Right. Same when you read the file: If an xlsx file says "Bottom-Right" in its alignment and it is rotated 90 degrees, FlexCel will report Top-Right. **If you are manually reviewing the xml inside an xlsx file, the values will differ from what FlexCel reports.**

The reasons we settled for the xls approach, even when xlsx is much newer and would normally be preferred are:

1. Our code predates xlsx, so we already had properties like [TShapeProperties.TextFlags](#) or [TShapeProperties.TextRotation](#) that used the xls approach. When we introduced simpler properties like [TShapeProperties.TextVerticalAlignment](#) it made sense to keep a consistent approach, instead of having some properties behave in one way and some in another.
 2. The xlsx way applies only to Autoshapes, but not to comments or the text in controls. So if we took the xlsx approach, we would also have to convert the comments and controls even in xlsx files.
 3. It is the way Excel displays it. Even when internally the file will say "Bottom-Right", in the Excel UI you will see the text aligned as "Top-right".
-

SVG files inside xlsx files.

A little bit of history

Back in the old times of 1997, Excel would store the images inside an xls file in a handful of (at the time) standard formats:

- **PNG**: Best for images with text or that required high fidelity. They use lossless compression, which meant images are preserved without modifications, but the files are also larger.
- **JPEG**: Best for photos or images where you want to compress a lot. They use lossy compression, which means files are smaller but modified slightly from the original.
- **BMP**: Those were similar to PNGs, but with a big difference: they weren't compressed at all. Being uncompressed made them useless except for tiny bitmaps, but if you had small images, they could be rendered faster in a time where computers were much slower than today.
- **WMF/EMF**: Windows Metafiles. Those are **vectorial formats**, which means they scale better than bitmaps and are usually smaller. They also require you to draw them with a vector-image application, and they wouldn't work well for a photo.
- **PICT**: This was the WMF equivalent for macs before OSX. They were also used to store vector images, but they were used when working in a mac instead of Windows.

Those formats provided a good set for 1997. They allowed vector and bitmap images, and the bitmaps could have lossy or lossless compression.

Let's now advance some years and go to 2007, when Excel introduced the xlsx file format. They used the opportunity to add some additional file formats to the xlsx files:

- **GIF**: This file format is equivalent to PNG but has worse compression. On the upside, they allow animated images. If you've never tried it, you can add animated gifs to Excel, and they will show with a "Play" button that you can press to start the animation.
- **TIFF**. Yet another bitmap format, used as the standard for FAX machines. The one thing they add to already supported bitmap formats is the ability to have "multi-page" images. But as far as we know, Excel doesn't support multi-page tiffs and will only show the first page.

Again, a good set for the time. But if you look closely, you will find that we have five bitmap formats against only three vector formats. And one of the vector formats (PICT) has been dead for years. The other two (WMF and EMF) are essentially the same; both are Windows-only and not really good. So it became clear that Excel needed to support a more modern, cross-platform vector-image format.

Enter SVG

Let's advance ten years more to somewhere in 2017 when Excel added SVG support to xlsx files in an [Office-365 rolling update](#).

This time the situation was different. In 2007, along with the support for tiff and gif, Excel introduced a new file format (xlsx). So it was simple to add new image formats: Old xls files would still support the old set of image formats (images would be converted to the supported formats when saving as xls), and new xlsx files would support the new set.

But SVG was introduced without a file format change. So if they just added SVG images to xlsx, this would break any existing xlsx processing tool, including any old Excel versions. The pictures would look great in the latest Excel, but if you shared the xlsx file with someone using Excel 2013, he wouldn't be able to see any.

To avoid this problem, Excel saves SVG images as both SVG *and* PNG inside the xlsx file. Old Excel versions will ignore the SVG and display the PNG, while SVG-Enabled Excels will show the SVG. Everyone is happy, and this is why old FlexCel versions will still display SVG images even when FlexCel wasn't SVG enabled.

How FlexCel deals with SVG images

Since version 7.10, FlexCel can handle SVG images. But we had a problem: Neither of the graphic engines we use for image processing (GDI+, SKIA, etc.) can render SVG images. So if we simply added support for reading the new SVG section and ignored the PNG inside the xlsx file, we would be in a strange situation: Old, not-SVG-enabled FlexCels would be able to render xlsx files with SVG correctly (because they would display the alternate PNG), and new, SVG-enabled FlexCels would not (because they would try to render the SVG, which our graphic engines don't support).

And even if you are not using the FlexCel rendering engine, you might be reading those images and doing something with them. If that is the case, upgrading from FlexCel 7.9 to 7.10 might break your app. Because now we start returning SVGs to it when SVG wasn't a file format that we could return and your app didn't have to support it. Even worse, SVG images can be tricky because they might use fonts not available on your machine. So again, this might mean that a file converted to PDF completely fine in 7.9 would start rendering wrong in 7.10 because you don't have the fonts needed to display the SVG correctly.

That would be a disaster. Backward compatibility is something we care a lot about, and our idea is that upgrading to newer FlexCel versions should not break anything. You install a new FlexCel version, and that's it. Nothing that was previously working should break. Of course, we can never guarantee 100% that there will be no breaking changes, but we make our best effort to ensure you never have to worry about breaking anything when updating.

So how did 7.10 added SVG support? We made the following changes:

1. We will now read (and write back) both PNGs and SVGs when opening and saving an xlsx file. So if you open a file, modify some cells, and save it back, the file will keep the SVG image as before. FlexCel 7.9 would ignore the SVG image and only write back the PNG. Note that this applies only to xlsx files. As xls doesn't support SVG, only the PNG will be saved to xls files.

2. When you call [ExcelFile.GetImage](#), we will still return the PNG for SVG images, ensuring that your existing code won't break. There is a new method [ExcelFile.GetImageAlternate](#) that will work the same as `GetImage` for most images but will return the SVG for SVG images. It is now your choice to modify your app to call `GetImageAlternate` instead of `GetImage` if you want to handle SVG images. If you don't change anything in your code, 7.10 will keep working as 7.9 did.
3. There are new methods [ExcelFile.SetImageAlternate](#) and [ExcelFile.AddImageAlternate](#) which will allow you to enter SVG images via the API. Those methods require both the SVG *and* the backup PNG because FlexCel doesn't have an SVG renderer that could automatically convert the SVG to PNG. As usual, APIMate will show you how to enter an SVG image from a file that has it.
4. When rendering a file to PDF or images, FlexCel will continue to use the backing PNG image. This makes it possible to display those images even if our graphics backend doesn't support SVG and also guarantees that there won't be rendering problems due to missing fonts. For SVG and HTML export, there are new properties [FlexCelHtmlExport.RasterizeSVGImages](#) and [FlexCelSvgExport.RasterizeSVGImages](#) which let you decide if you want to use the png or the SVG image for the rendered file. Those properties are false by default, which means that FlexCel will try to use the SVG images for HTML and SVG unless you change the properties.

WARNING

While not breaking backwards compatibility is one of the highest things on our list, there is a potential little thing that could break here. As mentioned in point 4, when rendering to HTML or SVG, FlexCel 7.10 will now by default use the SVG, not the PNG to do the rendering. This makes sense, because if your file contains a SVG file you will likely want the SVG image to be exported to HTML or SVG. But also as mentioned, if the SVG contains fonts that are not available in every machine, this could break the new files. If you want to be 100% sure that those files will render exactly as intended even if there are missing fonts, you should set `RasterizeSVGImages` to true.

NOTE

If adding an SVG file via the API with [ExcelFile.AddImageAlternate](#), make sure to provide a reasonably high-resolution PNG alternate image. Very low-resolution PNGs will display ok in a modern Excel (because it will ignore it and render from the SVG), but it will display wrong anywhere else.

NOTE

The property `RasterizeSVGImages` mentioned in this tip applies only to images stored inside the xlsx file as SVG. When exporting to HTML, FlexCel also has the ability to render the charts to fully-scalable SVG images instead of PNGs. For that, make sure to set the property [FlexCelHtmlExport.SavedImagesFormat](#) to SVG.

The maximum used column on a sheet.

This one looks simple enough. What is the maximum used column on a sheet? In Excel, you can find out with this macro:

```
Sub Macro1()  
    Range("a1").FormulaR1C1 = ActiveSheet.UsedRange.Address  
End Sub
```

In FlexCel, we have not one, but two methods to get this information:

- [ExcelFile.ColCount](#)
- [ExcelFile.ColCountOnlyData](#) And yet, how to get the column count is a common [question in our inbox](#). The main problem being, everyone has a different definition in mind of what "ColCount" should return.

The first question that comes to mind is: Should we include formatted columns in the result? For example, in the image below, should ColCount return 2 or 5?

	A	B	C	D	E	F
1						
2		Some text				
3						
4						
5						
6						
7						
8						

That's why we have two different methods to measure the column count. [ExcelFile.ColCount](#) will include formatted columns (column E in the example above), and [ExcelFile.ColCountOnlyData](#) won't.

But this is just the entrance to the rabbit hole. The next question to answer is, "Should we include blank formatted cells in the count?" And there is no simple answer, as it depends on why you want the column count for.

Let's say we want to calculate the range of visible cells that we want to print and let's imagine we have the following spreadsheet:

	A	B	C	D	E	F
1						
2		Some text				
3						
4						
5						
6						
7						
8						

It seems evident that if you want to print that sheet, you should include cell E5. And both ColCount and ColCountOnlyData in FlexCel will include cell E5 because both count blank formatted cells. But let's imagine now that cell E5 was bold, with a white background. It now makes no sense to include E5, since it won't be visible when you print it. So for printing, neither ColCountOnlyData nor ColCount will work. We would need a "MaxPrintableColumn" method, including an empty cell with a yellow background, but not including an empty cell with bold formatting.

Now let's consider a case like the following:

	A	B	C	D	E
1					
2	This text goes over columns B, C and D				
3					

Again, depending on what your need is for the column count, the result changes. If you need to process the cells with data, then the last column with Data is A. But if you need to print the sheet, you should print it up to column D.

And we can keep going for a while. What happens if the last column with data is hidden? It shouldn't count for printing, but it should count for processing data. And what if the cell has a formula, and the formula has an empty result? Should the cell be considered empty? Should we consider empty a cell that only has whitespace? Or a cell with a formula that returns 0, if the "display zeros" option in the workbook is off? When we start combining all the options, it turns out that we would need a lot of methods or parameters to accommodate them all, and we might still miss the particular combination that looks like the "logical" ColCount output given your situation.

The final issue with all ColCount variants is that no variant would be efficient. Given that FlexCel stores the cells grouped by rows, finding the maximum column means looping over all rows to find the maximum. It is not something that we can do more efficiently than you could. So to sum it up:

1. For most cases, you shouldn't use ColCount or ColCountOnlyData at all. Use [ExcelFile.ColCountInRow](#) instead. It is not just that ColCount is slow to compute, it is also that it calculates the maximum enclosing rectangle, and that can include lots of empty cells. See the [Performance Guide](#) for more information.
2. For the cases where you really need a column count, decide which is the exact column count that you want. Do you need to include empty formatted cells, formatted columns, etc? Then you can use the following code to calculate it:

```

static int CalcMaxColumn(ExcelFile xls)
{
    int RowCount = xls.RowCount;
    int MaxCol = 0;
    for (int row = 1; row <= RowCount; row++)
    {
        int ColCountInRow = xls.ColCountInRow(row);
        for (int colIndex = ColCountInRow; colIndex >= 1; colIndex--)
        {
            int XF = -1;
            object cellValue = xls.GetCellValueIndexed(row, colIndex, ref XF);
            if (cellValue == null) continue; // you can add some other
condition here like if it is a RichString and the richstring isNullOrWhiteSpace()

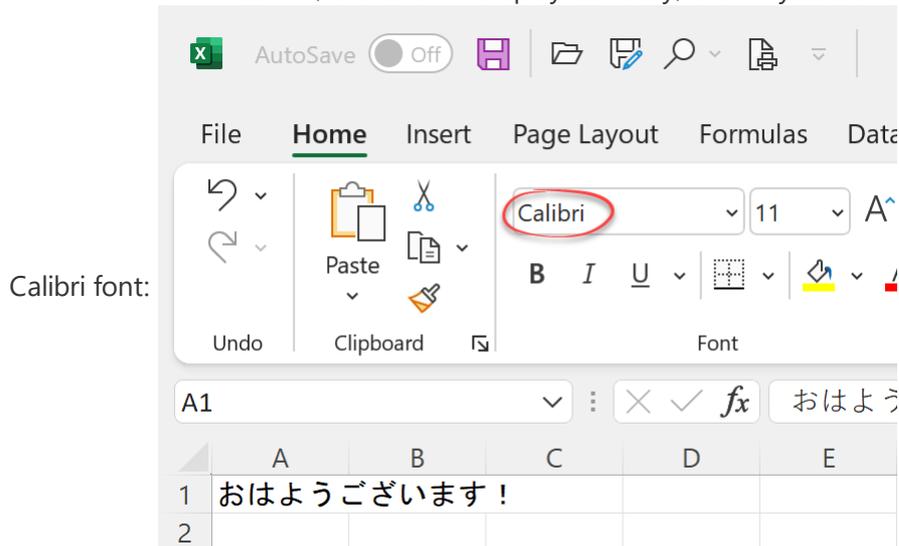
            int c = xls.ColFromIndex(row, colIndex);
            if (c > MaxCol) MaxCol = c;
            break;
        }
    }
    return MaxCol;
}

```

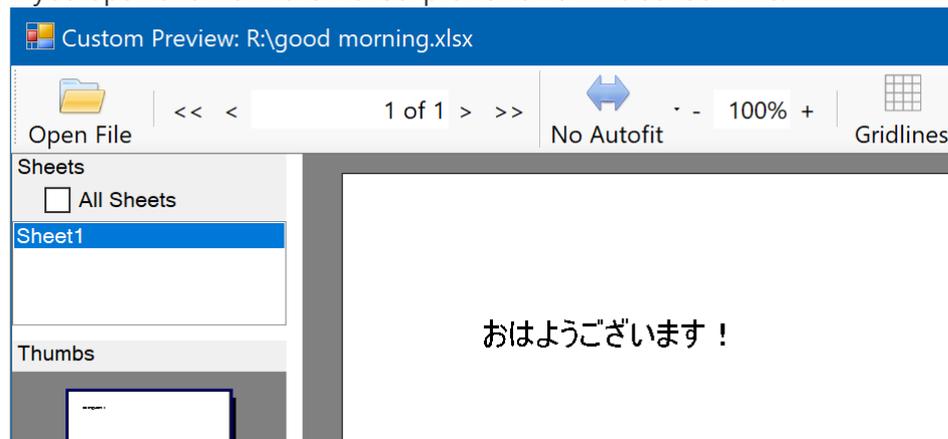
Finding the actual fonts used when exporting to PDF

As explained in the [PDF exporting guide](#), when you export a file to PDF, you might end up with white rectangles instead of the actual text.

The cause of those problems is font substitution: If you create a new file in Excel and write おはようございます！ in cell A1, the text will display correctly, even if you don't change the default



If you open this file in the FlexCel previewer it will also look fine:



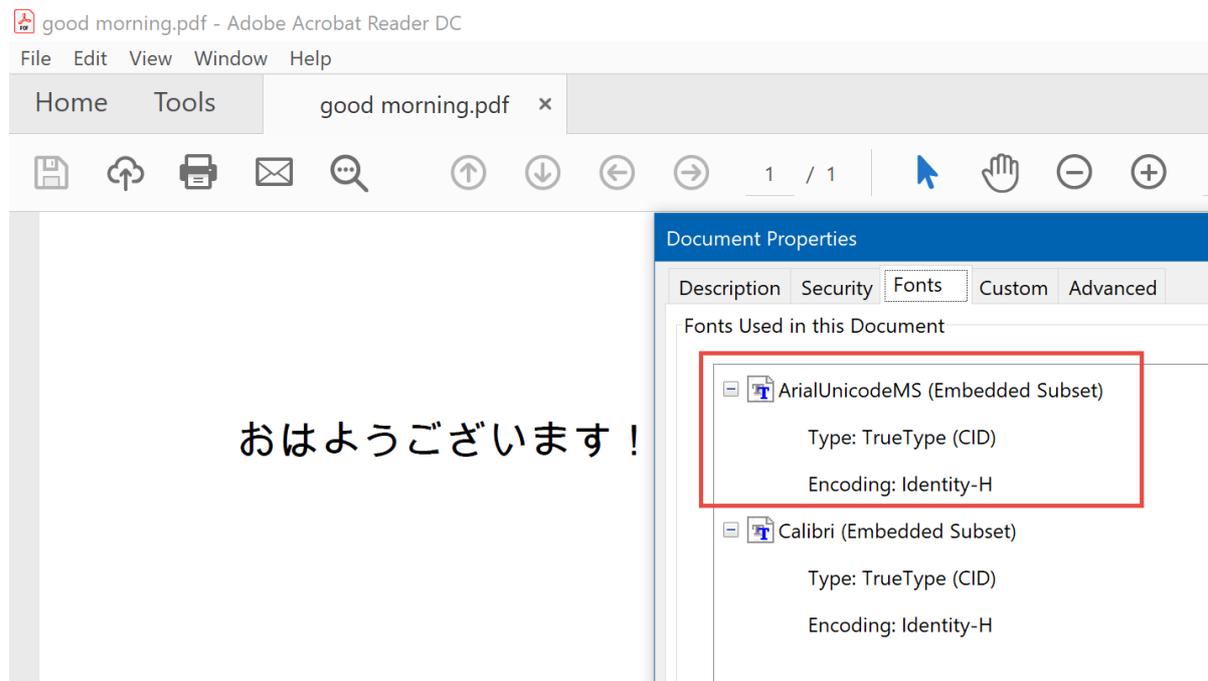
If you **print**, export to **HTML**, **images** or **SVG**, from Excel or FlexCel, the file will look fine. But what happens if you export to PDF? Given the fact that I just gave you a link about font problems in PDF, and that this tip is about fonts on PDF, you might expect the export to PDF to be wrong. But no, if you export to PDF, with Excel or FlexCel, you will see then same nice Japanese text.

And yet, everybody is lying here. Calibri doesn't contain definitions for Japanese characters, and so it is impossible to draw a お using Calibri. The image isn't there. So why do you see お instead of a blank square when you look at the text in Excel? Because under the hood, Windows is replacing Calibri by a different font that has a definition for お, and using that font to draw お instead. Same when you look at the FlexCel previewer, or when you export to PDF. Whoever is rendering the font realizes that they can't do it with Calibri, and they use a different font instead.

However, now we come to the part where PDF is different. In all the other cases, when printing, exporting to HTML or whatever, it is the Operating System or the Browser who does the font substitution. You just tell the OS to draw a お in Calibri, and it takes care of finding the best font.

PDF is different. If a PDF file tells a PDF reader to draw a お in Calibri, it will not do font replacement, and print a square. So we need to do the font replacement ourselves **before** writing the PDF file. The PDF file we generate must have the correct font to draw the characters we want.

But now you might be wondering: Why, when we exported to PDF with FlexCel, we didn't get blank squares? We can get a hint of what is happening if we look at the generated file in Acrobat and press "Ctrl-D" to see the fonts used in the document:



A-ha! There is an extra "Arial Unicode MS" font, and that is what Acrobat must be using to render the text. And so, where does "Arial Unicode MS" comes from? As mentioned in the [PDF exporting guide](#), FlexCel is using the [FlexCelPdfExport.FallbackFonts](#) property to figure out which font to use. It tries the fonts one by one until it finds one that supports the character, and then uses that instead when exporting to PDF.

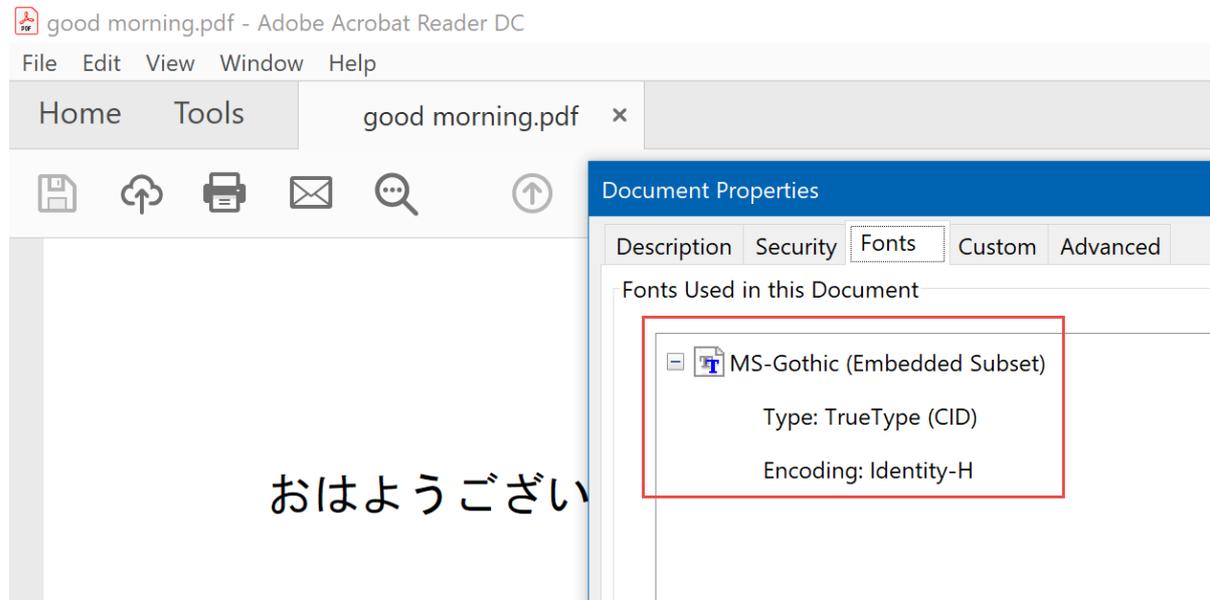
And the problem is that this is all transparent to you until it isn't. Because while we have a good set of default fallback fonts, we don't and can't cover every language. Some languages will still get a blank square (if the character is not in any of the fallback fonts), and some might get uglier fonts (because the replacement font with nicer glyphs is not in the FallbackFonts list). Also, there used to be a font with most glyphs out there (Arial Unicode) installed by default in Windows, but that font doesn't ship with Windows 10 anymore (probably because it was too big).

So let's go back to our Japanese. Let's imagine that we are in a computer without Arial Unicode or any other fallback font supporting Japanese, and we want to modify our FallbackFonts list to support it. Which font should we use?

Excel won't show you what font it is using to display the text on the screen. It will happily tell you it is "Calibri" even when we know it isn't. If you export the file to HTML, it will be exported as "Calibri" too, and the browser will perform the font substitution, but the browser won't tell you either. The process is so transparent; that nobody will tell you the real font they are using. They will just pretend it is Calibri.

Until we remember that PDF is different. And now, after so many words, this leads us to today's tip: **To find out the fallback fonts you need for exporting to PDF in FlexCel, export the file to PDF with Excel, and then press Ctrl-D in Acrobat to look at the fonts Excel used.**

In our example, if we exported the original file to PDF with Excel, not FlexCel, and pressed Ctrl-D, we would see:



And this solves the case. Excel used MS-Gothic, not Calibri, to draw the Japanese text. If you are getting white squares when exporting from FlexCel, consider exporting the file to PDF from Excel, and look at the fonts it used.

Using scalable images in your documentation

When writing FlexCel documentation at tms, we often need to include document screenshots.

In many cases, a simple screen capture will do. But in others, we'd like something that can scale with the monitor resolution, and a bitmap is not enough. In those cases, we use FlexCel to export the images as SVG (scalar vector graphics).

We will discuss two different cases here:

1. Sometimes you want to render a simple object. Like for example the chart at the top of this [blog post](#) We created that image from the file in the [Chart API](#) example, using the following code:

```
var svg = xls.RenderObjectAsSVG(-1, "@lines of code", "Lines of code over time", "This chart was rendered with RenderObjectAsSVG", Encoding.UTF8);
{
    File.WriteAllText("flexcel-lines-of-code.svg", svg);
}
```

2. Other times, you might want to render a part of a sheet, as we did in the [diagram at the end of this article](#). On those cases, we use code similar to the following:

```
using (var svg = new FlexCelSVGExport(xls, true))
{
    // Set a page size that has the size of the image you want.
    // You will need to experiment a little to get the correct size.
    svg.PageSize = new TPaperDimensions("custom", 780, 300);

    svg.SaveAsImage((x) => { x.FileName = "result" + x.PageNumber.ToString(CultureInfo.InvariantCulture) + ".svg"; });
}
```

You might change the print scaling of the xlsx file to get a smaller or bigger image.

Using Tokens to get information from formulas

Introduction

When reading formulas with FlexCel, you get a string with the formula definition, like for example:

```
=SQRT(A1^2 + A2^2)
```

And while this is enough in many cases, sometimes you need a little more information. For example, which are the cells used in the formula above? It is challenging to know that the cells used are A1 and A2 just by looking at the string. You could try to use some kind of regex to get the information, but you will hit limits soon. Formula syntax depends on the context, and depending on its surroundings, the string "A1" might be present in the formula and not mean a cell.

To solve this problem correctly, we need to parse the string. And guess what? FlexCel already has one parser which you can use. To print out the cells used in the formula above, you can use the following code:

```
//Set up a file to analyze.
XlsFile xls = new XlsFile(1, TExcelFileFormat.v2021, false);
xls.SetCellValue("A2", new TFormula("=SQRT(A1^2 + A2^2)"));

//Print the cell addresses in the formula.
var tokens = xls.GetFormulaTokens(2, 1);
var cellrefs = from x in tokens where x is TTokenCellAddress select x;

foreach (TTokenCellAddress cellref in cellrefs)
{
    Console.WriteLine(cellref.Address.CellRef);
}
```

NOTE

In the example above, we set cell A2 to the formula and then use [ExcelFile.GetFormulaTokens](#) to read the tokens. This was done to simulate a real-world situation where we already have the formulas in an existing file, and we want to parse them. But for this particular example, instead of setting the value of a cell and then reading the tokens from the cell, we could have used [ExcelFile.GetTokens](#) to get the tokens directly from the string.

Analyzing formulas in a file

[ExcelFile.GetFormulaTokens](#) returns an array with the tokens that make up the formula, using [RPN Notation](#)

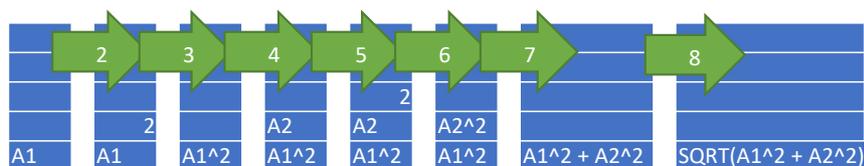
RPN is straightforward to evaluate mechanically. When you have a data token, you push it into the stack. When the token is an operator or function taking n parameters, you combine the n last members of the stack and replace them by the result. So let's see in more detail the tokens in the formula above. This is the full list of tokens returned by GetFormulaTokens:

- TTokenCellAddress - **A1**
- TTokenData - **2**
- TTokenOperator - **Power**
- TTokenWhitespace - the is the space before the "+". We will ignore whitespace
- TTokenCellAddress - **A2**
- TTokenData - **2**
- TTokenOperator - **Power**
- TTokenWhitespace
- TTokenOperator - **Add**
- TTokenFunction - **SQRT**

Using RPN, this is evaluated as follows:

1. Push A1 into the stack
2. Push 2 into the stack
3. Calculate $A1^2$ - Now the stack contains only $A1^2$
4. Push A2 into the stack
5. Push 2 into the stack
6. Calculate $A2^2$ - Now the stack contains $A1^2$ and $A2^2$
7. Add both entries in the stack. Now the stack contains $A1^2 + A2^2$
8. Calculate SQRT of the entries in the stack. The result is $SQRT(A1^2 + A2^2)$

You can visualize the stack after each step above in the following diagram:



Manipulating formulas in a file

Up to now, we've seen how to analyze existing formulas. This enough can be very powerful, and allows you to create custom static analyzers that can detect unwanted constructs in spreadsheets. But do you know a nice feature that most good static analyzers have? They allow you to apply fixes to your code automatically.

So let's imagine we have a file with thousands of formulas, and in cell B1 we have a constant which contains the tax to apply. We want to check all the formulas for references to B1, and make sure they reference a name "Tax" instead. We have half of the solution already:

[ExcelFile.GetFormulaTokens](#). This method will allow us to detect all references to B1 in the formulas. But now, to actually modify the code, we need the other half:

[ExcelFile.SetFormulaTokens](#)

We can now read the tokens of a formula, modify them, and write them back. The following code will replace all occurrences of B1 by the name "Tax":

```
//Set up a file to analyze.
XlsFile xls = new XlsFile(1, TExcelFileFormat.v2021, true);
xls.SetCellValue("A2", new TFormula("=B1 * C2"));
xls.SetNamedRange(new TXlsNamedRange("Tax", 0, 1, 1, 2, 1, 2, 0));

//Modify all references to B1 to be references to the name "Tax"
var tokens = xls.GetFormulaTokens(2, 1);

var modified = false;
for (int i = 0; i < tokens.Count; i++)
{
    if (tokens[i] is TTokenCellAddress address
        && address.Address.Row == 1
        && address.Address.Col == 2)
    {
        tokens[i] = new TTokenName("Tax", null, null);
        modified = true;
    }
}

if (modified)
{
    xls.SetFormulaTokens(2, 1, tokens);
}
```

Semi-absolute references

As you probably already know, in Excel, you have absolute and relative references. An absolute reference has a \$ before either the column or the row. So, for example, \$B4 has an absolute column and a relative row, while \$B\$4 has both absolute column and row.

Absolute references don't change when you copy the formula, while relative references adapt. If you have the following formula in cell A1:

```
=B1+$B$1
```

And you copy cell A1 into A2, you will get

```
=B2+$B$1
```

	A
1	=B1+\$B\$1
2	=B2+\$B\$1

The row in the relative reference changed, while the absolute reference didn't. This is all simple and makes sense: You use relative references for variables that change when you change the row or column, and absolute references for constants. It seems like this should cover all cases, and yet, you might find that sometimes none of them works in the way you want them.

When a reference needs to be relative sometimes, and absolute others

Let's imagine this simple case:

	A	B	C	D
1	VAT	0.21		
2				
3	Clothing			
4	Discount for clothing	0.3		
5	Item	Price without tax	Price with tax	Final price
6	Shoes	10	=B6*\$B\$1	=C6*B4
7	Shirts	15	=B7*\$B\$1	=C7*B4
8	Pants	8	=B8*\$B\$1	=C8*B4
9				
10	Toys			
11	Discount for Toys	0.5		
12	Item	Price without tax	Price with tax	Final price
13	Choo-choo train	7	=B13*\$B\$1	=C13*B11
14	Pet rock	1500	=B14*\$B\$1	=C14*B11
15	Dolls	9	=B15*\$B\$1	=C15*B11
16				

Here cell B1 is a clear case of an absolute reference. It contains the VAT, and the VAT is a constant for all cells in the sheet. On the other hand, column "Price without Tax" is a clear case of a relative reference: You want "Price with Tax" to be $=B6 * \$B\1 for row 6, and $=B7 * \$B\1 for row 7.

But take a look at the formula in column "Final price". For the first block (clothing), you want an absolute reference to the clothing discount (cell B4). But for the second block (toys) you want a different reference to the toys discount (cell B11)

If we made the formula in "Final price" column to be $=C6 * \$B\4 , it will work fine for the first block, but cell D13 would have the formula $=C13 * \$B\4 instead of $=C11 * B11$.

But if we made the formula relative: $=C6 * B4$, then the formula in D7 would be $=C7 * B5$

This is a case where neither relative or absolute references would work. And so FlexCel introduces a new mode, which we've called "semi-absolute references".

What is a semi-absolute reference

We define a semi-absolute reference as a reference which behaves as relative when inside the block being copied, and as absolute when outside. In the example above, when we copy the rows from 3 to 9 to row 10:

- Cell B1 is outside the block being copied (outside rows 3 to 9). So references to $\$B\1 will be absolute.
- Cell B4 is inside the block being copied. So even when we write $\$B\4 , this reference will be relative, and point to row 11 for the copied block (rows 10 to 15)

How to use FlexCel's semi-absolute references

Since we can't change Excel formula syntax, semi-absolute references don't introduce a new notation. That is, we can't add a new syntax like $=A1 + \text{€A€1}$ for semi-absolute references, because Excel wouldn't understand them. So semi-absolute references are a mode instead, which applies to all absolute references.

When in semi-absolute mode, all absolute references will behave as semi-absolute.

You can enter semi-absolute reference mode by either:

- When using the API, setting `ExcelFile.SemiAbsoluteReferences = true`.
- When using Reports, setting `FlexCelReport.SemiAbsoluteReferences = true`
- When using Reports, you can also write `<#Semi Absolute References>` or `<#Absolute References>` in the expressions column in the `<#config>` sheet. This allows you to decide if to use semi-absolute references on a report-by-report basis.

NOTE

You will find that semi-absolute references make more sense than absolute-absolute references in most cases. They simply behave more intuitively. So typically, there is no problem in setting `SemiAbsoluteReferences = true` by default in your reports. When an absolute reference is inside the block being copied, you usually want the new cells to refer to the copied absolute reference, not to the old reference.

The only reason we don't set semi-absolute references on by default is because this is not the way Excel behaves, and we try to be the as much compatible as possible with Excel.

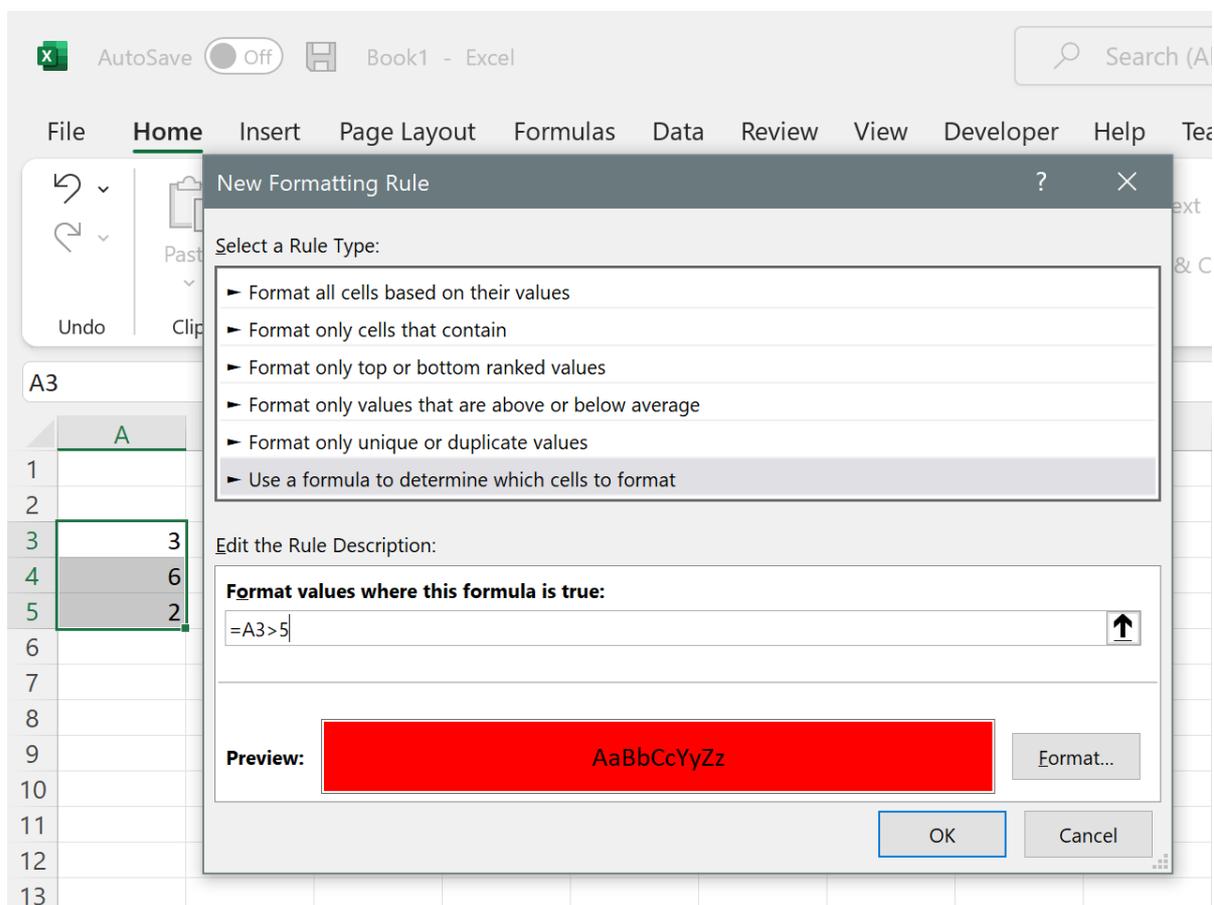
References in conditional formats and data validations

In theory, formulas in conditional formats should behave similarly to formulas in cells. But the thing is: A conditional format applies not to a **cell** (like cell formulas) but to a **range of cells**. As we will see below, this causes differences that are important to know.

NOTE

In this tip, we will focus on Conditional formats. But what we say here applies whenever a formula covers a range of cells, as it is the case with, for example, data validations too.

Say, you might apply a conditional format to cells A3:A5 where if the number is > 5 it will show red:



As you can see in the image, we apply the format if A3 is bigger than 5. But what does "A3" mean here? If you press enter, you will see that A4 becomes red, even when the formula we wrote said "a3>5" (and A3 is not bigger than 5):

	A	
1		
2		
3	3	
4	6	
5	2	
6		
7		

And if you think about it, you can't write a "normal" cell formula here, because the same formula applies to a range of cells, and you want the formula to apply to A3 when it is in A3, to A4 when it is in A4, and so on. So A3 in the formula above means "the cell where we are evaluating the format". If the cell is A3, it means A3; if the cell is A5, it means A5.

Here Excel and FlexCel behave a little different. Excel uses the top left corner cell of the range (in this case, A3) to mean "cell with 0-offset". So here, A3 means "current cell" and in this formula A4 would mean "cell which is 1 row below". This is because the range where the conditional format applies starts at A3. In FlexCel the "0 offset cell" is always A1.

So if you look at the formula in FlexCel, it will say "`=A1>5`" instead of "`=A3>5`". This is a cosmetic difference, as Excel internally uses (and stores in the file) the formula "`=A1>5`", and using A1 always as the 0-offset cell makes more sense. But it is a difference you should be aware of.

NOTE

The fact that references in conditional format's formulas are relative can also have some interesting consequences when the offset is negative. What happens if we want to refer to "the cell that is 4 rows above us"? In the example we used, A3 meant "the cell that is in the current row". So the cell 4 rows above should be A(-1)?

No, we can't have negative rows. We need to wrap up the formula and start counting from the bottom of the spreadsheet (row 1068576). So the cell that is 4 rows above is A1068575 for Excel, and A1068572 when in FlexCel.

And this creates yet another problem: What is the last row of the spreadsheet? Currently, for an xlsx file it is indeed 1068576, but for xls files is 65536. And it used to be less than that, and in the future it could be bigger than that. Every time Excel changes the grid size, references in conditional formats can break. (see ["Conditional Formats might become invalid" in the API Guide](#).)

While we wouldn't expect Excel to change the grid size again soon (because this and other problems it can cause), if possible, it is a good idea to not use negative offsets inside formulas that apply to ranges of cells.

Replacing a font by another in an Excel file.

Sometimes you might want to replace all occurrences of a font by another inside an Excel file. Maybe the old font isn't widely available in some new platforms that you want to support, maybe there are licensing issues with the font, or maybe your company changed their corporate font and you need to change it in all existing reports. The reason doesn't really matter. So, how do we do it?

NOTE

Sometimes you might want to only change the font in the resulting PDF file, not in the xls/x files. If you only care about PDF outputs, make sure also to read the [PDF exporting guide](#)

Step 1: Manually inspect the old and the new font.

Before even starting, let's make this clear: **Fonts can have different metrics**. Some fonts are very similar, like Arial and Helvetica, but some other fonts can be completely different.

If the fonts you are replacing have similar metrics, you can skip to the next step. But if they don't, the first thing to do is to analyze how different they are.

In this example we are going to replace Calibri by Verdana, so let's write a sentence in Calibri 11pt and Verdana 11pt to see how different they are:

	A	B	C	D	E	F
1	The secret of getting ahead is getting started.					
2	The secret of getting ahead is getting started.					
3						

Oops... they are quite different. So we can't just do a "search and replace" from Calibri to Arial, without breaking the report layout. For this particular phrase, Calibri 11pt should be replaced by

something like Verdana 8.5 pt:

	A	B	C	D	E
1	The secret of getting ahead is getting started.				
2	The secret of getting ahead is getting started.				
3					

And this means that the existing Calibri text must be reduced in size by about 0.8 to get a similar text in Verdana.

NOTE

0.8 is the factor for the phrase that we tested: "The secret of getting ahead is getting started." While it shouldn't vary too much, some phrases will be shorter or longer, depending on the metrics of every character in both fonts. We are doing no exact science here, and unless you are replacing very similar fonts like Arial by Helvetica, you must expect some layouts to break.

Step 2: Replace the fonts.

Ok, so we now know we have to replace Calibri 11pt by Verdana 8.5pt, or more in general Calibri Npt by Verdana $N \cdot 0.8$ pt.

In Excel, we would have to change the fonts in every cell, and the styles, and the themes, and text inside autoshapes, and text inside charts, and so on. But luckily for us, in FlexCel it is more straightforward. There are two places we have to change: The font list and the themes. And you can do so with the code below:

```
const string FontToReplace = "Calibri";
const string ReplaceWith = "Verdana";
const double FontFactor = 0.8;

XlsFile xls = new XlsFile("original_file.xlsx", true);

//Change the fonts
for (int i = 0; i <= xls.FontCount; i++)
{
    TFlxFont fnt = xls.GetFont(i);
    if (String.Equals(fnt.Name, FontToReplace, StringComparison.OrdinalIgnore
Case))
    {
        fnt.Name = ReplaceWith;
        fnt.Size20 = (int)(fnt.Size20 * FontFactor);
    }
    xls.SetFont(i, fnt);
}

//Change the theme font
var Theme = xls.GetTheme();
Theme.Name = "My theme";
Theme.Elements.FontScheme.Name = "My font scheme";
var textFont = new TThemeTextFont(ReplaceWith, "", TPitchFamily.FIXED_PITCH__
SWISS_FONT_FAMILY, TFontCharSet.Ansi);

if (String.Equals(Theme.Elements.FontScheme.MajorFont.Latin.Typeface, FontToR
eplace,
StringComparison.OrdinalIgnoreCase))
{
    Theme.Elements.FontScheme.MajorFont.Latin = textFont;
}

if (String.Equals(Theme.Elements.FontScheme.MajorFont.ComplexScript.Typeface,
FontToReplace,
StringComparison.OrdinalIgnoreCase))
{
    Theme.Elements.FontScheme.MajorFont.ComplexScript = textFont;
}

if (String.Equals(Theme.Elements.FontScheme.MajorFont.EastAsian.Typeface, Fon
```

```

tToReplace,
    StringComparison.OrdinalIgnoreCase))
    {
        Theme.Elements.FontScheme.MajorFont.EastAsian = textFont;
    }

    if (String.Equals(Theme.Elements.FontScheme.MinorFont.Latin.Typeface, FontToR
eplace,
        StringComparison.OrdinalIgnoreCase))
    {
        Theme.Elements.FontScheme.MinorFont.Latin = textFont;
    }

    if (String.Equals(Theme.Elements.FontScheme.MinorFont.ComplexScript.Typeface,
FontToReplace,
        StringComparison.OrdinalIgnoreCase))
    {
        Theme.Elements.FontScheme.MinorFont.ComplexScript = textFont;
    }

    if (String.Equals(Theme.Elements.FontScheme.MinorFont.EastAsian.Typeface, Fon
tToReplace,
        StringComparison.OrdinalIgnoreCase))
    {
        Theme.Elements.FontScheme.MinorFont.EastAsian = textFont;
    }

xls.SetTheme(Theme);

xls.Save("result_file.xlsx");

```

Finding out how many pages will be exported

If you need to calculate the number of pages a report will take without actually exporting it or printing it, you can use the code below.

```
public int NumberOfPagesToExport(ExcelFile xls)
{
    using (FlexCelImgExport img = new FlexCelImgExport(xls))
    {
        var ExportInfo = img.GetFirstPageExportInfo();
        return ExportInfo.TotalPages;
    }
}
```

You can use the [FlexCellImgExport.GetFirstPageExportInfo](#) method to find out a lot of information about how the pages will be exported. For example, you might have set a "Print to fit" in one page wide, and wonder which zoom FlexCel will use to fit the page. If the zoom is too small, you might want to change it to fit in two pages wide. To get the zoom used in the page, you can use:

```
public double FinalZoom(ExcelFile xls, int sheet)
{
    using (FlexCelImgExport img = new FlexCelImgExport(xls))
    {
        var ExportInfo = img.GetFirstPageExportInfo();
        return ExportInfo.Sheet(sheet).ZoomUsed;
    }
}
```

Fine-tuning row autofitting.

As discussed in the [Api guide](#) the text in Excel can change with the resolution or zoom of the printed page, and what fits at one resolution might not fit at another.

The solution for that problem is to make the column or row big enough that it will fit at any resolution, and to do so, `ExcelFile.AutofitRow` and `ExcelFile.AutofitCol` have two parameters: "adjustment" and "adjustmentFixed"

Basically, if for example `AutofitCol` calculates that the needed column width is 9, then it will set the actual column width to $9 * \text{adjustment} + \text{adjustment fixed}$.

If adjustment is 1.1 and adjustmentFixed is 0.2, then the column width will be set not to the 9 calculated, but to $9 * 1.1 + 0.2 = 10.1$

Similarly, if the autofit engine calculates that the exact row height to fit some lines of text is 100, it will set the row height to $100 * \text{adjustment} + \text{adjustmentFixed}$.

But rows and columns are not the same, and **autofitting rows is different from autofitting columns**. This is because we write horizontally from left to right (or right to left) and not from top to bottom. And adjustment and adjustmentFixed are not great for providing that extra margin when autofitting rows.

Let's imagine we are trying to fit the text "Hello World" into a column. As said, if we calculate that the width necessary to fit the text is 9, we will make the column 10.1 and so the text will fit anyway even if the resolution is different.

But now, let's imagine that we want to autofit the row containing "Hello World". Same as with the column, we provide an adjustment, so the row will be a little larger than what it needs to be. But, at some resolutions "Hello World" might fit in one line inside the column:

	A	B
1	Hello World	
2		

and at other resolutions it might break into "Hello" in the first line and "World" into the second line:

	A	B
1	Hello	
1	World	
2		

It is a very small difference in the width of the text, but it causes the row height to double. We would need to have an adjustment of 2.0 to correctly display the text at any resolution. But an adjustment of 2.0 will make **all other lines twice as big as needed**. Adjustment alone is not a solution when fitting rows (as it is for fitting columns).

To ensure the row with "Hello World" is always set to 2-line height, we need to reduce a little the effective width of the cell.

Imagine for example that at resolution A "Hello World" is 9.9 inches long, but at resolution B it is 1.1 inches long. If the cell width is 10 inches, then at resolution A it will fit in one line, but at resolution B it will take 2 lines. But if we reduce the effective width of the cell from 10 inches to 9.8 inches, now no matter which resolution we use, it will always take 2 lines.

On the other hand, the text "Hello" is 0.5 inches long, so it fits always in one line at resolution A or B. We got what we wanted: "Hello" always fits in one line, "Hello World" always fits in 2 lines.

In FlexCel, the way to reduce the effective width of the cell is using the property [ExcelFile.CellMarginFactor](#)

So if you are having issues with autofitting rows at different resolutions, **you might want to set [ExcelFile.CellMarginFactor](#) to a value like 1.1 before autofitting the rows.**

Understanding CSV files.

The problem with CSV

In theory, CSV files should be simple. They might not have the rich formatting abilities of an xls/xlsx file, but they are just text, and text is simple. They should be much better for interoperability between apps, because every app should be able to understand text, so they should be able to parse CSV files without much effort.

But in practice CSV is a poor format to share data to unknown apps, and the reason is the lack of a standards-defined CSV file format. Or actually the problem is not that there is not a standard, because there is one: <https://tools.ietf.org/html/rfc4180>

The problem is that nobody implements the standard. If you create CSV files which conform to the standard, most apps that can load CSV files (including Excel, Open Office and Google docs) will fail to open the files.

CSV files are what is known as a "de-facto" standard, and by de-facto we mean "Valid CSV files are the files that Excel generates". Other apps or tools will try to open and save "Excel-CSV" files, not "Standard-CSV" files, and those tools include FlexCel. FlexCel will generate Excel-compatible-CSV, so your users can open those CSV files with Excel or other tools that understand "Excel-compatible-CSV". In the same way, when reading FlexCel will assume the file is also Excel-compatible-CSV, because most CSV files out there are.

Ok, so going back to the title, what is the problem with CSV? The problem is that "Excel-compatible-CSV" is not a single standard: Different Excel versions and languages will generate different files. There are multiple CSV standards, and **when you open a file you have to know which standard was used to generate it. When you create a CSV file, you must know which application will read the CSV file**, and save the file in a format the target application will understand.

In the rest of this document we will discuss the differences.

The locale

Different languages use different decimal and thousands separators. In English, the decimal separator is the dot, and the thousands separator is the comma. So we would write the number "one thousand, ninety-nine point twenty-five" as "1,099.25". But in most other languages, the comma and the dot are reversed, and for example in Spanish, the same number would be written as "1.099,25".

Different languages also have different date ordering. In English, 2/1/1998 means "February 1, 1998". In many other languages it means "January 2, 1998"

Now imagine that you are reading a CSV file generated by an unknown application, and you find the number "1.234" in the file. Does this number mean "1 dot 234" or "1 thousand, 2 hundred and thirty-four"? You can't really know. If the file was created by some user with Excel in a Spanish machine, then it is 1234. If the user was in an English machine, then it is 1.234.

The same problem happens when you want to create a CSV file that other application(s) will read. Do you save "February 2, 1998" as "2/1/1998" or "1/2/1998"? You need to know who is going to read the file, and save a file they will understand.

The separator

It is there in the name: "**C S V**" stands for "**C**omma-**S**eparated **V**alues" So you would expect the different columns in a CSV to be separated by commas. But probably because in non-English locales the comma is used as the decimal separator, in non-English locales Excel uses the semi-colon (;) as the field separator. To avoid the confusion between "," and ";", some other applications use a tab (character 9) to separate between fields (this is also known as "TSV" or tab-separated values).

Once again, when you read a CSV file you need to know which separator was used in the file. When creating the file, you need to know which separator the application reading the CSV file will expect.

The encoding

Finally, you need to know the encoding to save the file. Excel by default uses the Windows' encoding when saving to CSV, and this means the meaning of the bytes in the CSV file vary depending on the machine reading them.

If for example a user saves a CSV in a Russian-locale machine, the file will be saved with a [Windows-1251 Encoding](#) This means that if he writes the character "Д" in the file, it will be saved as the character 196. When a user with an English locale ([Windows-1252](#)) opens the file, he will see an "Ä" instead since that is what the number 196 represents in Windows-1252.

In an ideal world, we would be using [UTF-8](#) as the universal encoding, but we are far from living in an ideal world. In this real world, once again, you need to know who is going to read the file and save the file in the encoding they expect. When reading files, you need to know what encoding was used to create them.

IMPORTANT

In Windows encodings like 1252 every byte is a valid character, so there is no way to know if the encoding is incorrect. The text you are reading might make no sense, but any combination of bytes is valid. But in UTF8 many combinations of characters are invalid, so when loading an UTF8 document, the UTF8 decoder can normally detect invalid documents.

What the decoder does when detecting invalid UTF-8 sequences depends in the OS and platform you are using: Sometimes it will change the invalid character by a "?" or a white square, but on others it might refuse to decode the full text.

If you are getting 0 rows when importing a CSV file in FlexCel, you most likely are using UTF-8 and the file is not in UTF-8. The UTF-8 decoder is detecting invalid characters and passing an empty string to FlexCel. When FlexCel loads the empty string, it will produce an empty document.

Dealing with CSV files in FlexCel

As discussed in the previous section, there are 3 things you need to know when reading and writing CSV files: The locale, the separator and the encoding.

To specify the locale of the files you are reading or writing, you need to [change the FlexCel locale](#).

You specify the separator and the encoding when you call [ExcelFile.Export](#) or [ExcelFile.Save](#) to save the file, or when you call [ExcelFile.Open](#) or [ExcelFile.Import](#) to open the file.

Below is an example about how to open a file saved in a Spanish locale, using ";" as field separator and Win1252 as Encoding, and save the file using "," as field separator, with a USA locale, and UTF-8 encoding:

```
CultureInfo SaveLocale = System.Threading.Thread.CurrentThread.CurrentCulture;
CultureInfo SaveUILocale = System.Threading.Thread.CurrentThread.CurrentUICul
ture;
System.Threading.Thread.CurrentThread.CurrentCulture = new CultureInfo("es-
ES");
try
{
    System.Threading.Thread.CurrentThread.CurrentUICulture =
System.Threading.Thread.CurrentThread.CurrentCulture;
    try
    {
        XlsFile xls = new XlsFile(true);
        xls.Open("test.csv", TFileFormats.Text, ';', 1, 1, null, Encoding.Get
Encoding(1252), true);
        System.Threading.Thread.CurrentThread.CurrentCulture = new
CultureInfo("en-US");
        System.Threading.Thread.CurrentThread.CurrentUICulture = System.Threa
ding.Thread.CurrentThread.CurrentCulture;
        xls.Save("result.csv", TFileFormats.Text, ',', Encoding.UTF8);
    }
    finally
    {
        System.Threading.Thread.CurrentThread.CurrentUICulture = SaveUILocale;
    }
}
finally
{
    System.Threading.Thread.CurrentThread.CurrentCulture = SaveLocale;
}
```

IMPORTANT

As if it wasn't enough with all the issues discussed in this documents, CSV files can also be slow to read. This is because for every cell we read, we have to figure out the cell type. Please read the [CSV section in the Performance Guide](#) for more information.

Conclusion

It would make (a lot of) sense if we all agreed in a single standard so we would be able to generate "universal CSV files" which everyone could read or write, but that is just not the way it is.

You can use CSV files to interchange data with known applications, but using CSV as a "general" interchange format for unknown users is not a great idea. For example, don't provide a link to download a CSV file in a website unless you know all your users will use the same settings when opening that file. Providing an xlsx file instead is a much safer way to ensure everyone can read it.

Embedding Excel files in your application

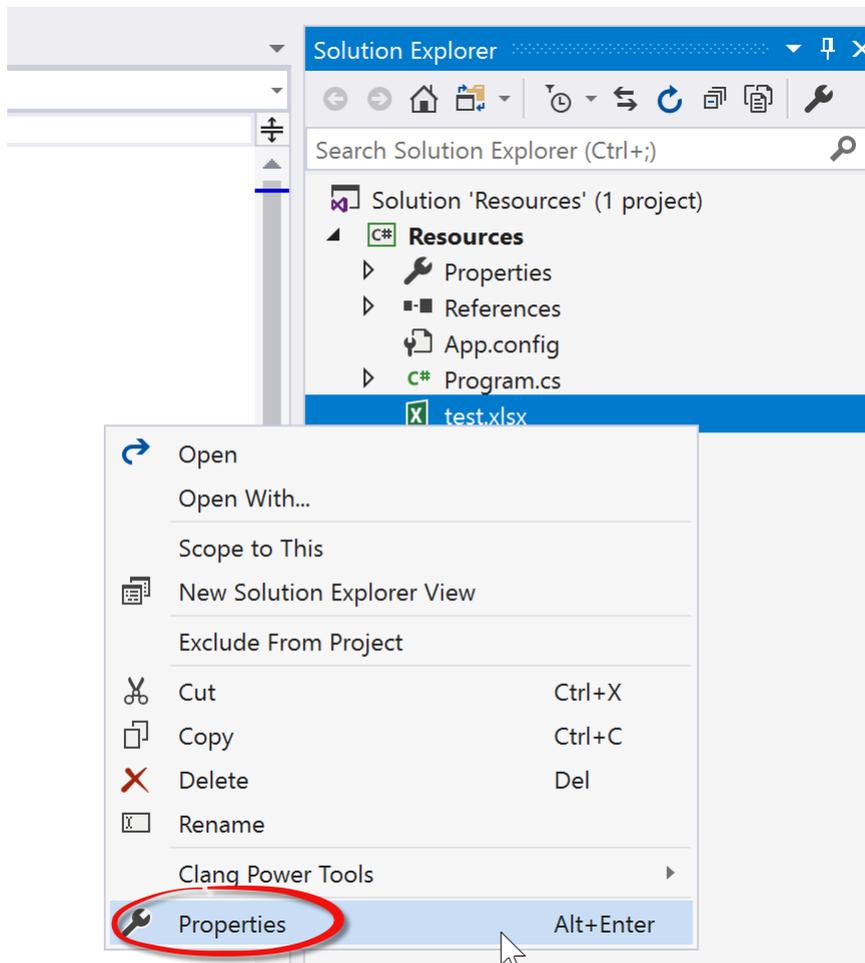
Sometimes, you need to include either templates for reports or pre-made Excel files inside your application. The simplest option is just to deploy your xls/x files together with your application, but you might want to embed them in your executable instead.

One way you might think of to include the xls file is using APIMate to "convert" the file into code, then paste that code inside your application. But while that would work, it isn't really a nice solution:

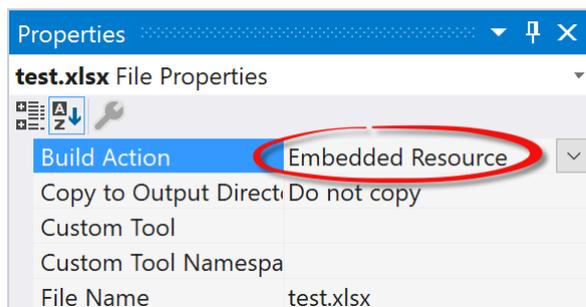
- The code generated by APIMate is very repetitive. So for example if you have the cells from A1 to A100 with the numbers 1 to 100, APIMate will write 100 calls to `ExcelFile.SetCellValue(1, 1, 1)`, `ExcelFile.SetCellValue(2, 1, 2)`, and so on, instead of calling `ExcelFile.SetCellValue(row, 1, row)` in a loop with row from 1 to 100. The idea of APIMate is to show you how to do stuff, so for example you want to know how to format a cell conditionally, you conditionally format it in Excel, then open the file in APIMate and look at the code APIMate generates. But it is not great to actually convert files to code.
- For the reason above, the code will be big, resulting in a bigger executable than if you embedded the file directly, and it will be slower to compile too.
- Embedded files as resources will normally open faster than the code that APIMate generates uses to create the file. Again, the code generated by APIMate is not designed for this, and it can be inefficient as it sets every cell with a different call.
- Resources are simpler to update. Just edit the file in Excel, save it and rebuild your app. While if you used APIMate, you would have to relaunch APIMate and re-copy-paste the code every time.

So how do you embed a file as a resource?

1. In Visual Studio, go to Menu->Project->Add Existing Item... (or press Shift+Alt+A)
2. In the dialog to select the file set the filter to All Files (*.*) and select the xls/x files you want to embed.
3. In the solution explorer, right click in the file(s) you added and select "Properties":



1. Set the **Build Action** of the files to **Embedded Resource**:



1. Now, to open the file, use the following code:

```

Assembly asm = Assembly.GetExecutingAssembly();
using (Stream InStream = asm.GetManifestResourceStream("Resources.test.xlsx"))
{
    var xls = new XlsFile(InStream, true);
}
  
```

Where the name of the resource is "Namespace.NameOfTheFile" (In this case the namespace is "Resources" so the name of the resource is Resources.test.xlsx)

If you can't figure out the name of the embedded resources, you can use the following method:

```
static string[] GetEmbeddedFiles()
{
    Assembly asm = Assembly.GetExecutingAssembly();
    return asm.GetManifestResourceNames();
}
```

This code will return the names of all embedded files in your executable.

Internal numeric formats.

Excel numeric formats should in theory be simple. For example you set a cell format to be "00.00", then the number 1 is formatted as 01.00. But it gets more complex than that, and in this tip we are going to dig deeper into format strings. Most of the information here is Excel -not FlexCel-specific, but it helps to know about how format strings work in detail.

Localized format strings

The first thing you need to be aware of is that Excel uses different format strings depending on the language of the Excel version that you have installed. For example, in English Excel you could use "yyyy-mm-dd" (year-month-day) to print a date like "2000-01-01". But if your Excel is in Spanish, you would have to write "aaaa-mm-dd" (año-mes-día)

This sounds like a nightmare for sharing files: I write a file with my English Excel, using a format like "yyyy-mm-dd" so the cell looks to me like "2000-01-01", but when I share it with you and you open it in your Spanish Excel, you will see "yyyy-01-01" in the cells instead of "2000-01-01" since the file is using "y" for year and it should be using "a" for año.

But luckily it is not that bad. Internally, in the file all formats are stored in English and converted on the fly by Excel when showing them to you. This means that inside the xls or xlsx file the format will always be stored as "yyyy-mm-dd". When you open it in a Spanish Excel, Excel will display it as "aaaa-mm-dd". When you enter "aaaa-mm-dd" as a format in a cell in a Spanish Excel, it will be saved as "yyyy-mm-dd" inside the file. So the file can actually be shared between different languages, without issues. Some users will see "yyyy-mm-dd" when they open it and others will see "aaaa-mm-dd", but the file is the same internally.

This is the reason **you always have to enter format strings in English when using FlexCel**. It doesn't matter if you have a Spanish Excel installed and you see the format strings using Spanish names, what goes on the file is always English format strings. **You can use APIMate** to get the English format strings that you need to use in FlexCel even if you have a Spanish or another language Excel. Just enter the format string in your language in Excel, save the file, and open it with APIMate. It will show you the English format string that you need to use.

WARNING

While numeric formats in cells will be automatically converted as mentioned above, you have to be extra careful when using functions like [Text](#)

In those cases the format will **not** be automatically translated. The formula

`=Text(A1, "yyyy-mm-dd")` will **not** work correctly when opened in a non-English Excel. Similarly, the formula `=Text(A1, "aaaa-mm-dd")` will work in your Spanish Excel, but users with other languages will see it wrong.

Internal format strings

Ok, now we know that the format strings are stored always as English strings, except when inside formula functions. This means that a format like "dd/mm/yyyy" should look the same everywhere, shouldn't it?

Sadly not always. The most commonly used format strings are not stored as strings inside a file, but as format ids. This is probably a file size optimization for times where floppy disks had 1.44 mb, but it still is like this today. And as you might guess, those internal formats are also a problem, because they change with the language of Excel.

So for example, format number 20 is "h:mm" in an Excel from the USA, but "hh:mm" in an Excel from England. If you set the cell format in the USA to be "h:mm" and then send the file to a customer in England, he will likely see "hh:mm" when he opens your file.

NOTE

The internal formats, except for formats number 14 and 22, are hardcoded into Excel itself, they don't change with the machine locale. This means that if you buy a USA version of Excel, it will always read format 20 as "h:mm" even if you are in a machine with a British locale and your Windows settings are "hh:mm".

Formats 14 and 22 are different, and we cover them in the next section.

This creates a problem for FlexCel. What should we display when we find a format 20 in a file? There are no "American" and "British" versions of FlexCel, there is a single flexcel.dll file that is the same for all users over the world. So we can't hardcode "h:mm" for American versions of flexcel.dll and "hh:mm" for British versions.

What FlexCel does instead is to behave as an American Excel by default, but letting you customize the built-in format to whatever you want. The method you use for that is [XlsFile.SetBuiltInFormat](#)

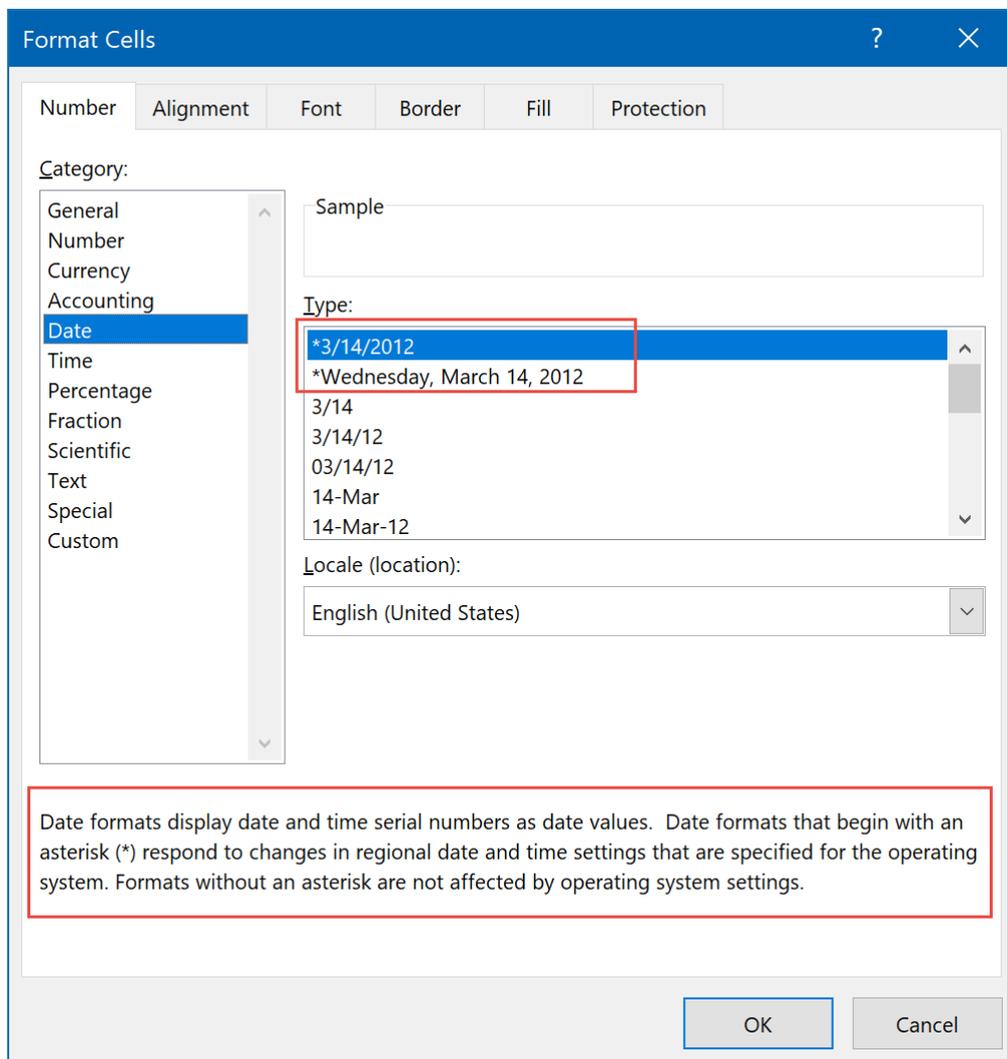
Please read the link above to see a list with every internal format and how it is interpreted by FlexCel.

Format strings that actually change with the locale

As it was mentioned above, all internal formats except 14 and 22 are hardcoded. But formats 14 and 22 behave differently: they change depending on your locale.

So if you save a file with the format "mm/dd/yyyy" in an American locale, this format will be saved as format 14. If you now change the Windows locale so dates are "dd/mm/yyyy", the same Excel will show the file differently.

Those two internal formats are shown by Excel starting with an "*", and it is explained in the format dialog that those formats change with the locale.



There are actually four formats that change with the locale. Two of them are for dates (long and short form) one is for date and time, and the last one is for time. Two of them use an internal format (14 and 22), the others use a [\$-F0000] syntax.

They are as follows:

1. **Short Date:** This is the internal format number 14. To set a cell to this format, set the format string to be RegionalDateString. RegionalDateString is a FlexCel variable which will change depending on your ShortDatePattern in your locale settings.

NOTE

This format is not directly the format string in your Windows settings converted to an Excel formatted string. While it will read the format string from the Windows settings, only some formats are allowed and extra stuff might be ignored. For example, if the Windows format string is "mm/dd/yyyy dd/mm" the Excel string will be "mm/dd/yyyy".

2. **Long Date:** This is not stored as an internal format, but as a format starting with the string "[\$-F800]". What goes after [\$-F800] doesn't really matter, but normally it is the format string for a long date in the language Excel was when creating the file. This allows third-party Excel readers which don't understand the [\$-F800] string to still display something.

So for example you could have the format string: "[`-$-F800`]dddd,\ mmmm\ dd,\ yyyy". As it starts with [`-$-F800`], it will be converted to a regional date by Excel (or FlexCel), and what is after the [`-$-F800`] will be ignored. But some other third-party Excel viewer which doesn't know about this will ignore the [`-$-F800`] and still display a long date, just not correctly localized.

3. **Date and Time:** This is the internal format number 22. To set a cell to this format, set the format string to be `RegionalDateTimeString`. `RegionalDateTimeString` is a FlexCel variable which will change depending on your locale settings.
4. **Time** This is not stored as an internal format, but as a format starting with the string "[`-$-F400`]". What goes after [`-$-F400`] doesn't really matter, but normally it is the format string for a time in the language Excel was when creating the file. This allows third-party Excel readers which don't understand the [`-$-F400`] string to still display something.

So for example you could have the format string: "[`-$-F400`]h:mm:ss\ AM/PM". As it starts with [`-$-F400`], it will be converted to a regional time by Excel (or FlexCel), and what is after the [`-$-F400`] will be ignored. But some other third-party Excel viewer which doesn't know about this will ignore the [`-$-F800`] and still display a time, just not correctly localized.

NOTE

Those formats will change in FlexCel depending in your machine locale, just as they would in Excel. But you can change what FlexCel uses without needing to change the Windows locale. See the link about [how to change the FlexCel locale](#)

TIP

Besides the formats [`-$-F800`] and [`-$-F400`] you can actually use any locale with the same syntax. For example, the string: [`-$-409`]m/d/yy h:mm AM/PM;@ is formatted with English locale. Those formats are what is shown at the bottom of the "Format cell" dialog, and FlexCel also fully supports them.

But those formats don't change with your machine locale: They show the same everywhere so they aren't really a problem and this is why we don't discuss them in this tip.

How to avoid internal format strings

As we have seen before, if you have an Excel from the USA and set the cell format to be "h:mm", this will be saved as format 20, not "h:mm" in the file. This format 20 then can be displayed differently to different customers with different localized versions of Excel.

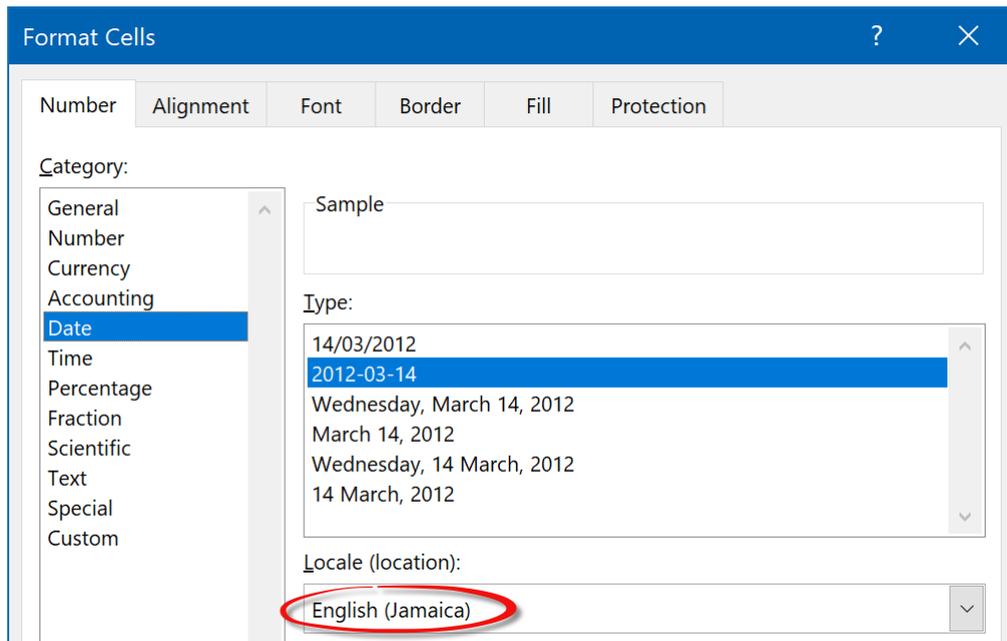
FlexCel behaves the same: If you set the format to be "h:mm", it will realize that it matches internal format 20, and save the internal format 20 into the file.

So how do you set the format to be "h:mm" for everybody, and not the Excel-Locale-Dependent format 20?

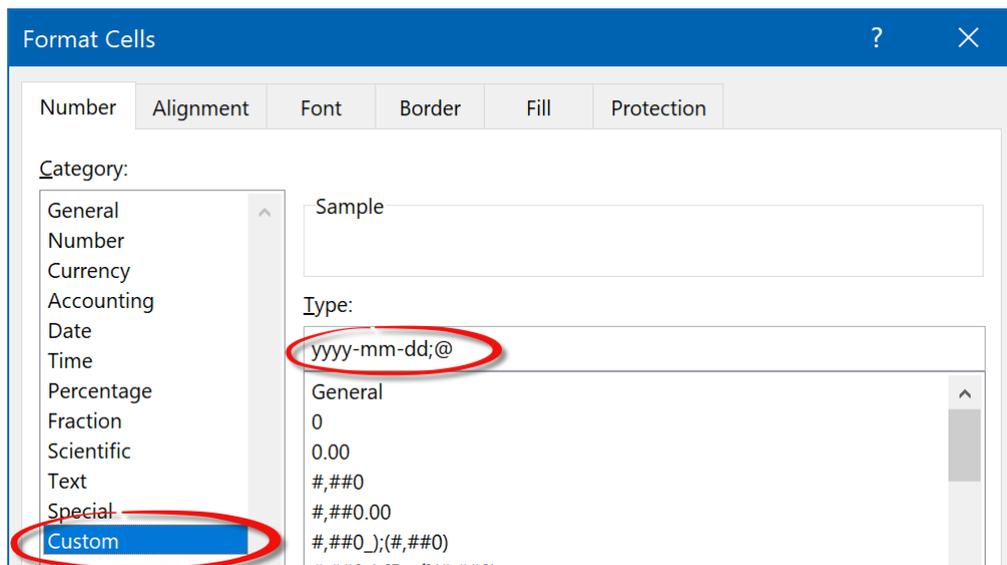
The bad news is that there is no easy way to do exactly that. Whenever Excel sees "h:mm" it will be converted to format 20. But the good news is that it is very simple to do something that has the same effect: ****You can set the format to be "h:mm;@" ****

As you can see [here](#) this is a format string with two parts: "h:mm" for positive numbers and zeros, and "@" for negative values (Excel doesn't support negative dates, so the value here doesn't really matter). "h:mm;@" works exactly the same as "h:mm", but due to the extra @ it won't match the internal format and won't be converted. To match the internal format the string must be exactly "h:mm".

This is the trick Excel uses when you select a "locale" in the format dialog. Let's for example select a "Jamaica" locale:



If you then go to "Custom" without closing the dialog, you will see that the format added was **yyyy-mm-dd;@**:



The @ at the end makes sure this won't be replaced by an internal format.

How to find out the internal formats in your Excel version

As discussed, every localized Excel version has its own version of the internal formats. Also as discussed, we can't ship a different flexcel.dll for every locale, so our flexcel.dll comes localized by default as a USA Excel. And finally as also mentioned, you should try not to use localized formats if you care about the differences, by adding a ";"@ at the end.

But what do you do if you are not creating the files, and your users are sending you files with internal formats created with their localized Excel versions?

Let's say you live in Spain, and your users are creating files with Spanish Excel with a format "dd/mm/yyyy". You are opening those files with FlexCel and exporting them to pdf, but in the export the dates appear as "mm/dd/yyyy" since FlexCel by default is localized as USA, and is behaving the same as if you had a USA Excel.

You can try to explain all of this to your users, but it is likely that they won't understand: They saved the files as "dd/mm/yyyy" and they expect the pdf to be "dd/mm/yyyy".

As we have learned in this article, the solution is to use [XlsFile.SetBuiltinFormat](#) to make FlexCel behave as a Spanish Excel. But how do you know which string to use in every one of the internal formats?

We can't sadly provide every translation from every locale Excel is translated too. And notice that even US locale is different from UK locale, so it is not just about different languages.

But what we can do is to give you a spreadsheet, which has all the relevant built-in formats in column B. This file has a user-defined function which will calculate the localized format on column B using the [NumberFormatLocal VBA property](#). Then it shows you the code you need to write on columns F and G, in C# or Delphi respectively. If you are using a different programming language, it should be simple to change the formulas in either column F or G to return the calls in your language.

You can get the spreadsheet from here:

[all-builtin-formats.xlsm](#)

TIP

Remember to enable macros in this spreadsheet so the user-defined function will work.

How to change the FlexCel locale.

Some formats in Excel change depending on the locale of your machine. We have discussed those formats more in depth in the tip about [internal numeric formats](#).

When FlexCel renders a file to pdf or other format, it needs to know how you want those locales printed. Is it dd/mm/yyyy or mm/dd/yyyy? As you would expect, if you don't do anything, FlexCel picks the locale from your machine settings. If you want to change how FlexCel renders those locale-dependent formats, the simplest way is to just change the machine settings. But sometimes you can't or don't want to change the machine locale, but still want FlexCel to render those formats as if the machine had some specific locale.

In .NET it is simple to change the locale, you can just change the thread locale by changing [CurrentCulture](#) and [CurrentUICulture](#)

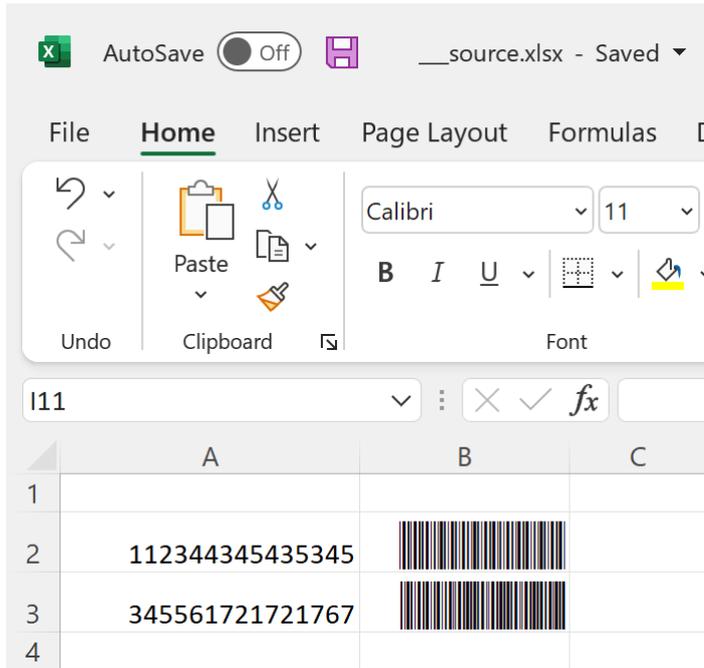
You might change them at the start of your application (or when creating new threads), or you might change it before working with FlexCel and set them back after the FlexCel work is done, as in the example below. You might even have different threads working in different locales.

Example:

```
CultureInfo SaveLocale = System.Threading.Thread.CurrentThread.CurrentCulture;
CultureInfo SaveUICulture = System.Threading.Thread.CurrentThread.CurrentUICul
ture;
System.Threading.Thread.CurrentThread.CurrentCulture = new CultureInfo("zh-
CN");
try
{
    System.Threading.Thread.CurrentThread.CurrentUICulture =
System.Threading.Thread.CurrentThread.CurrentCulture;
    try
    {
        XlsFile xls = new XlsFile("chinese.xlsx", true);
        using (FlexCelPdfExport pdf = new FlexCelPdfExport(xls, true))
        {
            pdf.Export("chinese.pdf");
        }
    }
    finally
    {
        System.Threading.Thread.CurrentThread.CurrentUICulture = SaveUICulture;
    }
}
finally
{
    System.Threading.Thread.CurrentThread.CurrentCulture = SaveLocale;
}
```


Using barcodes.

One question we get regularly is how to embed barcodes using FlexCel. Excel doesn't have any barcode-specific functionality, so the ways to do that are to either insert an image with a barcode, or write text with a special font which will render the text as the barcode representation.



Using a barcode font

This is the simplest, and in my opinion best method to get barcodes into Excel. Just install a barcode font, then write the barcode in text, and it will be rendered as expected.

Advantages:

1. The font is vectorial, not a bitmap, and fonts are great for scaling. That means that if you zoom in the page or print with a high-resolution printer, you won't see a pixelated image. It will always be crisp just like any other text in the document, because it is text, just rendered with a special font.
2. The generated file will probably be smaller. If you have thousands of barcodes, each barcode will be stored as an ASCII text instead of as an image. It can make a big difference.
3. The file can be easily edited by another user once you created it. They only need to change the text and the barcode will update.
4. It is just simpler to write the text in FlexCel to a cell than to generate an image and insert the image.

Disadvantages:

1. You need to find the special font that renders the barcode and you need and make sure you are legally allowed to redistribute the font. There are lots of free fonts on the internet, such as <https://www.barcodesinc.com/free-barcode-font/> but you will have to find one that suits your needs.
2. With images you might get more complex barcodes than with a font. Say for example a barcode with the shape of your logo, or with annotated text.
3. In Excel you can't embed fonts, so to redistribute an xls or xlsx file using a barcode font, you need to make sure your users also have the font installed in their machines. If they don't have the font installed, they will see the text but not the barcodes.

NOTE

When exporting to PDF, not only you can embed the fonts, but also this is what FlexCel does by default when exporting. So a PDF generated with a barcode font will look great anywhere, no matter if the user seeing the pdf actually has the font installed on his machine or not.

Using an image

If a barcode font won't work for you, then the other option is to insert an image with the barcode where you need it. You will still need a separate component for creating the barcode image, and it is best if you can export it to wmf or emf since those are vectorial formats which will look better at higher resolutions. If you can only get pngs, make sure they are of a resolution that is good enough for printing. Try not to use jpegs as they are bad for this kind of images and can have a lot of artifacts.

Using images is more complex than using a font, files will be bigger, and basically all "Advantages" in the previous section become the "Disadvantages" of using images. But on the nice side, the users won't need to have any font installed to see the barcodes. So if you need to send the xls/x files to random users, an image might be the only sane choice. Once again, remember that pdf files don't have this problem, so you might want to still use a barcode font, but redistribute the pdfs instead of the xls/x files.

How to get the hyperlink in a given cell?

Different from other cell properties like the background color, Excel hyperlinks are not stored tied to any cell. In fact, hyperlinks can cover more than one cell: you might have a hyperlink covering a range from A1 to C2 and the link will trigger if any of those cells is clicked.

So hyperlinks are stored in a list, and in FlexCel you have methods to know the number of hyperlinks ([ExcelFile.HyperLinkCount](#)), to retrieve the range of cells one hyperlink in the list applies to ([ExcelFile.GetHyperLinkCellRange](#)) and of course to get the information on the link like the url where it goes ([ExcelFile.GetHyperLink](#))

If you wanted to know if say cell A3 has a hyperlink, a way to find out would be to loop from 1 to HyperLinkCount and call GetHyperLinkCellRange in each entry of the list. If the range contains A3, you have found the hyperlink, and you can now call GetHyperLink to get the url of it.

But of course, this can be very slow, if you have thousands of links and you want to get the links in cell A2, A3, A4... For each one of the cells, you would be looping over all the list to find the correct links.

What we need here is an indexed search, so you spend $O(Ln)$ instead of $O(n)$ in each search. But as the search is in 2 dimensions (rows and columns) we need spatial indexing. Luckily, FlexCel offers a method [ExcelFile.LoopHyperLinks](#) that will do just that.

LoopHyperLinks will search for the links that apply to a range of cells, then call an anonymous method for every one of the links that apply. This will be much more efficient than a dumb loop over all links, since LoopHyperLinks internally holds a spatial index to locate the links efficiently.

Take a look at [ExcelFile.LoopHyperLinks](#) for a code example on how to get the links to a cell.

Why are xlsx files generated by FlexCel smaller than the same files generated by Excel?

This is a question we get from time to time: You take an existing xlsx file, open it with FlexCel and save it, without doing anything else. You will notice that the new file is several kilobytes smaller than the original, and yet, nothing changed. What is going on?

There can be actually many causes for this, but the main reason is simple: Xlsx files are zip files with a different extension. In a zip file, you can specify the compression level ranging from "**fastest**" (less compression) to "**best**" (slowest).

Excel is an interactive tool, and it is really important that when a user presses enter the file is saved as fast as possible and control returned to the user. So Excel uses the "fastest" compression setting. On the other hand, FlexCel is optimized for server use, and in this case, smaller sizes win over some milliseconds less to generate the file. So we use the "default" compression setting which is normally the best compromise between speed and compression. And this is why the files generated by FlexCel are normally smaller than the same files as generated by Excel.

NOTE

Comparing file sizes of zip files is not really indicative of anything, but if you want to verify this, you can easily do the test:

1. Create a file in Excel
2. Open and save it with FlexCel
3. Open the file generated in 2. with Excel again and save it. The file size for this file should be similar to the file in 1., and bigger than the file in 2.

Finally, note that while "default" compression is normally the best compromise, you can change the compression of the files generated by FlexCel changing the [ExcelFile.XlsxCompressionLevel](#) property.

TIP

You might also see the opposite case: You generate a file in FlexCel and when you open it and save it with Excel, the size is smaller.

This is normally because a setting in Excel. If you go to the Ribbon -> File -> Options -> Advanced, you will see the following options:

Excel Options

The screenshot shows the 'Excel Options' dialog box with the 'Advanced' tab selected. The left sidebar lists various categories: General, Formulas, Data, Proofing, Save, Language, Ease of Access, Advanced (highlighted), Customize Ribbon, and Quick Access Toolbar. The main area contains several settings:

- Open supported hyperlinks to Office files in Office desktop apps
- Pen**
- Use pen to select and interact with content by default
- Image Size and Quality** (Book1)
- Discard editing data ⓘ
- Do not compress images in file ⓘ (highlighted with a red box)
- Default resolution: ⓘ 220 ppi
- Print**

If the checkbox isn't checked (the default) Excel will reduce the resolution of the images in the file to match the given resolution. This will result in a smaller file with lower-resolution images.

On the other side, FlexCel will never try to automatically reduce the image quality. If you enter an image with AddImage, it will enter it "as-is" and assume you wanted that resolution. To get smaller images with FlexCel, reduce the image quality before entering the image into the file.

Automatically open generated Excel files.

In the FlexCel demos we follow a common pattern: When we generate a file we ask the user where he wants to save it, and then, we offer to open the generated file in Excel.

The code is something similar to this:

```
if (saveFileDialog1.ShowDialog() == DialogResult.OK)
{
    xls.Save(saveFileDialog1.FileName);
    if (MessageBox.Show("Do you want to open the generated file?", "Confirm",
        MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
        using (Process p = new Process())
        {
            p.StartInfo.FileName = saveFileDialog1.FileName;
            p.StartInfo.UseShellExecute = true;
            p.Start();
        }
    }
}
```

Now, if you have used OLE Automation before FlexCel, you are probably used to a different pattern: In OLE you can launch Excel, fill it with data using Automation, and then leave the file open without saving it. And while I personally think saving the file first is nicer, there might be cases where you want to emulate this behavior of just opening the file without asking the user to save it first.

The bad news is that it is not technically possible to create a file in FlexCel and have it open in Excel without saving it somewhere first. On the other side, the good news is that you can reasonably emulate the behavior and make it look to the user as if the file wasn't saved on disk first.

This trick uses [Excel templates](#), which have an extension **xlt** in Excel 2003 or older (equivalent to xls) and **xltx** in Excel 2007 or newer. An Excel template is similar to an xls/xlsx file, but it has two different characteristics:

1. When you open an xltx file in Excel, Excel makes a copy in memory of it, and doesn't lock the original file. This means that you can remove the template once Excel has opened it without worrying that Excel will be using it.
2. When a user presses "Save" in Excel, Excel won't save the template but show the "Save As..." dialog and let the user choose a filename for saving.

So what we are going to do is to create a temporary xltx file instead of a regular xlsx file, open it in Excel, then wait a little to make sure Excel finished loading it, and then remove the temporary file. To do so, we can use code like this:

```

string tmpFileName = Path.GetTempPath() + Guid.NewGuid().ToString("D") + ".xl
tx";
xls.Save(tmpFileName);

//The verb to open a Xltx template is "New", not "Open".
//Process.Start will use the default verb, which is also "New",
//So there is no need to specify it.
//Note also that latest .NET versions won't start a file without shellexecute.
using (Process p = new Process())
{
    p.StartInfo.FileName = tmpFileName;
    p.StartInfo.UseShellExecute = true;
    p.Start();
}

Task.Run(() =>
{
    //Wait 30 seconds to delete the file, so Excel has time to open it.
    Thread.Sleep(30000);
    File.Delete(tmpFileName);
});

```

NOTE

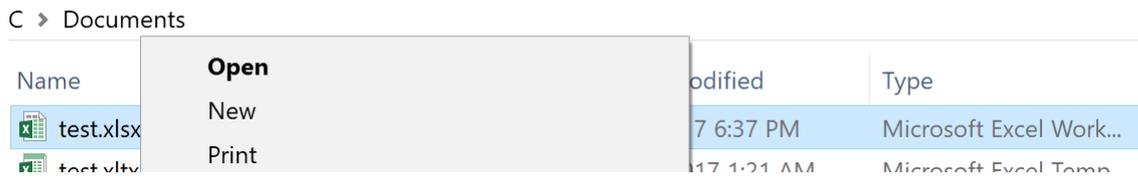
Once you use `Process.Start` to open the file, you have to wait until Excel finished loading it. In the code above we wait for 30 seconds before deleting the file, but if your files are huge or the machines where Excel is installed are too slow, you might want to increase the time before you try to delete the file.

NOTE

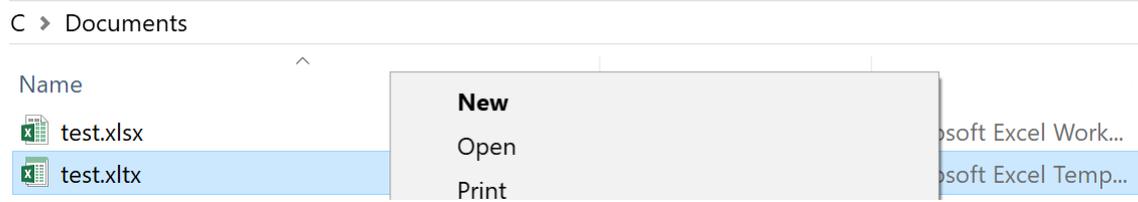
When saving a file to disk, FlexCel automatically detects from the extension that it should be saved as a template, so you don't need to do any extra work. Just save the file with an xlt/x extension and it will be saved as a template. But in other cases like when saving to a stream, FlexCel can't figure out that you want to save as a template, and you will have to explicitly say so. You can tell FlexCel that the file is a template by setting the property [ExcelFile.IsXltTemplate](#) to true.

NOTE

When starting the process to open the file, remember that the action needed to open the template in "template mode" is **New**, not **Open** as it is the default in most cases. For example, when you right-click in an xlsx file, "Open" is the default action:



But when you right click in an xltx file, "New" is the default action:



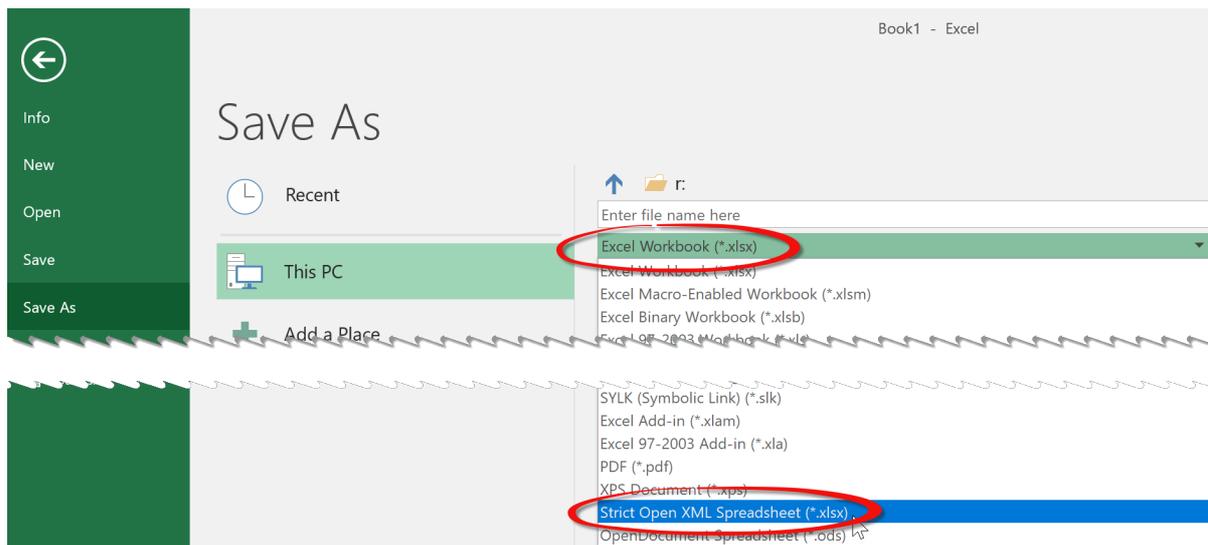
If you opened a template with "Open" instead of "New", it would behave like a normal xlsx file, being locked by Excel while in use. So in the code above, we used the default action instead of the default "Open" action. Had we called:

```
var startInfo = new ProcessStartInfo(tmpFileName);
startInfo.Verb = "Open";
startInfo.UseShellExecute = true;
Process.Start(startInfo);
```

Then the trick wouldn't work.

Deciding between standard xlsx files and strict xlsx files.

FlexCel 6.17 introduced full support for reading and saving strict xlsx files. Strict xlsx files are a different file format from normal xlsx files, and you can select them when doing a "File Save As..." in Excel:



Now the question is: Should I use normal or strict xlsx files? And the short answer is: **Use normal xlsx files** unless you have a non-negotiable requirement from your customers to use strict xlsx.

The long answer: When Microsoft introduced xlsx back in Excel 2007 they didn't have the time to define all the parts of the new file format, and so for some parts (mostly drawings) they just reused an older xml file format: [Office XML Formats](#).

Using the existing file formats for the parts that weren't ready gave them a head start and allowed them to ship xlsx earlier, but it introduced at least two problems:

1. The style and namespaces for the xml are different. This is mostly a cosmetic issue, but for example the old xml format uses pascal case for the identifiers (for example "LegacyDrawing") while the new system uses lower camel case (legacyDrawing). The old xml also relies more on attributes while the new one relies more on xml children, store booleans differently and a long list of etc. In general the xml in those legacy parts is very different from the rest of the newer parts. To read or write to them, they require a lot of specific code which is different from the code to read or write similar non-legacy parts.
2. More importantly the old format wasn't always valid xml. So for example the text of a button with an enter in the text would be stored as "Some text
Second line." This will crash any compliant xml readers, because xml readers are required to fail when they find invalid xml and
 is not valid xml as there is no ending tag.

So in order to fix this and other issues, Microsoft has been developing a parallel "Strict xml" standard which uses different xml namespaces and has no legacy parts. They renamed the "normal xlsx" to "transitional xlsx" and added the "strict xlsx" variant.

But the support for strict xlsx in Excel came late, with Excel 2010 being first able to read it, and Excel 2013 first able to write them. By that time, transitional xlsx was already the standard, and that hasn't changed up to the day I am writing this hint. And if strict xlsx hasn't gained traction yet, it is unlikely that it ever will. (even when I would like to be proven wrong as the strict xlsx format is simpler and free of legacy baggage)

So coming back to the answer: Normal xlsx is the standard, and the format that everyone else will understand (including old FlexCel versions). So if you have no specific need for the strict xml parts, just keep using the standard.

The strict xlsx format on the other side should be cleaner, but being "cleaner" is not a property of a file format that should concern you much. The fact that everyone can read the files you create should be a bigger concern. And actually, **strict xlsx files as saved by Excel aren't cleaner** because they still include the legacy parts inside conditionals so older Excels as Excel 2007 can still read them. And on the other side, "normal" xlsx files still include the new, non-legacy parts too for tools that can't understand the legacy parts, so both file formats are basically the same, except for the different xml namespaces and some syntax differences.

NOTE

As mentioned above, when saving a strict xlsx file with Excel it will still include all the legacy and potentially invalid xml parts. We believe this defeats the purpose of having a cleaner format, so when you save with FlexCel, we will only write the non-legacy parts. You will need to have Excel 2010 so it understands the new non-legacy parts, but you also need Excel 2010 for strict files saved with Excel, since the xml namespaces are different so older tools won't understand the files. If an xlsx reader knows about strict xml and the strict xml namespaces, it should be able to handle the non-legacy parts too.

Finding out which FlexCel version you are using

Sometimes you want to know the exact FlexCel version you are using from inside your app, and you might also want to make sure you aren't using a trial. You can use the following code to find out:

```
System.Reflection.Assembly assembly = System.Reflection.Assembly.GetAssembly(typeof(ExcelFile));

//Find out the version.
var version = assembly.GetName().Version;
Console.Write(version.ToString());

//Find out if it is a trial or a registered assembly. Trial has the word
"TRIAL" in the title.
var title = assembly.GetCustomAttributes(typeof(System.Reflection.AssemblyTitleAttribute), false)
    .OfType<System.Reflection.AssemblyTitleAttribute>()
    .FirstOrDefault().Title;
Console.WriteLine(title);
```

Sign your PDF files

A little story

Some time ago I requested some official documents from my government, and they provided them to me in PDF format. This is all as expected: PDF is the de-facto standard for digital documents.

But what I didn't expect was that those documents weren't digitally signed. This doesn't make them completely useless, but the fact that they are not digitally signed diminishes a lot their value for me. If in the future I need to use those documents myself to prove anything, I can't really prove that I didn't create those documents myself, or that I modified the data on them. Can you imagine them giving you any old-fashioned paper document that is not signed and/or rubberstamped?

The government should digitally sign all the digital documents they give us, in the same way they physically sign their paper documents. And if you are creating documents that can be used as proof of anything, so should you.

What is a digital sign

Digitally signing a PDF document is a way to prove that the document is the same as the one you produced, and it wasn't altered by anyone else. Someone might change the document, but the signature will become invalid. They might re-sign the document, but they would have to sign it with their signature, not with yours. The only way they can have a PDF signed by you is if you gave that exact PDF to them.

PDF signing is also a proven technology based on solid asymmetric encryption. It is not something that you (or anyone else) can fake with a 5-minute google search, and neither with months of hard work. A digital sign is actually much more secure than a physical sign, as long as you use a secure encryption algorithm.

WARNING

When working in FlexCel .NET, the examples used the default algorithm in a CMSSigner, which happens to be [SHA1](#). SHA1 is not considered secure anymore, so we have updated the examples to use SHA512. If you have old code, you might want to update it to use SHA512 too. In Delphi, we already use SHA512 by default.

How to keep your signing secure.

As mentioned in the previous section, signing is a safe way to ensure your documents are not altered. But you need to do it right, and in this section we will discuss some of the stuff that you need to care about.

Keep your key secure

The most important thing you need to ensure is that **nobody can get your certificate with your private key**. No matter how good the lock on your door, if the attacker gets the key, he will be able to enter. Or in this case, if he gets your signing key, he will be able to sign as if he was you.

In practice this means that you **can't bundle the certificate with your app**, even if you encrypt the certificate. If your application can read the certificate, a hacker can too. The only secure way to keep the certificate safe is to sign the PDFs in a server that your users don't have access to.

Keep up to date with the signing algorithms

As we advance in our technology, vulnerabilities can be found in the algorithms used to sign, and you need to make sure you aren't using an algorithm that is considered vulnerable. At the time of this writing, this means to use SHA256 or SHA512 and not SHA1 as [SHA1 is known to be non-secure](#)

Should SHA2 or SHA3 become vulnerable in the future, you will need to update the signing algorithm again.

Get a real certificate from a trusted CA

You can sign your PDFs with self-signed certificates and it will work, but then, nothing is preventing an attacker from creating his own self-signed certificates too, where he claims to be you.

If you teach your users to trust your self-signed certificates, they might trust other self-signed certificates too.

Getting more information

You can read more about how to sign PDF files with FlexCel at the [PDF guide](#)

Appendix: Also use PDF/A

As we are discussing how to best ship PDF files, I would also like to mention that it is a great idea to ship your documents as [PDF/A](#). Basically, a PDF/A file is a file designed so "it can be still be read 100 years from now". And why wouldn't you want your documents to be still readable 100 years from now? Documents created 20 years ago (like a wks lotus 123 file) might be already difficult to open today. Will it still be possible to open them 100 years from now?

Of course nobody can guarantee that any file (not PDF, nor PDF/A) will still be readable in the future. But PDF/A does have some characteristics that make it likely. For example:

- It is "frozen in time". Different from normal pdf which is constantly evolving, the PDF/A file format is done. It will never change, and it will be the same 100 years from now. Nobody knows how the PDF file format might have evolved by then.
- It removes complex and unnecessary stuff which is present in normal PDFs like JavaScript, Audio and Video.

- It requires that everything is self-contained in the document. So for example the fonts that you use (which might not be available in the future) must be embedded.
- There are multiple independent validators to check if a PDF/A file honors all the requisites. FlexCel currently uses 3 different validators to ensure that the documents it generates are valid.

It doesn't mean that there are not drawbacks too: By making your files self-contained they will be larger, and some PDF features you might want to use might not be available. But in most cases, if your documents are meant to last, you should consider PDF/A.

There are many varieties of PDF/A and all of them are supported by FlexCel. If you are not sure on what flavor to use, I would recommend PDF/A2, as PDF/A1 is a little too restrictive, especially because it doesn't support the modern signing algorithms we use in FlexCel. So if you want to sign **and** PDF/A your files, you can't use PDF/A1. PDF/A3 is similar to A2, but it allows attachments. Use A3 only if you want to attach files to your pdf documents.

You can read more on how to create PDF/A files in the [PDF guide](#)

Conditionally format all things!

In Excel 2007, Excel completely changed conditional formatting, turning it from a very useful tool to something that you just need to know about.

NOTE

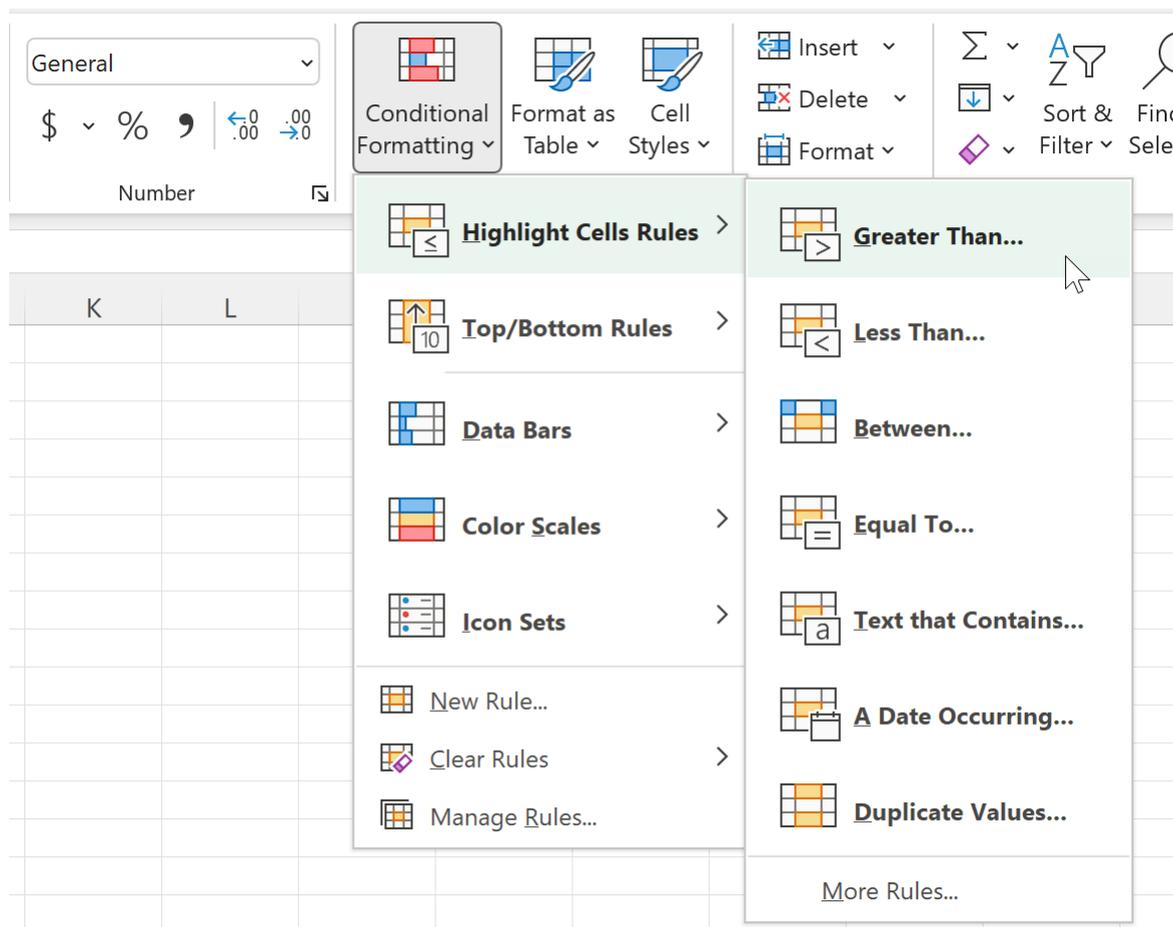
FlexCel has supported the old conditional formats basically since forever, but it also added complete support for the new Excel 2007 conditional formats in version 6.9.

This means that in the current FlexCel version you can add, remove and modify any existing conditional formatting in any file. Every little thing is supported up to the features introduced in Excel 2016 (the latest version at the time of writing this tip). And FlexCel will also calculate all of them and export them to PDF, HTML, SVG, etc.

If you weren't using them before because FlexCel didn't support them, make sure to give them a second look now.

As usual [ApiMate](#) will tell you everything about how to enter a conditional format with code, but **conditional formats are especially useful when doing reports.**

Let's imagine you want to paint a cell in red if it is bigger than 10. You could do this with tags like `<#Format cell>`, but that can get tiring soon and it won't be so simple to maintain. It is much simpler to just press the "Conditional Formatting" button in the home tab of the ribbon:



Then go to "Highlight cell rules" and select "Greater than" in the cell that you want to format. Then save the template, and when you run the report, if the value is bigger than 10 it will be red. No tags or code required!

But this is just scratching the surface. Conditional formats will not only allow you to do stuff that is possible with normal formatting (with more work), they will also allow you to do stuff that is just not possible at all without them. Like for example:

Data bars

	A	B
1	 1	
2	 4	
3	 5	
4	 7	
5	 8	
6		

Color scales

	C	D	E
1	0.360445	0.017858	0.274937
2	0.583899	0.351879	0.92607
3	0.693038	0.679804	0.525019
4	0.695422	0.886188	0.922286
5	0.890098	0.444989	0.525459

Icon sets

	F	G
1	 987	
2	 520	
3	 394	
4	 856	
5	 1	
6	 956	
7	 59	
8	 115	

Conditional formats allow a full new world of possibilities to the formatting of your documents, and better of all, they are fun to use. FlexCel currently supports everything you can throw at it, up to the latest features in Excel 2016.

NOTE

Not only FlexCel supports all the features in conditional formatting, it sometimes supports them better than Excel itself!

For example, iconssets in Excel (up to the latest Excel 2021) are bitmaps, which means that they won't look so good when zoomed in or printed.

This is a screenshot of one of the icons above with Excel at 350% zoom:



856

To be fair to Excel, they did increase the resolution of those bitmaps in newer versions, they used to look much more jagged. You might also need a high-dpi monitor to see the differences better.

On the other hand, FlexCel will draw those icons as vectors and export them as vectors to pdf, which means that they will look great no matter the resolution or size of the final result.

This is a screenshot of the same icons as exported to pdf by FlexCel, at 400% zoom:



856

As we value conditional formats so much, we put a lot of effort and months of work to make sure every little detail is right: From the rendering of icons and databars to the performance and memory usage with huge files. Take advantage of them.

Expanding formulas in consecutive cells

One common thing you might want to do when entering formulas in FlexCel is to change the column or rows where they appear. So for example, let's imagine you have this formula in A1:

```
= A2 * 2
```

And you want to expand the formula to Columns **B** to **Z**. In **B1**, you will want the formula **=B2 * 2**, in **C1** you will want **=C2 * 2** and so on.

	A	B	C
1	= A2 * 2	= B2 * 2	= C2 * 2
2			

There are multiple ways to achieve this:

1) You can enter the formula in A1:

```
xls.SetCellValue(1, 1, new TFormula("= A2 * 2"));
```

And then copy the cell to the range **B:X**:

```
xls.InsertAndCopyRange(new TXlsCellRange(1, 1, 1, 1), 1, 2, 25,
TFlxInsertMode.NoneRight, TRangeCopyMode.All);
```

This will work the same as in Excel, and the formulas will be adapted when copied. Same as in Excel *absolute references* (like for example \$B\$1) won't be changed, but **relative references** will change when copied.

2) You can manually create the formulas by using [TCellAddress](#)

TCellAddress is the record you use in FlexCel to convert cell references from/to numbers to letters. Here is a little example on how you can get the **row** and **column** from the string "B5" and also how to get the string "B5" from the row and column:

```
// From string to number
// We will extract the row (5)
// and the column (2) from the reference "B5"
var a = new TCellAddress("B5");
xls.GetCellValue(a.Row, a.Col);

// From numbers to string
// We will get the string "B5" from the row (5)
// and the column(2)
a = new TCellAddress(5, 2);
DoSomething(a.CellRef);
```

So, for our original example, we could use some code like this:

```
for (int col = 1; col <= 26; col++)
{
    xls.SetCellValue(1, col, new TFormula("= " + new TCellAddress(2, col).CellRef + " * 2"));
}
```

3) Using **R1C1** notation. **R1C1** is an alternative notation to the classical **A1** notation, to describe formulas based in their rows and columns, instead of in a letter and a number.

R1C1 is completely equivalent to A1, but has the advantage of always using column numbers, and that the cells are relative to their position, which is what you usually want.

NOTE

You can find a lot of information in R1C1 cell references internet just by a web search, and I recommend you to search for R1C1 if you weren't aware that this mode existed. As it makes no sense to repeat all the information in this doc, here we will focus on how to use R1C1 from FlexCel.

There are a couple of properties that govern R1C1 in FlexCel:

- `ExcelFile.FormulaReferenceStyle = TReferenceStyle.R1C1;`

This will affect how you can insert the formulas in FlexCel, but it is independent of how Excel will show them. To change how Excel displays R1C1 formulas, you need:

- `ExcelFile.OptionsR1C1 = true`

OptionsR1C1 only affects how Excel shows the formulas, but not how you enter them in FlexCel, or how FlexCel will return the text of the formulas to you. That is the job of `ExcelFile.FormulaReferenceStyle`.

So for our original example, here is the code to do it with R1C1 notation. Note that due to the fact that R1C1 is relative, the formula is always exactly the same. There is no need to calculate a formula for each cell as we did in Solution 2; in fact **we can move the creation of the formula outside of the loop** to avoid creating temporary objects:

```
xls.FormulaReferenceStyle = TReferenceStyle.R1C1;
var Formula = new TFormula("= R[1]C * 2");
for (int col = 1; col <= 26; col++)
{
    xls.SetCellValue(1, col, Formula);
}
```

IMPORTANT

While we used **R1C1** internally to enter the formulas, in Excel they will show in A1 notation exactly the same as they do with the other 2 solutions. That is because as explained above R1C1 support is divided in a property that only affects FlexCel: [ExcelFile.FormulaReferenceStyle](#) and a property that only affects Excel: [ExcelFile.OptionsR1C1](#)

So you can work in R1C1 mode in FlexCel while keeping A1 mode in Excel, or vice-versa.

FlexCel comes with full R1C1 support built-in.

Bonus track - Checking the formulas in a spreadsheet

R1C1 references are not only nice to enter formulas, but also to check for consistency in existing files.

Imagine you have a file with formulas like in our example above, and you want to check that they are all as they are supposed to be. So for example in Column J, you have =J2 * 2 and not =A2 * 2. Checking this in A1 notation can be very complex, especially if the formulas are not simple.

But retrieve the formulas in R1C1 instead, and all you need to do to check for consistency is to **check that all formulas in A1:Z1 are the same!**

That is, retrieve the formula in A1 (in this case `"=R[1]C * 2"`) and then check that all other formulas in the range have the same as the text in A1. If a formula is different, then it is not consistent.

Changing the font name

Before Excel 2007, changing the font name in a cell was a matter of well... just changing the font name. To use for example `noto`, you would write:

```
var fmt = xls.GetFormat(xls.GetCellFormat(row, col));
fmt.Font.Name = "noto";

xls.SetCellFormat(row, col, xls.AddFormat(fmt));
```

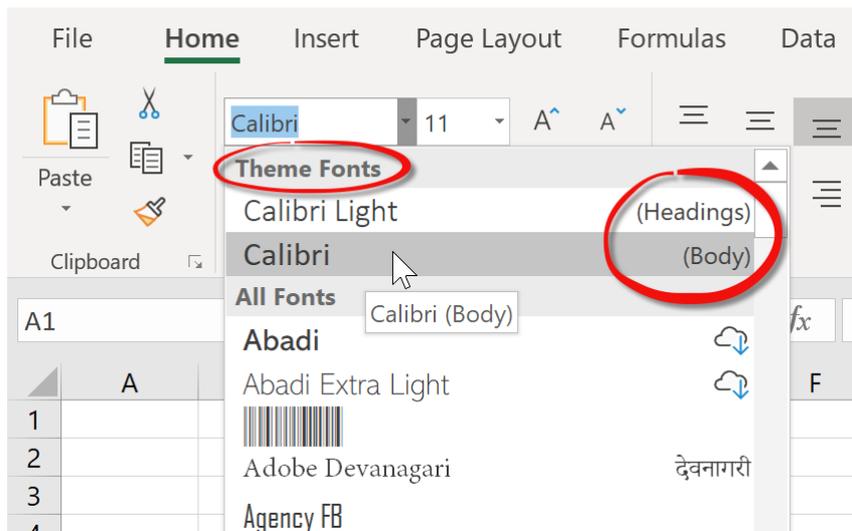
and that would be it. But if you try this code and open the generated file in a modern Excel, you will see that the font didn't actually change. What is going on?

Excel 2007 introduced themes, which are a collection of settings like fonts or colors which you can change to alter the look of the document all at once.

Themes mostly affect colors, and you can find a more in-depth explanation of them at the [Api Developer Guide](#)

But **themes also affect font names**. The theme defines 2 fonts: **Major** (used in the headings) and **Minor** (used in the normal cells)

When you select a font in the font combo-box in Excel, you might have noticed that the first 2 entries are under the category "Theme fonts" and that they have **(Headings)** and **(Body)** written at the right:



The "(Headings)" font corresponds to `TFontScheme.Major` and the "(Body)" font corresponds to `TFontScheme.Minor`.

NOTE

Those 2 fonts in Excel are **different** from the same fonts without the (Headings) and (Body) labels at the right.

Imagine that you set cell A1 to **Calibri (Body)** and cell A2 to just **Calibri**. And now you change the spreadsheet theme to a theme where the Body (or minor) font is Arial. When you change the theme, **A1 will change to Arial**, but **A2 will continue to be Calibri**.

So how does this affect FlexCel? Notice that in the code above, we didn't change the font scheme; only the font name. As the theme takes priority over the font name, a font linked to the **Minor** scheme will continue to be whatever font the theme defines for minor, and not what you specify in the font name.

To make that code work you need to also change the `TFlxFont.Scheme` to none.

This is the correct way to change the font name:

```
var fmt = xls.GetFormat(xls.GetCellFormat(row, col));
fmt.Font.Name = "noto";
// For the font name to change, we need to
// detach the font from the theme!
fmt.Font.Scheme = TFontScheme.None;

xls.SetCellFormat(row, col, xls.AddFormat(fmt));
```

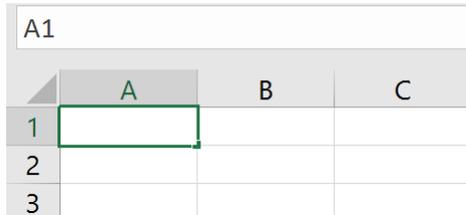
NOTE

To change **all** fonts of a given type to a different type in a file, take a look at [Replacing a font by another in an Excel file](#).

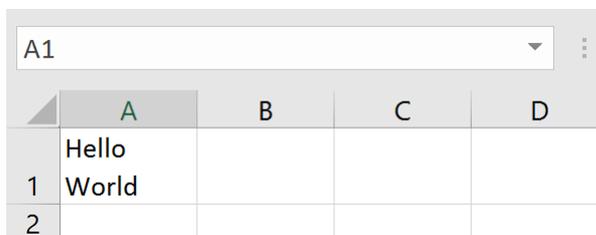
Entering multiple lines of text inside a cell

In Excel, you can enter multiple lines of text by pressing **Alt-Enter** between words.

You start with this:

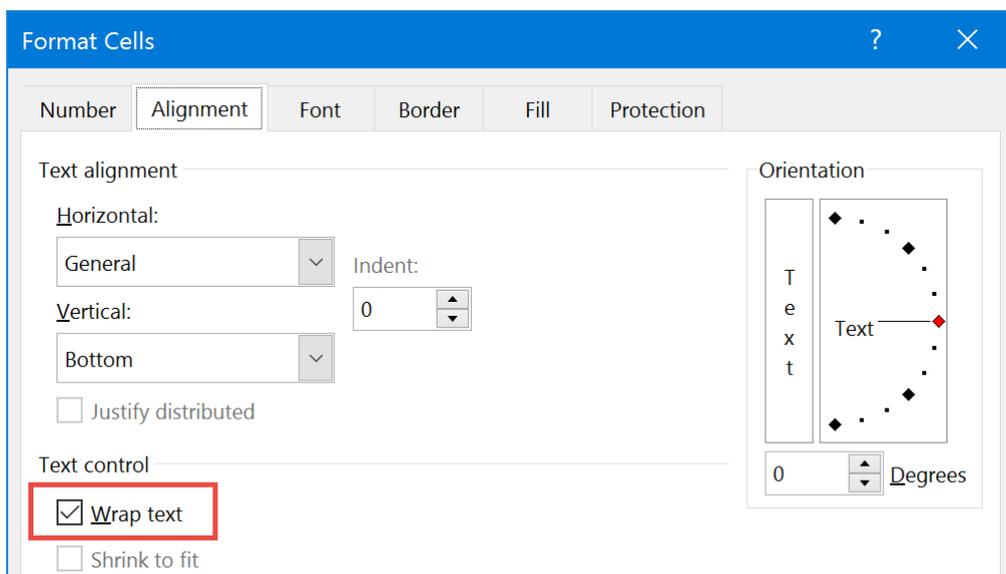


Then type **Hello**, press **Alt-Enter**, then type **World**, and you end up with this:

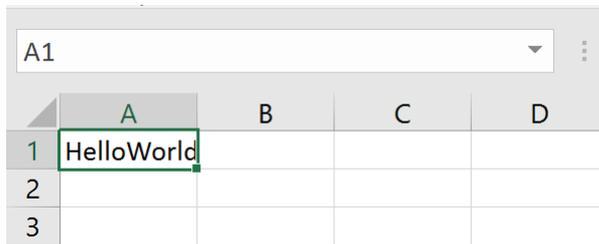


In the surface, you have just entered text into the cell. But digging a little below the surface, you have not only entered data, but also **changed the format of the cell!**

When you entered the text, Excel noticed that it had multiple lines, so it helpfully decided to change the format of the cell to **Wrap text**. If you look at the cell format now, you will see that it has wrap text enabled:



And it wasn't enabled before. If you manually disable it, you will end up with this:



Because the format changed without anybody telling you when you pressed Alt-Enter, you might not be aware of the fact that in order to display multiline cells you need to set `TFlxFormat.WrapText = true`.

So in order to enter a multiline text with FlexCel you need to:

1. Set `TFlxFormat.WrapText = true`.
2. Make sure you use a character 10 (`\n`) to separate the lines.

WARNING

It doesn't matter if your OS uses `\r\n` or any other character to separate lines: **Excel always uses the character 10 as line separator.**

So don't use `Environment.NewLine` to separate lines.

NOTE

You might wonder why FlexCel isn't as "smart" as Excel, and when it detects that you have carriage returns in your text, automatically change the format to wrap text.

The reason is simple: Excel and FlexCel have different target users. Excel is an interactive application, and in those cases every help the application can give you by guessing what you want to do is welcome. (well not every one, but most. Some guessings can be infuriating even in interactive mode.)

If Excel guesses wrong, you can see it immediately and just press ctrl-z to undo.

In FlexCel case, you are writing a program to create files, and every "smart" guess just makes the outcome more unpredictable. You tell FlexCel to do A, but it will end up doing B because it guessed B is what you really wanted to do.

So **FlexCel is dumb on purpose**. If you tell it to do A, it will do A and nothing else. If you set a cell value, it won't also change the cell format because it thinks you might want to do that too.

NOTE

Still here? Ok, what we told you in the last note was only partially true.

There is this method `ExcelFile.SetCellFromString` in FlexCel that is designed to work as Excel, and will change the format to wrap text if it detects carriage returns inside the string. But this is so because **SetCellFromString is specifically designed to replicate the Excel behavior** in case you need it. `SetCellFromString` is not the preferred way to enter text with FlexCel.

For most cases what was said in the last note stands: FlexCel won't do B when you tell it to do A. That is the only sane way to write a reliable program.

Understanding Excel measurement units

Excel uses many different units for measuring different things, and FlexCel uses the same.

For most rendering stuff, including the PDF API, FlexCel uses points, and those are simple: **A point is 1/72 of an inch.** As you can see in the [Google calculator here](#).

A point is also normally used to specify font sizes: For example a font with 11 points has a bounding box of 11/72 inches.

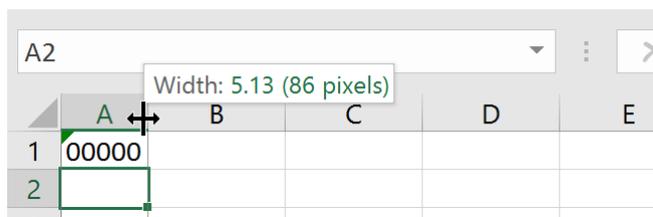
Sometimes we use pixels, where a pixel is not defined as a physical point on the screen, but just as 1/96 of an inch, no matter the actual resolution of the screen. **Excel and FlexCel are resolution-independent, so physical pixels are never used.** When we mention "pixels", we always refer to resolution-independent pixels.

Now, the Excel units are a little more complex. The simplest ones are the y coordinates (rows): **A row is measured in twips, or 1/20 of a point.** `ExcelFile.GetRowHeight` returns the current row height in twips.

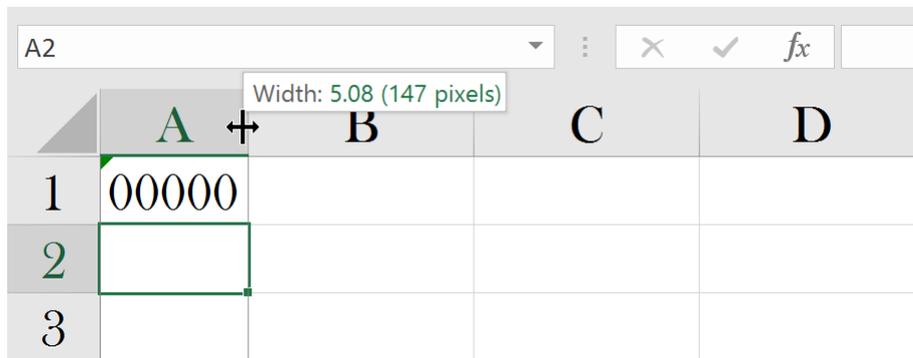
But the x coordinates (columns) are more complex. They measure how many "0"s you can put in a cell in the font used in the "Normal" style, plus some margins.

For example, the normal style for an Excel 2007 or newer file is "Calibri 11", so if you can fit 5 "0"s in a column with that font, ("00000") then the column width is 5. But not exactly, there are some margins too, which aren't correctly documented in Excel docs, and which actually change depending on the screen resolution. So the width is not going to be 5, but 5.something.

Here you can see how a column autofitted to "00000" looks like:



It is interesting to see what happens when we change the "normal" font of the spreadsheet to some bigger font. The column width in pixels will get bigger, but the internal units will still be around 5, because you can still fit 5 "0"s in that column. In fact, even when physically much bigger, the "size" got a little smaller when selecting a bigger font: from 5.13 to 5.08.



You can convert between columns and rows and pixels using [ExcelMetrics.ColMult](#) and [FlxConsts.RowMult](#). For example:

```
double RowHeightInPixels = xls.GetRowHeight(Row) / FlxConsts.RowMult;
```

```
double ColWidthInPixels = xls.GetColWidth(Col) / ExcelMetrics.ColMult(xls);
```

NOTE

The "pixels" in those examples are the "resolution-independent-pixels" we mentioned at the start. That is, 1/96 of an inch.

Now, sadly Excel isn't very exact about the units, so this isn't an exact conversion. You can see it simply by pressing a screen preview and measuring the real width of a cell, and compare it with what you see on the screen: you'll see it is not exactly the same.

The actual box width for a cell can also change if you use a different screen scaling: They will look different if you use say 100% or 175% screen scaling. FlexCel can offset this with the property [ExcelFile.ScreenScaling](#). While by default the screen scaling is 100 (which corresponds to 100% scaling or 96 dpi), if you are always working in a different setting, changing this property will make FlexCel work as if it was Excel showing the file at that particular screen scaling.

Changes between resolutions are not much, but they exist so in general it is a bad idea to try to do "pixel perfect" designs in Excel. You always need to leave some extra room because in different resolutions the cells might be slightly smaller or bigger.

For more info about how Excel units can change with the resolution, you can also read [Autofitting Rows and Columns in the API Developer Guide](#)

NOTE

As you can see in the images above, Excel will also show you the column size in "**Pixels**" besides the internal units.

But sadly those "pixels" aren't actual pixels either, or "resolution-independent-pixels" which we mentioned at the start which are 1/96 of an inch. If you took the effort to count them, you would realize that the pixels in that column are not what Excel says they are. Also, if you double the resolution of the screen those "pixels" will remain the same, even if the actual count of pixels doubled with the double resolution.

But the main issue is that those "pixels" are as unreliable as the other units Excel uses. They share the same resolution problems as the internal Excel units, so they don't really solve any problem.

Dumping a Dataset into Excel

One question we get asked from time to time is if FlexCel has a method like “**DumpDataset**” or “**CopyFromRecordset**” (which is the actual name in OLE Automation). This method should take a dataset and dump it into a sheet.

And the answer is no, we don’t include such a method because it would be too limiting: Normally you will have to customize how the cells are written, and you can’t do that with a single method that does the full export in one piece. So instead, we provide 2 other different solutions to the problem.

1. FlexCel has [reports](#), which you can use to dump a dataset “as is”, mostly as a DumpDataSet method would, but it also lets you do the formatting.

As an added advantage reports allow you to easily modify the resulting sheets without modifying the code at all, so even the final user can do it.

From reports, the most similar thing to "DumpDataset" is to create an empty spreadsheet, write "<#mydataset.*>" on it, save it, and then run the report against your dataset. You can find an example of this in the demos [Generic Reports](#) and [Generic Reports 2](#)

Of course those 2 demos focus in “generic” datasets which you don’t know beforehand. If you know what data you are outputting, then you can just write <#dataset.field> in the cells where you want that field written, and format each field as you want.

Reports are explicitly designed to deal with the problem of exporting datasets into Excel, but allowing formatting.

2. The reports in FlexCel work exclusively by calling the FlexCel API, so anything you can do in a report, you can do it directly with the API. So if for any reason you can’t or don’t want to use reports, you can use a method like this:

```
static void DumpDataSet(DataTable ds)
{
    var xls = new XlsFile(1, TExcelFileFormat.v2019, true);
    //Generate the formats we will be using to format dates and times.
    var Fmt = xls.GetDefaultFormat;
    Fmt.Format = "dd/mm/yyyy hh:mm";
    var DateTimeXF = xls.AddFormat(Fmt);

    //Now loop over all records and send them to the file.
    for (int row = 0; row < ds.Rows.Count; row++)
    {
        for (int col = 0; col < ds.Columns.Count; col++)
        {
            object value = ds.Rows[row][col];
            if (value is DateTime)
            {
                xls.SetCellValue(row + 1, col + 1, value, DateTimeXF);
            }
        }
    }
}
```

```
        else
        {
            xls.SetCellValue(row + 1, col + 1, value);
        }
    }
    xls.Save("test.xlsx");
}
}
```

Reading only the first row of a file

Sometimes you need to process a big number of files depending on their data in the first rows. If the data isn't what you want, you skip the file.

For those cases, it might make sense not to load the full file in memory, just to read the cell A1 and then close it. And FlexCel has the right tool for the job: **Virtual Mode**

Virtual mode is described in more detail in the [Performance Guide](#), but in this tip we will just give you the code to read a file and stop reading after the first cell.

You can also find this example in the [ExcelFile.VirtualMode](#) documentation.

```
static void ReadFirstRowOfAllSheets(string fileName)
{
    XlsFile xls = new XlsFile(); //Create the object but don't open the file
yet.
    xls.VirtualMode = true; //Set the mode to Virtual.
    xls.VirtualCellRead += (sender, e) =>
    {
        if (e.Cell.Row > 1)
        {
            if (e.Cell.Sheet < e.SheetNames.Length) e.NextSheet =
e.SheetNames[e.Cell.Sheet]; //Stop reading this sheet, move to the next.
            else e.NextSheet = null; //Stop reading the file.
        }
        else
        {
            Console.WriteLine("Cell: " + new
TCellAddress(e.SheetNames[e.Cell.Sheet - 1], e.Cell.Row, e.Cell.Col, false, false)
.CellRef);
            Console.Write("Value: ");
            Console.WriteLine(e.Cell.Value);
        }
    };
    xls.Open(fileName);
}
```

About

This documentation is for [FlexCel Studio for the .NET Framework v7.21.0.0](#)

Copyright (c) 2002 - 2024 tmssoftware.com

In this section:

What's new

Everything that changed in FlexCel since the last time!

Supported Excel functions

A list with all the built-in Excel functions that FlexCel can recalculate. Remember that for used defined functions, you can write your own implementation too.

Supported Excel charts

A list with all the Excel charts that FlexCel can render.

Copyright information

Information about the third party copyrights and licenses of code used by FlexCel.

Licensing information

[Single developer license](#)

A license for a single developer. Non transferable.

[Site license](#)

A license for all developers in a single company.

What's new in FlexCel Studio for .NET

New in v 7.21 - February 2024

- **Support for exporting Persian to PDF.** FlexCel supported arabic but not persian when exporting to PDF. Now both are supported.
- **Bug Fix.** Reviewed bidi algorithm because some arabic numbers could be written in the wrong order when exporting to PDF
- **Bug Fix.** Data labels in scatter chart were incorrect when the axis was reversed.
- **Bug Fix.** When rendering charts that had labels linked to "Value from Cells" FlexCel could fail to render the labels if they weren't manually specified.
- **Bug Fix.** There was an error when rendering PDF Khmer text in some corner cases.
- **Bug Fix.** Header images wouldn't be copied when calling the InsertAndCopySheets overload that takes an array of integers for the sheets to copy.
- **Bug Fix.** SetCellFromHtml could raise an exception with some surrogated unicode characters.
- **Bug Fix.** Charts with hidden series could raise an exception when drawing trendlines.

New in v 7.20 - December 2023

- **Support for .NET 8.** We now provide packages optimized for .NET 8
- **Updated SKIASharp.** SkiaSharp 2.88.3 had a serious vulnerability, we have updated FlexCel to require at least 2.88.6
- **Support for creating your own cultures when rendering Excel files.** Now you can override the dates returned by your app so they look exactly like Excel. For more information read the tip [localized month names](#)
- **Better support for subnormal numbers.** Excel doesn't support [subnormal floating point numbers](#) and instead converts them to 0. Now FlexCel does the same, instead of saving those numbers to the files.
- **FlexCel is more forgiving with too long data validation formulas.** FlexCel used to raise an error if a formula in a Data Validation is longer than 255 characters, because that is what the spec says and what Excel allows you to enter in its Data Validation Dialog. But, if you save a longer formula by manually editing the xlsx file, Excel won't complain, and the formula will still work. It won't work well, since you won't be able to edit the formula with the UI (and you might be corrupting Excel memory), but Excel won't complain. In order to let you edit those files, now FlexCel won't complain either when you open a file with a longer formula, and it will preserve the formula when you save the file back. But it still will throw an Exception if you try to manually enter that formula, so you know it is invalid. You can still enter the invalid formula manually by turning [Recovery Mode](#) on.

- **New setting `FlexCelConfig.LocalizedTEXTFunction` allows you to fine tune how FlexCel recalculates the "`=TEXT()`" function.** The setting [FlexCelConfig.LocalizedTEXTFunction](#) allows to change the way the `=TEXT()` function behaves in FlexCel. Look at the link above for more information.
- **Improved Label rendering when drawing charts.** Now we allow to show the fields "X Value" and "Y Value" in labels of scatter charts. Sometimes the marker if the label was set to show markers was drawn left-aligned, not centered.
- **Removed `canvas.getMatrix` calls in Android as they are deprecated.** FlexCel now keeps internal track of its Matrix, to avoid calling `canvas.getMatrix` which was deprecated in Android 16.
- **Bug Fix.** If the column names of a table had a format like "@_@" or similar, FlexCel would apply them and the column name would be wrong. Now the names are the same as in Excel. Some other improvements in handling of column names.
- **Bug Fix.** The XLOOKUP function would return "not found" when searching for a value bigger than any on the list and the match mode was "Exact match or next larger item", when there were empty entries in the list. This is the logical way to do it, it is how Google Sheets does it, and it is how Excel itself behaves in the other match modes. But for this match mode, Excel returns the first empty item in the list, not "not found". Now FlexCel mimics this behavior.
- **Bug Fix.** The SINGLE function could return wrong results in very special cases.
- **Bug Fix.** When autofitting rows FlexCel ignored empty columns formatted with big fonts.
- **Bug Fix.** When parsing some PNGs FlexCel could raise an arithmetic overflow.
- **Bug Fix.** FlexCel could fail to detect circular references in some corner cases, causing a Stack Overflow.
- **Bug Fix.** When drawing Error Bars in charts, if there was no line color assigned, FlexCel would not draw the lines instead of drawing them in Black.
- **Bug Fix.** Some files could show the error: "Value was either too large or too small for an Int32" when opened.

New in v 7.19.1 - September 2023

- **Bug Fix.** `InsertAndCopyRange` failed to copy cells in some corner cases

New in v 7.19 - September 2023

- **New property `FlexCelSVGExport.Encoding` allows you to change the encoding when generating SVG files.** Now you can change the Encoding of generated SVG files by using the new property [FlexCelSVGExport.Encoding](#)
- **Breaking Change: Now FlexCel will default the encoding when to exporting to CSV or SVG to UTF-8 without BOM.** FlexCel used to use UTF-8 with BOM to save CSV or SVG files if you didn't specify an encoding. Now the default encoding is UTF-8 with BOM, which

is more common. You can still specify an explicit Encoding as UTF-8 with BOM when saving those files if you want to, but if you are not specifying a default, the generated text files will change.

- **Now FlexCel will write the xml inside xlsx files as UTF-8 without BOM.** FlexCel used to write the xml inside xlsx files as UTF-8 with BOM, which is valid and works fine, but some third party tools could have issues reading those files. Excel writes the xml as UTF-8 without BOM itself, so this is not a breaking change. See <https://support.tmssoftware.com/t/save-utf-8-without-bom/21577>
- **Bug Fix.** In some rare cases when you entered #N/A as a parameter in a formula, the full formula would become #N/A. See <https://support.tmssoftware.com/t/handling-of-n-a-in-formuals-on-loading-excel-file/21425>
- **Bug Fix.** Page numbers in headers and footers were invalid when exporting to SVG.
- **Bug Fix.** FlexCel could write some invalid DXF records when saving table styles to xls files.
- **Bug Fix.** InsertAndCopyRange could fail to copy cells in some corner cases

New in v 7.18 - August 2023

- **Support for the new Excel2023 default format including the default aptos font.** Now when calling [ExcelFile.NewFile](#) there is a new option to create a file with the new office 2023 theme and fonts.
- **FlexCelPreview now is available for .NET 6+ Winform apps.** The FlexCel.WinForms nuget package now includes FlexCelPreview, and you can now use it from .NET 6 and newer Winform apps.
- **New property TExcelFile.PrintComments allows to directly manipulate how to print the comments.** The new property [ExcelFile.PrintComments](#) allows you to directly change how comments are printed without having to use [TExcelFile.PrintOptions](#).
- **FlexCel can now render comments "As displayed".** When [ExcelFile.PrintComments](#) is "As Displayed", FlexCel will now print and export to PDF/HTML/SVG the comments as they are displayed. See <https://support.tmssoftware.com/t/excel-with-comments-export-to-pdf-with-comments-showing/19377>
- **New property TExcelFile.PrintErrors allows to directly manipulate how to print the errors in formulas inside the sheet.** The new property [ExcelFile.PrintErrors](#) allows you to change how the errors in the sheet will be printed.
- **FlexCel can now render error in formulas according to the printer settings.** When [ExcelFile.PrintErrors](#) is not "As Displayed", FlexCel will now print and export to PDF/HTML/SVG the correct values.

- **New properties TExcelFile.PrintOptionsInitializedFromPrinter, TExcelFile.PrintOverThenDown and TExcelFile.PrintDraftQuality.** The new properties [ExcelFile.PrintOptionsInitializedFromPrinter](#), [ExcelFile.PrintOverThenDown](#) and [ExcelFile.PrintDraftQuality](#) allow to change the [TExcelFile.PrintOptions](#) in a simpler way. Now there are properties to change every one of the individual [PrintOptions](#), and so APIMate won't suggest changing [PrintOptions](#), but the standalone properties instead. APIMate will still show the code to change [PrintOptions](#), but commented out.
- **ApiMate won't show PrinterDriverSettings by default.** PrinterDriverSettings are a huge binary blob which can make the output of APIMate much harder to read. It was already commented out, but now it is not shown at all by default. There is a checkbox to show it if you need it.
- **Bug Fix.** When rendering charts inside xlsx files, sometimes FlexCel could fail to render the correct colors of some series, using black instead.
- **Bug Fix.** Formatted numbers inside cells with "Shrink to fit" didn't shrink when exporting to HTML
- **Bug Fix.** `<#if>` tag in reports would consider the condition true if it evaluated to NAN or a number.
- **Bug Fix.** `<#ref>` tag in reports now returns a real reference instead of a string with the cell reference. While for most uses it is the same, in some cases like in the "Cell" function, the old `<#ref>` tag wouldn't work.

New in v 7.17 - June 2023

- **Optimized support for .NET 7.** We include now packages compiled against .NET 7
- **SkiaSharp updated to 2.88.3.** The minimum SkiaSharp now required is 2.88.3.
- **Removed support for .NET Core 3.1 and 5.0.** As .NET Core 3.1 and 5.0 reached EOL, now the minimum .NET Core version supported is 6
- **Improved API for defining columns in tables.** Now you can define a totals formula or a column formula for the columns in the table, if needed. As usual, APIMate will show you how to do it.
- **Now FlexCel preserves digital signatures in macros.** When you have digitally signed macros in a file, now FlexCel will preserve them when opening and saving that file
- **New property DeleteEmptyBandsFixed in FlexCelReport controls what to do with empty fixed bands.** The new function [DeleteEmptyBandsFixed](#) lets you define what happens if a fixed band has zero records.
- **Bug Fix.** When recreating a table by calling AddTable and SetTable, the cell references could become invalid
- **Bug Fix.** The functions IFERROR, ISERROR and ISERR could sometimes return the error instead of the result of the function.

- **Bug Fix.** The functions COUNTIF, SUMIF and similar could behave wrong in some cases where you used wildcards. See <https://support.tmssoftware.com/t/countif-formula-with-wildcard-failing-after-recalc-method-is-called/19266>
- **Bug Fix.** The function TRIM in Excel removes double spaces in the middle of a text, while FlexCel's implementation would remove only spaces at the beginning at end. Also Excel's TRIM only removes spaces (character 32) and not other whitespace like tabs. FlexCel's implementation now does the same.
- **Bug Fix.** The function =NUMBERVALUE() could throw an Exception in some border cases

New in v 7.16 - September 2022

- **Support for using .NET 6 directly in iOS and Android, instead of Xamarin.** Now we support the new multi-target packages in .NET 6, which replace Xamarin.
- **Breaking Change: The default for the GraphicFramework property is now Native.** The property [GraphicFramework](#) now defaults to use the native framework if not manually specified. The native graphic frameworks normally work better than Skia, and don't require SkiaSharp.
- **Support for using native graphics engine in .NET 6 for Android, macOS and iOS.** In Nov 2020 [FlexCel 7.8](#) added support for switching graphics engines in Windows. Now when in .NET 6 you can also switch graphics engines in macOS, iOS (you can use CoreGraphics or Skia) and Android (you can use Native or Skia). To select between native or SkiaSharp in any platform, use the property [GraphicFramework](#)
- **SkiaSharp updated to 2.88.1.** The minimum SkiaSharp now required is 2.88.1. This was needed to support multi-targeting in .NET 6
- **Support for different numeric systems in cell formatting.** Now if you format a cell with a different numeric system like for example "\$-2000000]#,##0.00", FlexCel will render those numbers correctly. (see <https://ansarichat.wordpress.com/2018/02/20/how-to-type-arabic-numerals-in-excel/>)
- **Bug Fix.** When rendering charts that used =Offset to define the data, and some columns or rows were hidden, FlexCel could fail to hide the values when the chart was set to not plot hidden cells.
- **Bug Fix.** In some rare cases when there was merged cells whose first cell was hidden the background might not be exported to pdf.
- **Bug Fix.** If printing gridlines and there were hidden columns or rows, the gridlines could be printed over the real borders of a cell.
- **Bug Fix.** When exporting to CSV, there could be errors if you manually set cell values to NaN.
- **Bug Fix.** If exporting to PDF and the "normal" font of the spreadsheet was Calibri 9 columns could be wider than expected.
- **Bug Fix.** FlexCel could hang while loading some invalid third-party files.

New in v 7.15 - July 2022

- **Support for functions FILTER, SORT, SORTBY, and UNIQUE.** There is now support for the [functions introduced in office 2021](#) : FILTER, SORT, SORTBY, and UNIQUE.
- **Support for functions MAP, REDUCE, SCAN, MAKEARRAY, BYROW, and BYCOL.** There is now support for the [functions introduced in office 365](#) : MAP, REDUCE, SCAN, MAKEARRAY, BYROW, and BYCOL.
- **Support for functions TEXTBEFORE, TEXTAFTER, TEXTSPLIT, HSTACK, VSTACK, TOROW, TOCOL, WRAPROWS, WRAPCOLS, TAKE, DROP, CHOOSEROWS, CHOOSECOLS, and EXPAND.** There is now support for the new [functions introduced in office 365 beta](#) : TEXTBEFORE, TEXTAFTER, TEXTSPLIT, HSTACK, VSTACK, TOROW, TOCOL, WRAPROWS, WRAPCOLS, TAKE, DROP, CHOOSEROWS, CHOOSECOLS, and EXPAND.
- **New SHEET VISIBLE tag that allows to change the visibility of a sheet in a report.** You can now use the new [Sheet Visible](#) tag to hide or unhide sheets. See <https://support.tmssoftware.com/t/hide-worksheet-in-report-tags/18336/2>
- **You can now run reports on List<primitive> like List<double>.** Now you can use a list of a primitive type like List<string> in reports. You have to write <#list.Value> in the template for it to work. See the modified example of [using lists as datasets](#). See <https://support.tmssoftware.com/t/using-list-as-datasource/18513/4>
- **Ability to change the newline separator when exporting to CSV or Fixed-length text files.** By default FlexCel exports to CSV or fixed length using the newline separator from the OS. (that's CR/LF in Windows and LF in Unix/macOS). Now you can explicitly define the newline string when calling [Export](#) See <https://support.tmssoftware.com/t/settings-for-new-line-character-when-saving-texcelfile-to-csv/17886/3>
- **Support for Array formulas, UDFs, external names and Lambda names in Tokens.** [Tokens](#) now support Array formulas, user-defined functions (UDF), names that refer to external files, and Lambda names (like if you write in a formula "=MyName(4)" and MyName is a lambda function.). Standalone lambda functions (without using names) were already supported. The only thing not supported in tokens right now is what-if tables.
- **Function "INDEX" is now array-enabled.** Now you can pass arrays in the col and row parameters of the [INDEX](#) function and FlexCel will return an array of results.
- **Improved behavior when inside Docker containers.** Now when you try to run FlexCel in a Docker container without any fonts installed, FlexCel will tell you the problem and how to fix it. There is also a [new tip](#) explaining how to handle fonts inside docker containers.
- **FlexCelReport.AddConnection now supports adding IDbCommand directly.** There is a new overload of [AddConnection](#) that takes a IDbCommand instead of a IDbDataAdapter. This allows you to use [DirectSQL](#) in reports when the data framework doesn't has DataAdapters (as is currently the case with Microsoft.Data.SQLite)
- **Bug Fix.** The functions SWITCH and IFS could fail in some border cases.
- **Bug Fix.** Bubble charts could render bubbles of the wrong size if there were empty points in the chart data with bubble size different from 0.

- **Bug Fix.** Negative dates now show empty instead of "#####" when they are labels of charts. This is the behavior Excel has too.
- **Bug Fix.** Cells with diagonal borders, but borders style set to none could be rendered by FlexCel in some cases.
- **Bug Fix.** Some files containing khmer characters (or other complex scripts) could raise an exception when exporting to pdf.
- **Bug Fix.** Sometimes RenderObjects would not render the images. See <https://support.tmssoftware.com/t/images-disappear-sometimes/18567/6>
- **Bug Fix.** There could be errors in some specific cases when copying sheets from one file to another which had linked formulas.
- **Bug Fix.** The limit for custom formats in xls files was in 4000 when it really is 4050. We've updated FlexCel to allow 4050 custom formats when saving as xls.
- **Bug Fix.** The `TYPE` function didn't return 128 for lambda functions.
- **Bug Fix.** When third-party files had invalid modify/creation dates, FlexCel would refuse to open them. Now it will just ignore invalid dates and let those properties empty.

New in v 7.14 - January 2022

- **Improved floating point handling in .NET Core 3.0 or newer, .NET 5 or newer.** .Net 3.0 [completely rewrote the floating point parsing and formatting](#) and this lead to many small but noticeable changes in how FlexCel rendered files. For example, in .NET Core 3 the code `Console.WriteLine((-1.0/3.0).ToString("0"));` will write "-0" instead of "0" as before. Or `Console.WriteLine((4.5).ToString("F0"));` will write "4" instead of "5" as before. There are many other similar changes. We've done a big review of the code to try to ensure we behave like Excel, not like .NET Core 3 when rendering a file.
- **Removed support for .NET framework 2.0.** Now the minimum .NET version supported is 3.5
- **Support for bubble charts.** Now [FlexCel can render Bubble charts](#). You can also enter them with the API and APIMate will show you the code to do it.
- **New `<#Swap Series>` tag for reports.** The new `<#Swap Series>` tag allows you to create charts that grow or decrease their number of series according to the data available.
- **New `CustomizeChart` event for reports.** The new `CustomizeChart` event allows you to further customize the charts in the report once they have been generated.
- **Support for optional lambda parameters.** There is now full support for the new [optional lambda parameters in Excel](#).
- **IsOmitted function support.** There is now full support for the new [IsOmitted function](#).
- **Improved recovery mode.** `RecoveryMode` can now load more types of invalid files.
- **Support for localized versions of the CELL function.** Now you can write the first argument of function CELL in 24 languages, and FlexCel will understand them anyway. Before only English was understood. The languages added are Catalan, Croatian, Czech,

Danish, Dutch, Finnish, French, Galician, German, Hungarian, Italian, Kazakh, Korean, Norwegian, Polish, Portuguese-Brazil, Portuguese-Portugal, Russian, Slovak, Slovenian, Spanish, Swedish, Turkish and Ukrainian

- **Bug Fix.** Now FlexCel will throw an exception if you try to save a chart with more than 255 series. Before this release, FlexCel would just save the file, but a file with more than 255 series crashes Excel.
- **Bug Fix.** APIMate wouldn't report deleted chart titles, which could lead to chart titles appearing when there was a series with a name.
- **Bug Fix.** It was impossible to manually enter lambda formulas referring to names if [AllowEnteringUnknownFunctionsAndNames](#) was false.
- **Bug Fix.** A horizontal fixed band in a report would insert columns if using more than the fixed space, instead of just overwriting the cells.
- **Bug Fix.** Sometimes it was not possible to read properties from xls files.

New in v 7.13 - December 2021

- **Support for office 2021.** While we already supported it since we support Office 365, we added enumerations to create files with office 2021 defaults, and identify the file as being created by Excel 2021.
- **Support for Radar charts.** Now FlexCel can render Radar (Spider) charts. You can also enter them with the API and APIMate will show you the code to do it.
- **Support for .NET 6.** While we already supported .NET 6 beta, now that it is officially released, we tested and compiled FlexCel against the RTM release.
- **Breaking Change: Removed support for .NET Core 2.1.** .NET Core 2.1 is out of support and will not receive security updates in the future.
- **Support for the "Black and white" printing option in Excel.** Now when exporting to pdf, html, or printing, FlexCel will honor the "Black and White" option in the print options. Note that Black and White printing in Excel doesn't mean to print in grayscale, but printing all backgrounds white and all lines black no matter the actual colors. It also depends in what you render, for example colors in a chart bar will be replaced by patterns. There is also a new convenience method [PrintBlackAndWhite](#) that will let you modify the [PrintOptions](#) directly to print in black and white.
- **Ability to set the bottom row when specifying an autofilter.** Now [SetAutoFilter](#) can have an extra parameter to specify the bottom row of the Autofilter. Note that if there are more rows after the bottom row, those will be included anyway. Setting the bottom row is only useful to ensure that rows *up to* bottom rows are included, no matter if there are blank cells in the middle.
- **New convenience method [LoopOverUsedRange](#) that can be used to loop over a range of cells.** There is a new method [LoopOverUsedRange](#) which will let you loop over a range of cells in a simple and efficient way.

- **New properties `TextHorizontalOverflow` and `TextVerticalOverflow` in `TShapeProperties`.** The new properties [TextHorizontalOverflow](#) and [TextVerticalOverflow](#) allow you to set how the text overflows a shape. This corresponds to the checkbox in Excel "Allow text to overflow shape"
- **New convenience properties `TextRotated`, `TextVerticalAlignment`, `TextHorizontalAlignment` and `LockText` in `TShapeProperties`.** The new properties [LockText](#), [TextVerticalAlignment](#) and [TextHorizontalAlignment](#) allow to set or get [TextFlags](#) in a simpler way. The new property [TextRotated](#) allows to set [TextRotation](#) in a simpler way. APIMate now shows those properties instead of `TextFlags` and `TextRotation`. Read more in the tip about [xlsx and xls approaches to text rotation](#).
- **New method `SetObjectProperties` to set all the object properties in one step.** There is a new method [SetObjectProperties](#) that will allow to easily change the properties of an object. You just call [GetObjectProperties](#), modify the values you want and then call [SetObjectProperties](#).
- **New property `IsLocked` in `TShapeOptions`.** There is a new property [IsLocked](#) which controls how if the shape is locked or not.
- **New overloaded version of `SetObjectProperty` for booleans.** There is a new overload of [SetObjectProperty](#) that will allow you to set the property directly, without calculating the position in the set.
- **Improved legend drawing in charts.** Legends items in charts should render better now when some items are empty, and multiline items were improved too.
- **Bug Fix.** In iOS or Android you could get an error when saving files
- **Bug Fix.** Background of Axis text set by the API was ignored.
- **Bug Fix.** Rotation of Axis text set by the API was ignored.
- **Bug Fix.** When setting an Axis position in a chart to cross in the max value with the API, the value was ignored and it always used the manual crossing point.
- **Bug Fix.** Some hidden fills in files could be read as normal fills and so would appear if reading and saving a file.
- **Android demos now use AndroidX instead of Support Library.** The [Android Support Library](#) is deprecated and replaced by [AndroidX](#). So we've modified the demos using the old Support library to use AndroidX instead. Also the [documentation](#) has been updated to use AndroidX.

New in v 7.12.1 - October 2021

- **Bug Fix.** Workarounded bug in SkiaSharp: <https://github.com/mono/SkiaSharp/issues/1551> We also now require a minimum of SkiaSharp 2.80.2 instead of 2.80.3, since 2.80.2 doesn't has the bug.

New in v 7.12 - October 2021

- **PowerBI Data models are now preserved.** FlexCel will now preserve the spreadsheet data models used by Power BI technologies (PowerPivot, PowerMap, etc) inside xlsx files.
- **Bug Fix.** GetImageProperties could return an index out of bounds when passed a valid imageindex but there were grouped images.

New in v 7.11 - September 2021

- **Support for .NET 6 and Visual Studio 2022.** Official support for .NET 6 and Visual Studio 2022
- **New overloads of methods for getting image information that take objectIndexes instead of imageIndexes.** The methods [GetImageProperties](#), [GetImageName](#), [SetImageProperties](#), [DeleteImage](#) and [ClearImage](#) now have overloads taking an objectIndex/objectPath pair instead of an imageIndex.
- **New methods to convert between imageindexes and objectindexes with support for grouped shapes.** The new methods [ImageIndexToObjectPath](#) and [ObjectPathToImageIndex](#) can convert between imageIndexes and objectIndexes taking care of nested objects. The existing methods would only work in non-grouped objects.
- **You can now read and write the links of a camera object.** The methods [GetShapeLinkedCell](#) and [SetShapeLinkedCell](#) now work in pictures too, allowing to read or create camera objects besides to shapes with their text linked.
- **Improved loading of Excel 3, 4 and 95 files.** Now FlexCel can load camera objects in Excel 3, 4 and 95 files. It will also load the image names for images in Excel 95 files (Excel 4 and older don't store an image name)
- **SkiaSharp updated to 2.80.3.** The SkiaSharp library was updated from 2.80.2 to 2.80.3.

New in v 7.10 - August 2021

- **Support for SVG images embedded in xlsx files.** Excel has recently started allowing SVG images inside xlsx files. This releases adds full support for adding and reading SVG images to/from xlsx files. Note that we don't currently have a SVG renderer, so to add a SVG image you need to provide both an SVG and a PNG image. you can get more details in [this tip](#)
- **Breaking Change: Now when exporting to HTML and SVG, the SVG images stored inside the file will be embedded as SVG.** Before, FlexCel would always embed the PNG fallback image. To keep the old behavior, there are 2 new properties: [FlexCelHtmlExport.RasterizeSVGImages](#) and [FlexCelSvgExport.RasterizeSVGImages](#)
- **Ability to add chart sheets with the API.** There is a new method [AddChartSheet](#) which will allow you to add chart sheets with the API. As usual, APIMate will show you the code needed to add a chart sheet.

- **Ability to link shape text to cells via the API.** The new methods [GetShapeLinkedCell](#), [SetShapeLinkedCell](#) in [ExcelFile](#) and [GetShapeLinkedCell](#), [SetShapeLinkedCell](#) in [ExcelChart](#) allow you to read and write linked text in shapes.
- **Full Window management via API.** There is a new property [ActiveWindow](#) which allows you to select the [window](#) you are working on. You can then set the zoom, selected cells, etc. for that window, leaving the other windows unaffected. The new commands [AddWindow](#) and [DeleteWindow](#) allow you to add or delete windows. [WindowCount](#) will let you know how many windows you have in the file. [ActiveSheetForActiveWindow](#) will let you select an active sheet for each window, even if the ActiveSheet for FlexCel won't change.
- **Includes in reports can now be FIXED.** Now you can use the word FIXED in the "Shift type" parameter of the [include tag](#) . Fixed includes won't insert rows or columns, just overwriting the cells in the main report.
- **Support for recalculation of function NUMBERVALUE.** Now FlexCel can recalculate the [NUMBERVALUE](#) function introduced in Excel2013. As usual, the list of supported Excel functions is at [supported-excel-functions.html](#) in the docs
- **Ability to set shape effects like glow or shadow with the API.** Now you can set shape effects with the API, and APIMate will tell you the code to do it.
- **Improved recalculation speed.** We've implemented caches for some common formula patterns which should make your recalculations go much faster if your files use those patterns.
- **Improved support for Tiff and Gif images.** FlexCel used to convert tiff and gif images to png when loading them, so they could be saved inside xls files (xls files don't support those formats). Now FlexCel will preserve the file formats, and only convert them to pngs if you are saving in xls format.
- **Improved HTML5 exporting.** We've made the html5 files generated by FlexCel more compliant with html5 validators.
- **Improved drawing of shape shadows for xlsx files.** FlexCel will now render better the shadows in shapes inside xlsx files.
- **Improved drawing of log-chart gridlines.** Now the gridlines in logarithmic charts behave more similar to Excel in border cases
- **Comments added with the API won't include a shadow.** Now when you add a comment with the API, it won't include a shadow, same as modern Excel doesn't when you add a note. You can always use [SetCommentProperties](#) to add a shadow if you want to, and APIMate will show you the code.
- **Bug Fix.** Excel could crash with files including charts with Soft edges effect.
- **Improved handling of linked text in autoshapes.** Now FlexCel will preserve the properties of empty linked text in autoshapes. It will also handle better shapes with text linked to names that reference different sheets.
- **Bug Fix.** Sometimes FlexCel could fail to parse formulas with hard-coded arrays which had strings inside.

- **Bug Fix.** Conditional formats with iconsets where some values of the iconset were "No icon" could be saved wrong.
- **Bug Fix.** Comments could lose or gain a shadow when converting from xls to xlsx or xlsx-strict. Also colors in the comments could be wrong in border cases.
- **Bug Fix.** When renaming tables FlexCel wasn't renaming references in column formulas
- **Bug Fix.** FlexCel could crash when rendering chart labels with "Value from cells" if the range existed but was null.
- **Bug Fix.** Accessing some Conditional formats with inner borders could cause an Exception.
- **Bug Fix.** FlexCel would not export to pdf 3rd-party files which had unreadable file properties.
- **Bug Fix.** When using `<#database.#rowcount>` in expressions outside the sheet, you could get an exception.
- **Bug Fix.** FlexCel could throw an exception when inserting columns in xls files with invalid external references
- **Bug Fix.** FlexCel could throw an Exception when manually adding an autoshape to a chart that was created via the API.
- **Bug Fix.** Better compatibility with files generated by FastReports. Excel ignores border style 0 and fill styles 0 and 1, and now FlexCel ignores those too.
- **Better handling of third-party xls files.** Now FlexCel will convert the deprecated labels in biff8 xls files to sstlabels instead of keeping them as-is, allowing for much decreased memory usage when reading those files, and smaller result files.
- **Bug Fix.** Text to autoshapes added with the API would always be left-aligned.
- **Bug Fix.** FlexCel could report the BOM when reading custom XML parts inside xlsx files. Now the BOM is stripped out as it should.

New in v 7.9 - April 2021

- **Support for functions LAMBDA and LET.** We've reworked the recalculation engine to add support for [LAMBDA](#) and [LET](#) functions. With Lambda, the calculation engine is now turing-complete.
- **Support for functions SINGLE, VALUETOTEXT and VALUETOARRAY.** Added support for [SINGLE](#), [VALUETOTEXT](#) and [ARRAYTOTEXT](#) functions.
- **Breaking Change: Deprecated Android support in Visual Studio 2017.** As google Play is now requiring a target framework of Android 10, we can't ship Android support in VS 2017 since it supports up to Android 9. To deploy FlexCel to android, you will need to upgrade to Visual Studio 2019 or compile it manually with an older target framework.
- **Support for rendering multi-level labels in category axis.** In Excel you can set a category axis to have more than one row/column, and Excel will render those multi-level ranges in a different way than normal ranges. Now FlexCel will behave the same.

- **Support for legend keys in chart labels.** Now when exporting to PDF/HTML, if the option "Legend key" is enabled in the label options of a chart, FlexCel will render them.
- **Support for "Label contains Value from range" option in charts.** Now FlexCel will correctly handle the "Label Contains: " "Value from Cells" options for chart labels available in newer Excel versions. They will be exported to PDF/HTML and APIMate will show the code to create them in your programs.
- **Improved drawing of x-axis in charts.** Now FlexCel will automatically adjust the x-axis labels to 45 degrees if needed, and also take the space from near labels if those are empty.
- **TXIs3DRange now supports an external filename.** The object [TXIs3DRange](#) now has a property with the filename, in case that the range is from another file. This allows to use external files in user-defined functions.
- **Hyperlink Base support.** Now FlexCel will correctly preserve [Hyperlink Base](#) in xlsx files (it was already preserved in xls). Now the hyperlink base is also used when exporting to HTML, SVG or PDF.
- **Support for using an expression like `<#joinedtable.tablejoined.*>` to make a generic report in only one of the joined tables.** Now when you join tables in the template you can use `<#joinedtable.tablejoined.*>` or `<#joinedtable.tablejoined.**>` to create a generic report only in the fields of that subtable.
- **The `<#ref>` tag can now use tags in its parameters.** Now you can write something like `<#ref(<#dataset.#rowcount>,3)>`. Before this version tags were not allowed as parameters.
- **New property "IsCameraObject" in TImageProperties.** The new property [IsCameraObject](#) lets you know if an image is a camera object or not.
- **Improved compatibility with xlsx files created by SoftMarker Office.** SoftMaker office adds many extensibility points in places of the xlsx where they are not allowed. FlexCel complained about that, but in the new version we ignore the ones we could identify.
- **Axis labels will now render with a background color if they have one.** Now the axis labels will render the background color if you assign a color to them.
- **Bug Fix.** FlexCel would always render labels in the category axis as not "linked to source" even if they were.
- **Bug Fix.** Labels which come from cells that are formatted to show negative values in different colors show with that color in Excel, except in pie charts. FlexCel used to ignore that color, now it will display it.
- **Bug Fix.** When rendering xlsx charts, labels which were manually positioned would ignore the default numeric formatting.
- **Bug Fix.** Structured references with text formatting could be saved wrong to new xlsx files.
- **Bug Fix.** Rotated labels in charts could be a little below or above from where they should go.
- **Bug Fix.** FlexCel could fail to parse a formula where the sheet name started with some Unicode characters, like for example "※MySheet"
- **Bug Fix.** When reading structured references in Virtual Mode, the structure reference text would be wrong since it wasn't calculated until after the sheet was loaded.

- **Bug Fix.** FlexCel didn't preserve or render text linked to cells in shapes inside charts.
- **Bug Fix.** FlexCel would allow you to name a sheet starting with a single quote ('), and that would cause an invalid file. Now the quote at the start of the name will be replaced by a "_" as other invalid characters do.

New in v 7.8 - November 2020

- **.NET 5 Support.** The codebase has been updated to support .NET 5. **Important** : To use FlexCel in .NET 5, please update to at least this FlexCel version. .NET 5 changed string handing in Windows to use ICU, and that causes problems with older versions of FlexCel in Windows. See <https://github.com/dotnet/runtime/issues/43736>
- **Breaking Change: Removed support for .NET Core 2.0 and 3.0.** We removed support for .NET Core 2.0 and 3.0 as both reached end of life. We keep supporting .NET Core 2.1 and 3.1. See <https://dotnet.microsoft.com/platform/support/policy/dotnet-core>
- **Support for switching Graphics engines in .NET Core or .NET 5.** There is a new property in FlexCelConfig: [GraphicFramework](#) which allows you to select between using GDI+ in Windows (native) or SKIA (better compatibility with other platforms which use SKIA)
- **Support for reading fonts from the disk even if the graphics library returns that information.** There is a new property in FlexCelConfig: [ForcePdfFontsFromDisk](#) which allows you to select if FlexCel should use the font data returned by the graphics library if possible, or always scan a folder with fonts.
- **Improved performance in the SKIA graphics backend for .NET 5.** Now SKIA is faster than GDI+ in windows if you use it as the graphics library.
- **Improved compatibility with invalid xls files.** Now FlexCel can ignore invalid ministreams when reading corrupt/invalid xls files.

New in v 7.7 - October 2020

- **SkiaSharp used by .NET Core updated to 2.80.2.** We've updated the .NET Core code so it uses SkiaSharp 2.80.2
- **Improved handling of unknown parts inside xlsx files and improved Google sheets compatibility.** Now FlexCel has a more solid implementation for preserving unknown parts and relationships inside xlsx files. This fixes issues that could happen when editing xlsx files generated by google sheets.
- **Bug Fix.** When copying a sheet to a different file, ImageCount in the target file would return 0 even if there were images.
- **Bug Fix.** FlexCel wouldn't allow some unicode characters in 3d formulas, while Excel would allow them.
- **Bug Fix.** Improved support for camera objects. A new property [MaxNestedCameraObjects](#) allows to specify how many times a camera object can recursively draw itself.
- **Bug Fix.** FlexCel could not read some files with deleted what-if tables.

New in v 7.6.4 - July 2020

- **New syntax for ALIAS DataSets.** The ALIAS syntax introduced in FlexCel 7.1.1 could only be used in simple bands. The new "..ALIAS.." syntax doesn't have those limitations and can be used in all cases. Note that the old ALIAS syntax is still supported and you can use it for simple cases, but we recommend the new syntax for new development. Take a look at [Alias bands in the report designer's guide](#)
- **New "COUNT" parameter in <#Aggregate> tag.** The [Aggregate Tag](#) now accepts "Count" as aggregating method.
- **Bug Fix.** Now nested <#aggregate> and <#list> tags will be put in master-detail if they are related.
- **Bug Fix.** When inserting cell ranges with multiple rows used as data by pivot tables, the pivot table might not adapt correctly.
- **Bug Fix.** Sometimes when copying only format from cells, and the cells had only a column or row format, the format wouldn't be copied.
- **Bug Fix.** When exporting to HTML both headers and footers would be ignored if you specified THidePrintObjects.Header, and THidePrintObjects.Footer was ignored. Now footers will work with THidePrintObjects.Footer and THidePrintObjects.Header will only hide the headers.
- **Bug Fix.** Fixed issues with <#evaluate> tag when it evaluates recursively
- **Bug Fix.** Some invalid third party xls files could fail to load.

New in v 7.6.2 - June 2020

- **SkiaSharp used by .NET Core updated to 1.68.3.** We've updated the .NET Core code so it uses SkiaSharp 1.68.3
- **Bug Fix.** When adding a chart to a file via the API and immediately rendering it to PDF without saving it, the chart might not be rendered in the PDF file.
- **Bug Fix.** Previously the last row in "X" Bands in reports was deleted before the detail bands were inserted. This could cause unwanted behavior if the details shared the same rows as the master. Now last rows in X Bands will be removed after the details are inserted.
- **Bug Fix.** A fixed band inside a master-detail bidirectional report would behave as non fixed.

New in v 7.6.1 - May 2020

- **Improved conversion from strings to numbers.** Now FlexCel will behave more like Excel when converting strings to numbers, and convert for example the string "(1)" to the number -1.
- **Bug Fix.** Deeply nested array formulas could return N/A results in some corner cases

New in v 7.6 - May 2020

- **Support for rendering logarithmic charts.** Now FlexCel will render logarithmic charts to PDF or HTML.
- **When rendering pages to PDF and PNG, if an image or chart goes over the columns or rows in a page, now it won't overflow.** In previous FlexCel versions, if an image spilled over to the next page, it would also go over the last cell in the current page.
- **SkiaSharp used by .NET Core updated to 1.68.2.1.** We've updated the .NET Core code so it uses SkiaSharp 1.68.2.1
- **Now FlexCel will search in c:\Windows\Fonts and %localappdata%\Microsoft\Windows\Fonts for fonts when exporting to PDF.** Windows 10 version 1809 introduced the concept of "local fonts", that is fonts that are installed for the current user only. (see <https://blogs.windows.com/windowsexperience/2018/06/27/announcing-windows-10-insider-preview-build-17704/>) So now FlexCel will search in the traditional Windows fonts folder and the current-user-font-folder by default. You can always change the behavior with the [GetFontFolder](#) event. See [the new section about fonts in Windows inside the PDF exporting guide](#)
- **Now FlexCel won't throw an Exception if a Font folder in the PDF FontFolder path doesn't exist.** Now when you specify multiple paths in the [GetFontFolder](#) event, FlexCel won't show an error unless none of those paths exist. In previous version, if you returned for example "c:\mypath1;c:\mypath2" and mypath2 didn't exist, FlexCel would shown an error. Now it will only show an error if both mypath1 and mypath2 don't exist. You can change this behavior with the new property [OnFontFolderNotFound](#)
- **Improved handling of chart gaps when there are null values.** In Excel 2003, an area chart would never have a gap: Null values would be considered 0. After Excel 2007, area charts can have gaps. FlexCel behaved like Excel 2003, never showing gaps for area charts. Now FlexCel will behave like Excel 2007 when reading newer xls/x files, while still behaving like 2003 when reading old xls files.
- **Bug Fix.** Sometimes when calling RenderObjects the border of a chart would not be exported to PDF or PNG.
- **Bug Fix.** Leader lines in stacked bar charts were wrong when the axis was reversed
- **Bug Fix.** Manually positioned labels in stacked bar charts were a little offset from their manual position.
- **Bug Fix.** Now FlexCel will draw a maximum of 10000 ticks per axis in charts, to avoid taking too long drawing too many ticks that aren't visible anyway.
- **Bug Fix.** Xlsm files containing macros and with sheet names starting with a number and bigger than 24 characters, could generate invalid files when saved in FlexCel.
- **Bug Fix.** Now FlexCel will validate when manually setting a sheet codename, that the name is ASCII and starts with a letter.
- **Bug Fix.** Bidirectional reports could fail to delete rows or columns in complex reports.

- **Bug Fix.** FlexCel would consider a protected range title containing a "?" invalid. This would prevent it from loading files that used "?" in protected ranges.

New in v 7.5 - April 2020

- **Support for adding charts to a sheet with the API (xlsx files only).** A new method [AddChart](#) will allow you to add a chart to a sheet in xlsx files, which you can then customize with other methods like [AddSeries](#). There is a new demo [Chart API](#), and as usual, **APIMate** will show how to add a chart similar to one in Excel. Note that this method will work only in xlsx files, not xls.
- **New methods [SetTitle](#), [SetOptions](#), [SetChartLegend](#), [SubchartCount](#), [GetSeriesInSubchart](#), [SetSeriesInSubchart](#) and [AddSubchart](#) in [ExcelChart](#).** The new method [SetTitle](#) will allow you to set the title of a chart. [SetOptions](#) will allow you to customize the properties of the chart. [SetChartLegend](#) customizes the legend or adds a new one. [SubchartCount](#) will tell you how many subcharts there are in the main chart. [GetSeriesInSubchart](#) and [SetSeriesInSubchart](#) allow you to read or set one series of one subchart. [AddSubchart](#) adds a new subchart to the chart. Note that the methods are only for xlsx files.
- **New set method in the properties [PlotArea](#), [Background](#) in [ExcelChart](#).** [PlotArea](#) can now change the properties of the plot area like the position or fill color. [Background](#) can now change the background of the chart. Note that all the methods here will only work in xlsx files.
- **Now [SetSeries](#) and [AddSeries](#), [DeleteSeries](#) work also in xlsx charts.** [SetSeries](#), [AddSeries](#) and [DeleteSeries](#) now work in xlsx charts the same as they work in xls charts.
- **Support for calculating the upcoming [XLookup](#), [XMatch](#), [RandArray](#) and [Sequence](#) functions.** FlexCel can now calculate the functions [XLookup](#), [XMatch](#), [RandArray](#) and [Sequence](#) which are coming to Excel in July 2020.
- **Better chart rendering when there are date axis.** Now in some cases of date axis, FlexCel should render them better.
- **Support for .NET Core 3.1.** Now there is a specific dll for .NET Core 3.1.
- **Breaking Change: Removed support for .NET Core 1.0 and .NET Standard 1.5.** We've deprecated support for .NET Core 1.0 and .NET Standard 1.5 as both reached end of life.
- **Support for importing bullet lists when importing html.** Now when calling [SetCellFromHtml](#), doing reports from html strings, or in general when importing html into a cell, FlexCel will import ordered and unordered bullet lists (and).
- **Preserving and adapting single cell mappings in XML Maps in xlsx files.** Now FlexCel will preserve and modify the references to single cells in an XML map inside an xlsx file.
- **Improved compatibility with invalid third party files.** Now FlexCel will ignore some parts of the xlsx file that should exist but might not when the xlsx files are manually edited. This will allow you to open those files anyway if there are no more errors besides that one.
- **Bug Fix.** FlexCel wasn't calculating conditional formats if the formulas defining the conditions were array formulas.

- **Bug Fix.** FlexCel could fail to process some files where a shape had an ending coordinate smaller than the starting coordinate.
- **Bug Fix.** When rendering charts you could get an index out of bounds in some corner cases.
- **FlexCel will ignore invalid themes when reading xls files.** Now when an xls file has an invalid theme, FlexCel will ignore it and just use the default theme instead of throwing an exception. This is the way Excel behaves.

New in v 7.1.1 - January 2020

- **New __ALIAS postfix in named ranges allows multiple ranges to the same database.** Now if you need to define 2 names to the same dataset in the same sheet, you can name them like "__dataset1__ALIAS_SomeUniqueId" and "__dataset1__ALIAS_OtherUniqueId". Take a look at [Alias bands in the report designer's guide](#)
- **Bug Fix.** Effects for an image in the background of an image could be incorrectly saved. In some cases, this could cause Excel to crash.
- **Bug Fix.** When using a not standard row height for the whole sheet, FlexCel would create new rows to have automatic height, instead of having the standard row height for the sheet. This could confuse Excel.
- **Bug Fix.** FlexCel will now check that font sizes saved are between 1 and 409 points. Before it would let you save any font size, and invalid font sizes would crash Excel when opening the file.
- **Bug Fix.** The report designer failed to start.
- **Bug Fix.** In some corner cases charts saved by FlexCel inside xlsx files would fail to load.

New in v 7.1 - December 2019

- **Support for .NET Core 3.** While the dll for .NET core 3 preview still works in the final .NET core, we have now tested and released an official .NET core 3 build.
- **Improved chart rendering.** Many small tweaks. We now support different line cap and join styles. The chart axis now goes above bar charts but below line charts. There is support for arrows at the end of lines in charts. The legend items draw a little larger to be more like Excel. And many more details.
- **Reports can now use nested properties in Aggregates, Filters, Sort and Master-details relationships.** Now in the config sheet you can filter or sort by a nested property, like for example sorting in the value of field1.field2.field3. Also you can use nested properties in relationships and in aggregates like `<#aggregate(max;table.field1.field2.field3)>`
- **Now for reports you can set semi-absolute references in the config sheet.** Besides the old way to set semi-absolute references with [SemiAbsoluteReferences](#), now you can change the setting directly in the config sheet. There is also a new [tip on what semi-absolute references are](#).

- **Now you can use report expressions that call themselves recursively, as long as the recursion converges.** Now you can have a `<#tag>` that depends on other `<#tag2>` which at the end depends on `<#tag1>` again, as long as it is not an infinite recursion. FlexCel now limits the number of recursions via the new property [ExpressionRecursionLimit](#)
- **Improved recalculation speed and decreased memory usage.** We've fine tuned the calculation engine so it is faster and uses a less memory. We've also added 2 new methods: [StartBatchRecalcCells](#) and [EndBatchRecalcCells](#). When you are doing multiple calls to [RecalcCell](#) you can speed up the recalculation by writing the calls between Start/EndBatchRecalcCell calls.
- **Breaking Change: Removed overload method `ExcelFile.GetImage(Integer, string, TXIsImgType, TStream)`.** The method `ExcelFile.GetImage(Integer, string, TXIsImgType, TStream)` was confusing, because the `ObjectPath` (the second parameter), was always ignored. To use the `objectPath`, you needed to use [GetImage\(Int32, String, TXIsImgType, Stream, Boolean\)](#) and set the last parameter to true. If you were passing an empty object path, then you could just call [GetImage\(Int32, TXIsImgType, Stream\)](#)
- **The HTML engine can now parse `` tags.** When setting a cell or a `RichString` from an HTML string, the old FlexCel could parse HTML like `font color="red"` or `font size=3` but not `font style="color:red;size:16px"` Now both ways are supported.
- **Support for calculating BAHTTEXT function.** Now FlexCel can calculate BAHTTEXT.
- **Improved compatibility with invalid xlsx files.** Now FlexCel will ignore some missing parts in a corrupt xlsx file the same way Excel ignores them.
- **Bug Fix.** FlexCel might not preserve comment backgrounds in xlsx files if the background was an image or texture.
- **Bug Fix.** FlexCel would not correctly read or write left and right cell borders in strict xlsx files.
- **Bug Fix.** FlexCel was failing to render images which had an image filled background.
- **Bug Fix.** FlexCel could fail to open some files which didn't completely implement the xlsx spec but which Excel could open.

New in v 7.0 - August 2019

- **Breaking Change: Support for rendering charts inside xlsx files.** We have fully rewritten the charting engine so it can now render charts inside xlsx files too. As charts inside xlsx files are completely different from charts inside xls files, this support meant that we had to modify some of the APIs to retrieve the chart information. If you are manually retrieving chart information like the color of the plot area or the title of a chart, some methods might have been moved or changed, so this is why this is a breaking change. But for most uses, nothing should break.
- **Support for Visual Studio 2019, .NET Framework 4.8 and .NET Core 3 preview.** Now FlexCel supports .NET core 3 and .NET Framework 4.8. And the setup will offer integration with visual studio 2019.

- **Removed FlexCel-Portable for windows 8.1.** As Windows 8.1 reached end-of-life support in January 8, 2018, and the reason we kept FlexCel-Portable81 was to target Windows 8.1, we decided to remove it. Due to lack of sync APIs in Win8.1, FlexCel-Portable was a completely different API from all the other FlexCel versions, which made it more difficult to share code, both for us and for you. For UWP apps you should now use FlexCel-UWP or FlexCel for .NET Core.
- **UWP Support updated to version 6.** We updated the dependency in FlexCel-UWP to version 6, and now the minimum required Windows version is Windows 10 "Fall Creators Update"
- **Breaking Change: Improved compatibility with "Autosave" in Excel 2019.** When [OptionsCheckCompatibility](#) is set in a file, Excel disables the Autosave function. To avoid accidentally writing this setting and thus disabling the Autosave function, now FlexCel will ignore this option by default and not write it to xlsx files. If you really want to save this setting to the file, you now also have to set [ForceUseOptionsCheckCompatibility](#) to true.
- **FlexCel will now generate "faux" bold and italics when exporting to PDF.** When exporting to pdf and the used font doesn't have italics or bold variants, FlexCel tries to simulate the styles by making the pen wider (for bold) or slanting the characters (for italics). The method used in older FlexCel versions only worked when not embedding the fonts, but today most fonts are (and should be) embedded. This new FlexCel version creates "faux" italics or bold variants even when the fonts are embedded. To turn this feature off, you can use the property [UseFauxStyles](#) property in [FlexCelPdfExport](#) .
- **Ability to specify different fallback fonts for italic, bold or bold-italic variants when exporting to PDF.** In addition to the existing [FallBackFonts](#) property in [FlexCelPdfExport](#) we have now added 3 new properties: [FallBackFontsBold](#), [FallBackFontsItalic](#) and [FallBackFontsBoldItalic](#). If set to empty (the default) then FlexCel will keep looking for fonts in the usual way with [FallBackFonts](#). But if you have fonts that have only regular, bold, italic or bold-italic variants, now you can specify different fall back fonts for each. So for example, you could use "Font1Bold" as a bold fallback, and "Font3Regular" as the regular fallback.
- **GetHtmlFromCell now can add the cell formatting to the resulting string.** The old version of [GetHtmlFromCell](#) would return only the format of the rich string inside the cell, but not include the format of the cell itself. So if a cell was formatted as bold, but inside there was a plain "text" string, [GetHtmlFromCell](#) would return "text", and expect the text was hosted inside a table cell with style bold (<td style="bold">text</td>). Now there is a new parameter "includeCellFormatting" that when true, FlexCel will return "text" so you can use it outside formatted td tags.
- **Breaking Change: Support for semantic theme colors.** We added 4 new values for [TThemeColor](#) : Dark1, Light1, Dark2 and Light2. In the xlsx specification, there are 12 theme colors, which include "Dark1/2" and "Light1/2" variants. In Excel cells, the text color is "Dark" and the background color is "Light". But in drawings or charts, you can use 4 semantic colors: "Text1", "Text2", "Background1" and "Background2". While those colors are mapped Text->Dark and Background->Light, it is possible to manually edit an xlsx file and change the mapping. So now FlexCel differences between [TThemeColor](#) (which now includes semantic colors) and [TPrimaryThemeColor](#) which includes only the 12 real theme colors and no semantic colors. Most of your code should stay the same, but if you are changing themes in code, you might need to replace some instances of [TThemeColor](#) by

TPrimaryThemeColor. This is a compile-time breaking change: If your code compiles then nothing is broken. If your code doesn't compile anymore, you need to change TThemeColor by TPrimaryThemeColor where the compiler complains, and the code will keep working like before.

- **Breaking Change: Most properties in TDrawingRichString are now nullable.** Properties like bold or italics in a [TDrawingRichString](#) now can have a null value, which means that the value of the parent should be used. This change shouldn't affect most uses of TDrawingRichString, but if you were manually parsing DrawingRichStrings, you might now need to check if the nullable properties have a value before using them.
- **The <#evaluate> tag in reports can now evaluate a string multiple times.** There is a new optional parameter in the [evaluate tag](#) in reports that allows you to evaluate a string multiple times. This is useful if you store tags in the database itself. For example, if you have a field in the database named "Expression" with value "<#other tag>", then `<#evaluate(<#db.expression>;2)>` will evaluate first the value of expression, find out it is <#other tag>, then evaluate again <#other tag> and write the value of other tag in the cell.
- **Support for creating or reading xlsx files with uncompressed size bigger than 4 Gb.** FlexCel now uses the Zip64 file format automatically when creating xlsx files which won't fit in a standard zip container. It also can now read xlsx files saved with Zip64 file format.
- **New overloads for DeleteSheet allow to delete a sheet by its name or index.** There are now 2 new variants of [DeleteSheet](#). The first variant will let you delete a sheet given its name -- `DeleteSheet("sheet1")` --, and the other will let you delete n sheets since some index -- `DeleteSheet(SheetIndex, NumberOfSheetsToDelete)`
- **Improved bidirectional text handling.** We updated the Unicode bidi algorithm to the latest version, and added support for glyph mirroring and bracket matching algorithm. This should render right-to-left text in a way that is more compliant with the Unicode standard.
- **Improved drawing of autoshapes.** Excel 2007 changed the way in which it draws autoshapes in many small but visible ways. We made some tweaks in the autoshape algorithms and color handling routines to make autoshapes render even more like Excel 2007 and not Excel 2003.
- **Improved drawing of gradients and conversions from xls to xlsx.** We did a big rewrite of the gradient-handling code to better support the newer gradient styles in xlsx.
- **Improved drawing of patterns and conversions from xls to xlsx.** All pattern rendering code was reviewed and updated to better match Excel. Now every pattern style is exported to pdf/html/printed/etc.
- **Improved rendering in iOS, macOS, Android and .NET Core.** The rendering engines for CoreGraphics (used in iOS and macOS), SKIA (used in .NET Core) and Android have been updated to generate more accurate rendering of Excel files.
- **Improved support for Excel themes.** We've improved the theme engine to better handle files with wrong data in the themes. Now FlexCel can fix those files when you save them.
- **Improved compatibility when saving strict xlsx files.** While the strict xlsx files FlexCel created before were valid, now we create files that are more similar to what Excel creates when you save as strict xlsx.

- **Improved compatibility with invalid files created by third-party tools.** Now FlexCel can read files which have invalid cell references in the cell table, ignoring the reference completely as Excel does.
- **Improved conversion between strict and transitional xlsx files.** Now when preserving full parts of the xml of a strict or transitional file, FlexCel will convert the namespaces accordingly if you save as transitional or strict respectively.
- **Function Cell("filename") now returns the filename.** Now FlexCel can recalculate cell with Cell("filename"), which can be used to know the sheet where a cell is. Note that for security reasons, FlexCel won't return the folder there the file is, only the filename.
- **Breaking word in hyphens.** Now FlexCel will break words in hyphens when it has to fit multiple lines of text, same as Excel does.
- **Breaking Change: The Links property of TSheetSelector now is a readonly list of TSheetSelectorLink.** Now [Links](#) in TSheetSelector contains not only the name of the html sheet but also the name of the corresponding excel sheet. If you were using this property, you will need to use Links[index].HtmlSheetName instead of Links[index]. A new method [GetHtmlSheetNameFromExcel](#) allows you to easily find the associated html sheet name from an Excel sheet name.
- **Bug Fix.** Some non visual characters like "right to left mark" were exported to pdf, even when they are invisible. Now they don't show in the generated pdfs.
- **Bug Fix.** FlexCel won't let you save files where tables have 0 rows of data, since that would become an invalid xlsx file. Now it will raise an exception if trying to save such file.
- **Bug Fix.** FlexCel won't let you enter empty array members anymore, like in the formula ={1,,2} which would create an xlsx file which could crash Excel. It will also automatically remove spaces before and after the element, so the formula ={ 1, 2, 3 } will be entered as {1,2,3}. Before this version, the formula ={ 1, 2, 3 } would be considered invalid.
- **Bug Fix.** When autofitting columns with line feeds (character 10) inside, FlexCel might fail to recognize them and try to fit everything in one line.
- **Bug Fix.** Arrows in lines were not scaling when printed or exported at a zoom different from 100%.
- **Bug Fix.** In html exporting, a cell which expanded over adjacent cells could cause the output to shift if there were hidden columns in the middle.
- **Bug Fix.** When exporting a file as HTML with tabs for sheets and there were hidden sheets between sheets, the links in cells to a different tab could be incorrect.
- **Bug Fix.** FlexCel for .NET core would fail to create PDF/A files.
- **Bug Fix.** Some hyperlinks in xls files could return an empty string when read, even if they had data.
- **Bug Fix.** Lines with 0 width were not showing in SVG files.
- **Bug Fix.** Sometimes when copying cells between files the indexed colors could be converted to similar but not equal RGB colors.

- **Bug Fix.** When using the SKIA/Android graphic stack some lines in the charts could appear not connected.

New in v 6.26 - March 2019

- **In Reports now you can reference tables which include dots by writing `<#[db.something].field>`.** Now you can use square brackets in both the table name or field name to reference tables or fields which include dots. This is useful especially for reports from classes as shown in the new [Advanced LINQ example](#).
- **SkiaSharp used by .NET Core updated to v1.68.** We've updated the .NET Core code so it uses SkiaSharp 1.68
- **Improved compatibility with LibreOffice/OpenOffice.** LibreOffice/OpenOffice can't at the time of this writing understand indexed colors inside xlsx files. Now we introduce a new property (false by default) named `XlsxCompatibilityConvertIndexedColorsToRGB` which when true, will make FlexCel convert the indexed colors to RGB colors when saving xlsx files. Set it to true if you have xlsx files with indexed colors and you want them to be compatible with Libre/OpenOffice.
- **Now FlexCel won't throw an exception when reading custom properties in an xls file if the values of the property aren't defined.** Either because of corruption or because they were created with a tool that created wrong files, some xls files might end up having a custom property but no value associated with them. FlexCel was throwing exceptions when you tried to read the properties of those files, but that didn't allow you to get other properties which might be set correctly. So now FlexCel will just ignore those errors.
- **Bug Fix.** FlexCel would fail to read xlsx files with formulas that contained unknown user-defined functions that returned a reference type.
- **Bug Fix.** FlexCel will now render labels in a 100% stacked chart as the values, not the percent in the charts.
- **Bug Fix.** If an xlsx file contained negative offsets to a shape, FlexCel could render the shape incorrectly.
- **Bug Fix.** FlexCel will now render labels in stacked charts more like Excel renders them.
- **Bug Fix.** When rendering charts, if the axis was reversed and the labels were aligned to the right, FlexCel would render them to the left and vice-versa.

New in v 6.25 - February 2019

- **New parameter in ATLEAST tag in reports allows for the number of rows of a dataset to be multiple of a number.** Now when using `ATLEAST` you can specify that the number of rows must be a multiple of some number. That is, that the dataset must have for example 20, 40, 60... rows but not 30 or 45. Take a look at the new [Fixed Footer demo](#)
- **Improved handling of invalid "," in numeric formats.** A comma in a numeric format means "thousands separator" if it goes after the 3rd digit, like in "#,000". But when a comma is at the end of the format, it means scale: A format like "0," means divide the number by 1000. FlexCel already handled those cases correctly, but there are some

"impossible" cases like "0,0" which are not actually valid but might be saved to xlsx files. FlexCel was interpreting that the "," in some of those cases meant scale, while for the same cases Excel was interpreting "thousands separator". Now we should behave like Excel even in the invalid cases.

- **FlexCel could fail when rendering cells with more than 32000 characters.** A cell in Excel is limited to 32767 characters, but a string in GDI+ is limited to 32000 characters. So if a cell had between 32000 and 32767 characters, FlexCel would raise an Exception when rendering the file because GDI+ would fail to render the string. Now it should render correctly.

New in v 6.24 - January 2019

- **The INDIRECT function can now understand structured references in tables.** Now FlexCel can calculate formulas where INDIRECT references a table. For example if you have a table named "Table1", FlexCel will now understand a formula like `=SUM(INDIRECT("Table1[Column1]"))`
- **Breaking Change: Cell indent is now printed and rendered to pdf/images proportional to the print scale.** Before this version, FlexCel behaved just like Excel and kept the cell indent always the same no matter the print scale. Now we behave in a more logical way, and if the print scale is 50%, the cell indents will be 50% smaller. If you want to revert to the old behavior (which is how Excel behaves), there is a new property [CellIndentationRendering](#) which allows to control this behavior and revert it back to what it was. For more information read the new [section about cell indentation in the API guide](#).
- **The examples for Android show a newer way to share the documents.** The revised examples for Android now use a sharing method that is compatible with Android Nougat or newer. There is new documentation available at the [Android guide](#)
- **New methods SetRange3DRef and TrySetRange3DRef in TXIs3DRange.** The new methods [SetRange3DRef](#) and [TrySetRange3DRef](#) allow you to set a 3D range from a string like `"=Sheet1:Sheet2!A1:A3"`
- **DbValue in reports now supports fields with dots.** DbValue tag in reports will now work with fields with dots like "data.value"
- **Bug Fix.** When deleting columns the data validations formulas could be adapted wrong.
- **Bug Fix.** When a line in rich text inside a text box had a length 0 (an empty line), the font might not be preserved for that line.
- **Bug Fix.** FlexCel considered some special characters like "" in a name to be invalid when they are not. This could cause that opening and saving an xlsx file with names like that would make Excel crash opening the file.

New in v 6.23 - November 2018

- **Updated minimum Required Android version to 8.0 Oreo.** As required by Xamarin and Google Play, now the minimum supported Android version is 8.0 (API Level 26). We removed calls to deprecated methods and now require methods only available in API Level 26 or newer.
- **New methods UnshareWorkbook and IsSharedWorkbook in ExcelFile.** The method [UnshareWorkbook](#) allows you to remove all tracking changes from an xls file. (FlexCel doesn't preserve tracking changes in xlsx files). [IsSharedWorkbook](#) allows you to know if an xls file is a shared workbook.
- **New method PivotTableCountInSheet in ExcelFile.** The method [PivotTableCountInSheet](#) returns the number of pivot tables in the active sheet.
- **Support for calculating function RANK.AVG.** Added support to calculate the Excel function Rank.AVG which was introduced in Excel 2010. See [supported excel functions](#).
- **Now you can find see the call stack in circular formula references when you call RecalcAndVerify.** Now [RecalcAndVerify](#) will report the call stack that lead to a cell recursively calling itself, making it simpler for you to track those down in complex spreadsheets. Take a look at the modified [Validate Recalc demo](#) with a file with circular references to see how it works.
- **Bug Fix.** Some xlsx files with legacy headers could fail to load.
- **Bug Fix.** The function IFNA could in very rare corner cases return #N/A if its first parameter was #N/A instead of returning the second parameter.
- **Bug Fix.** There could be an error when copying sheets between workbooks and the sheet copied had a shape with a gradient.
- **Bug Fix.** Floating point numbers that were either infinity or not-a-number were saved wrong in the files and Excel would complain when opening them. Now they will be saved as #NUM! errors. Note that this only happened if you set a cell value explicitly to Double.NAN or Double.Infinity. Formula results which were infinity or nan were already handled fine.

New in v 6.22 - October 2018

- **Support for Excel 2019.** Because we support Excel 365 and changes in Excel 2019 are a recollection from the changes in office 365 from 2016 up to now, FlexCel already supported Excel 2019. For example, support for recalculating the new functions introduced in Excel 2019 was introduced by [FlexCel 6.7.16](#) back in march 2016. But this new FlexCel version adds a new `TRecalcVersion.Excel2019` enumeration which will [avoid the question about saving for changes when closing the file](#). It also adds a "v2019" enumeration to `TFileFormats`, which allows you to specify you want the file to identify itself as office 2019 and comes with empty 2019 files to be created with `NewFile`. Empty 2019 files are virtually identical to empty 2016 files, but the colors "Accent1" and "Accent5" in Excel 2016 are swapped to correspond to "Accent5" and "Accent1" respectively in Excel 2019.

- **Reports now can use tables as datasources.** Now you can use Excel tables as sources for reports. Take a look at the new [Tables as datasources](#) demo and the [section about excel tables in the Report designers guide](#).
- **New method to rename tables.** The new method [RenameTable](#) can rename a table to a newer name, changing all references in formulas to the new name.
- **New Debug mode for Intelligent Page Breaks.** Now you can use the property [DebugIntelligentPageBreaks](#) in a report, or the methods [DumpKeepRowsTogetherLevels](#) and [DumpKeepColsTogetherLevels](#) in the API to debug how intelligent page breaks are working. Look at [intelligent page breaks in the API Guide](#) for more information on how to use the feature.
- **Better drawing of conditional formats at very low or high zoom levels.** Now icons and databars in conditional formats dynamically adjust the margins to look better at high or low zoom levels.
- **Bug Fix.** Cell indent was not being considered when autofitting rows or columns.
- **Bug Fix.** FlexCel wouldn't let you rename a sheet to the same name but with different upper or lower cases.
- **Bug Fix.** CountIF, CountIFs and similar xlf/xlfs functions could return ERRNA if one of the conditions was an unknown user function, instead of returning 0 as Excel does.
- **Bug Fix.** The function Rank.EQ was ignoring cells with errors while Excel returns the first cell with error if any cell in the range has an error.
- **Bug Fix.** Inside a `<#preprocess>` section of a report a `<#delete row>` or `<#delete column>` tag could end up deleting the wrong column.
- **Bug Fix.** Error when calculating What-If tables that had their variables in a different sheet.
- **Bug Fix.** When deleting rows in reports with multiple levels of intelligent page breaks the engine could calculate more page breaks than necessary.
- **Bug Fix.** FlexCel will now validate that a table isn't named the same as a defined name or vice-versa, to avoid creating invalid Excel files.
- **Bug Fix.** When rendering a file to pdf or images FlexCel could pick the wrong normal font in very rare cases.
- **Bug Fix.** APIMate could report code that wouldn't compile for embedded xml content.

New in v 6.21.6 - September 2018

- **Updated SkiaSharp to 1.60.3.** FlexCel will now use SkiaSharp 1.60.3
- **Improved Linux support.** Many small bug fixes and updates to the Delphi Linux support.
- **Bug Fix.** FlexCel would fail to load "Strict Open XML files" with formulas which returned dates.
- **Bug Fix.** FlexCel could crash when rendering xls files with rare images.

New in v 6.21.5 - August 2018

- **Bug Fix.** FlexCel could fail to parse complex structured references in tables.
- **Bug Fix.** Formulas that referred to different files could refer to the wrong sheet on those linked files in some rare cases.
- **Bug Fix.** the IFERROR function could give a #VALUE! error in some cases when used chained with other functions.

New in v 6.21 - July 2018

- **Bug Fix.** If a "rows to repeat at top" or "columns to repeat at left" range was outside the print area, FlexCel would ignore it, while Excel will use it anyway. Now FlexCel behaves like Excel and uses the repeating range even if it is outside the print area.
- **Bug Fix.** When in R1C1 mode, full ranges expanding more than 1 row or column like for example Sheet1!3:5 could be returned as Sheet1!5 only.
- **Bug Fix.** Sometimes cells formatted as "center on selection" were not rendered when exporting them to pdf or html.
- **Bug Fix.** When hiding a column without a given width and the default column width was different from the Excel default, the column wouldn't be hidden when saving as xls.
- **Bug Fix.** There could be an error in ClearSheet with some special images.
- **ApiMate now reports hidden sheets.** ApiMate will now tell you how to hide sheets.
- **Improved chm help.** The chm help shipped with FlexCel could show javascript errors in some Windows versions.

New in v 6.20 - June 2018

- **Full support for reading and writing Data Connections in xlsx files.** Now you can use the new methods [GetDataConnections](#) and [SetDataConnections](#) for reading and writing the data connections in xlsx files. As usual, APIMate will show you the commands needed to enter new DataConnections. Note that the new methods only work in xlsx files, not xls, and there is no support for refreshing data queries from FlexCel. Only to read or write connections.
- **Improved performance with thousands of merged cells.** We rewrote the merged cell handling engine to make it faster and work better when there are thousands of merged cells.
- **Breaking Change: Improved chart rendering.** Now FlexCel recalculates the size of the legends of the charts if those are docked to the top, bottom, right, left or top-right. So if the size of the series change, the legend box and the rest of the chart will adapt. There are also other small tweaks in the chart rendering engine to make xls charts more faithful to what Excel shows. **Note:** The Excel chart engine has changed a lot since the Excel 2003 times, and Excel 2003 doesn't display charts exactly as Excel 2016. We can't make it work both ways, so this update makes the chart rendering more like Excel 2016. If you were

rendering old files and relied in the exact position of the legend, this update might move the position of the legend a little, to position it how Excel 2016 would and not how Excel 2003 would. This is why it is a breaking change.

- **New overloads for methods `SetCellFromString` and `GetStringFromCell` now accept cell references.** The methods `SetCellFromString` and `GetStringFromCell` now can use references like A1 or "Sheet1!B3". This is a shortcut in using a `TCellAddress` object to get the row and column from the reference.
- **New overload for method `TPartialExportStart.SaveCss` which allows to save the css without the `<style>` and `</style>` tags.** There is now an overload of `TPartialExportState.SaveCss` with a parameter that allows to get the inner html of the classes definition, without the `<style>` and `</style>` enclosing tags.

New in v 6.19.5 - May 2018

- **Now functions `CUMIPMT` and `CUMPRINC` are supported when recalculating.** Now FlexCel can recalculate the functions `CUMIPMT` and `CUMPRINC`
- **New methods `GetTokens` and `SetTokens` in `ExcelFile` allow you to parse arbitrary text.** The new methods `GetTokens` and `SetTokens` allow you to parse any text into tokens and then convert those tokens back into a string. Those methods complement the existing `GetFormulaTokens` and `SetFormulaTokens`
- **The `XlsChart` object now returns the 3D properties for xls charts.** Now you can read the 3D properties in charts inside xls files.
- **Improved Excel 95 compatibility.** Now FlexCel can read some Excel 95 files which would throw errors before.
- **Now FlexCel preserves "new style" sheet and workbook protections in xlsx files.** Both FlexCel and Excel use an old algorithm to compute sheet and workbook protections, and they both keep doing it this way as it is the only way to port the protections between xlsx and xls files. But some third-party generated files could have a newer style of protections which are incompatible with xls and FlexCel wasn't understanding them. Now FlexCel will preserve those new style protections in xlsx files too. The new style protections will be lost if you save as xls, but that happens in Excel too.
- **When wrapping text, now FlexCel recognizes different kind of unicode spaces.** Now other spaces in addition of character 32 are used as separators when rendering the file and wrapping the text. Note that not breaking spaces (char 160) are still not used as separators as they aren't supposed to break a line.
- **Bug Fix.** `SetCellFormat` with `ApplyFormat` could format the cells wrong if the cells were empty and there was column or row format applied.
- **Bug Fix.** Sometimes when copying sheets form different files, some named ranges would not be copied.
- **Bug Fix.** Khmer text could be rendered wrong in some rare cases.
- **Bug Fix.** When exporting to pdf you could get an error if a character didn't exist and `fallbackfonts` was empty.

New in v 6.19.0 - March 2018

- **Support for Khmer language when exporting to pdf.** The PDF engine in FlexCel now includes a Khmer shaper which is able to correctly create Khmer documents, as long as the Khmer fonts you are using are OpenType (that is they contain GSUB and GPOS tables).
- **Reduced memory usage when exporting.** Exporting to PDF and SVG were tweaked to consume less memory in high-performance environments where many threads are exporting at the same time. Also the performance of the pdf engine was improved.
- **Updated the SkiaSharp library used in .NET Core to the latest.** SkiaSharp used is now 1.60, and code was adapted to remove LockPixels/UnlockPixels which don't exist anymore. Note that due to this changes, FlexCel won't work anymore correctly with SkiaSharp 1.59.
- **Images made transparent with Excel now are converted between xls and xlsx files.** Now FlexCel will convert the transparent color parameter between xls and xlsx files.
- **Bug Fix.** In some cases after copying rows, then deleting sheets and then inserting or deleting rows, the formula bounds cache could be invalid and formulas would fail to update in the lase deleting of the rows.
- **Bug Fix.** The round function now behaves more like Excel and not like C# in some border cases.
- **Bug Fix.** Formulas with intersections of a name with a 3d range would be interpreted as #name instead of the correct formula.
- **Bug Fix.** In some invalid cases the indirect function would throw exceptions that would be later processed. While the result was still ok, those exceptions could slow down a lot recalculation in a file with thousands of formulas.

New in v 6.18.5.0 - January 2018

- **New convenience methods SetCellValue(cellRef, value) and GetCellValue(cellRef).** The new methods [SetCellValue\(cellRef, value\)](#) and [GetCellValue\(cellRef\)](#) allow you to set or get a cell value directly from a text reference like "A1" without having to use a [TCellAddress](#) class.
- **Support for shape connectors in xlsx.** Now FlexCel will preserve connections between shapes in xlsx, and convert them from xls to xlsx and viceversa. We've also added the properties [IsConnector](#) [StartShapeConnection](#) and [EndShapeConnection](#) to allow you to enter connections with the API. As usual, APIMate will tell you the code needed to add a connector from an Excel file.
- **Bug Fix.** The VLOOKUP and HLOOKUP functions now support wildcards (* and ?) in search strings.
- **Improved compatibility with invalid files generated by third-party tools.** Some xlsx generators can write invalid column widths. Now when exporting to html/pdf, FlexCel will consider those widths as default (like Excel does) and not 0 (as it used to do).
- **Bug Fix.** FlexCel could fail to parse some structured references in tables.

- **Bug Fix.** When calculating UDFs and there were errors in the arguments, FlexCel could in some cases return #ERRNAME! instead of evaluating the UDF.
- **Bug Fix.** In some files the calculated height for items inside Forms Listboxes was too big.
- **Bug Fix.** FlexCel failed to read custom document properties saved in UTF16.
- **Bug Fix.** Reports with `DeleteEmptyBands = TDeleteEmptyBands.ClearDataOnly` would not clear text inside textboxes or hyperlinks.
- **Bug Fix.** In some bidirectional reports with `report.DeleteEmptyBands = TDeleteEmptyBands.MoveRangeUp` the tag text was not erased.

New in v 6.18.0.0 - December 2017

- **Support for default CryptoAPI xls encrypted files.** Now FlexCel can read and write xls files encrypted with the CryptoAPI encryption. This is the default encryption algorithm for files created by Excel 2003 or newer. With this addition, all modes and encryption algorithms in both xls and xlsx are now supported.
- **Full support for manipulating XML Mappings in xlsx files.** Now XML Mappings will be preserved when opening and saving xlsx/m files, and there are two new commands in the API to set them or read them with code. The new commands are `GetXmlMap` and `SetXmlMap`. As usual, APIMate will show how to use `SetXmlMap`. **Note:** The new API only works in xlsx/x files, not xls. Xml mappings inside xls files will still be preserved when opening and saving xls files, but not converted between xls and xlsx.
- **Bug Fix.** Images made transparent with Excel tools might not preserve their transparency when saved as xlsx.
- **Bug Fix.** in .NET Core 2.0 Exceptions thrown by FlexCel would display the message '*Secure binary serialization is not supported on this platform*' instead of the actual error message.
- **Bug Fix.** When rendering shapes with semi-transparent gradients to PDF or SVG the gradients were exported as fully opaque.
- **Bug Fix.** Files with table slicers saved by FlexCel might not open in Excel 2013. (They already worked fine in Excel 2016, and Excel 2010 doesn't support table slicers).
- **Bug Fix.** Rotated shapes inside groups in xlsx files could be rendered wrong.
- **Bug Fix.** Groups that were flipped horizontally or vertically weren't flipped when rendering. Objects inside were flipped, but the groups themselves weren't.
- **Bug Fix.** Filled polygons could be exported wrong to PDF in some border cases.
- **Bug Fix.** Filled polygons could be exported wrong to images with the SKIA backend used in .NET Core and Android.
- **Bug Fix.** Legacy system colors in drawings inside xls files could be rendered as transparent instead of the correct color in border cases.
- **Bug Fix.** Xlsx files with complex gradients where the stops were not sorted could cause invalid PDF files.

- **Bug Fix.** Textboxes with more than 8224 characters would corrupt the file when saved as xls.
- **Updated SkiaSharp to 1.59.2 for .NET Core.** Now FlexCel will require SkiaSharp 1.59.2 when using it in .NET Core.

New in v 6.17.4.0 - November 2017

- **Breaking Change: Subtotal command allows more customization.** Now the [Subtotal](#) command provides more parameters in the callbacks to allow for more customization. In addition, by default it will write a better text for non sum aggregates (like for example "Customers Average" instead of "Customers Total" if you are using the Average to aggregate). There is also a new example on how to use the command. **Note:** This is a breaking change if you are using the callbacks since now the callbacks have more parameters. But it is easy to fix at compile time, just add those parameters to the callbacks and recompile.
- **New SubtotalDefaultEnglishString command.** Now the [SubtotalDefaultEnglishString](#) provides the string used by the different aggregate functions used in [Subtotal](#) . You can use this method as a parameter to subtotal to calculate the grand total and subtotal labels.
- **Ability to copy OLE objects between different files while using xlsx file format.** Now the restriction that you can't copy sheets from one file to another if they have embedded OLE object has been removed for xlsx files. It is still not possible to copy sheets between different files with embedded OLE objects in xls.
- **Ability to read custom document properties in xls files.** Up to now FlexCel could only read custom document properties in xlsx files. Now it can also read them in xls files. And now custom properties are migrated from xls files to xlsx too.
- **Better handling of URL encoding when encoding some filenames.** Now some filenames containing some characters like "#" will be correctly encoded when linked from FlexCel. The events that allow you to manually define the links have a new parameter "UrlNeedsEncoding" which you can set to false to avoid all encoding by FlexCel if you provide an already encoded URL to the event.
- **Bug Fix.** The **Last print time** document property wasn't read in xlsx files.
- **Bug Fix.** When copying cells from one file to another autofilters would be copied even if they were not in the range being copied.
- **Bug Fix.** Formulas referencing sheets which could be interpreted like a R1C1 cell reference (like "R3C5") were saved without quotes in the sheet name, and thus became invalid formulas.
- **Bug Fix.** Modified and creation time were read in UTC, but saved in local time which would result in a different date being saved back. Now it is all handled in UTC.
- **Bug Fix.** In some very complex bidirectional reports with sorting in the template the fields might end up not being sorted correctly, and some might appear twice.

New in v 6.17.3.0 - October 2017

- **New TFlxNumberFormat.PercentCount method.** The new method [TFlxNumberFormat.PercentCount](#) allows you to know how many non escaped % symbols are in a format string.
- **Better display of negative zero numbers.** Now a negative number that displays as zero like "-0.001" formatted with a "0.0" format string will display as "0.0" and not "-0.0"
- **iOS demos updated to require iOS 8 or later so they can be compiled with XCode 9.** iOS demos targeted iOS 6, which isn't supported in XCode 9. Now we target iOS 8.

New in v 6.17.2.0 - October 2017

- **Better support of machine locale formats.** Before this version, whenever you used a "machine dependent" date numeric format (those shown with * in Excel), FlexCel would use always 2-digit months, 2-digit days and 4-digit years. (as in 01/02/2000). Now it can use single digits for days and months, and 2 digits for years (as in 1/2/00) if your machine locale is set to a format that uses those.
- **Improved xls chart rendering.** Now series which are in hidden columns or rows won't count as series at the moment of drawing the chart, to better copy Excel behavior. Before this version those series would appear empty, but still take space in the chart.
- **Improved compatibility with invalid 3rd party xlsx files.** Now FlexCel can open files where the case of the files inside the container is incorrect. This happens with files generated by "1C" database and might happen with other 3rd party files.

New in v 6.17.0.0 - September 2017

- **Full Support for Excel tables in xlsx files.** This release completes the support for tables in the FlexCel API introduced in 6.11.
 - Tables are now exported to PDF/HTML/SVG/Images and printed with all the table formatting including banded columns and rows, etc. All formatting is supported.
 - Now FlexCel can recalculate the [structured references](#) used in tables. Everything is supported, from simple references like Table1[@column] to references in tables from another file. (for external table references you need to create a [Workspace](#))
 - Complete API for adding, deleting or modifying tables with code. APIMate was modified to show how to use the new things in the API.
 - API for adding, deleting or modifying custom table styles. APIMate shows how to enter table styles with code.
- **Support for reading and writing Strict Open Xml files.** Now FlexCel can read and write [Strict Open XML spreadsheets](#). The default is to save to strict xml only if you opened a strict xlsx file and saved it, in the other cases we fall back to the standard transitional xlsx. There is a new property [StrictOpenXml](#) which you can set to force saving as strict xlsx, and read to know if the file you opened was strict xlsx.

- **Support for .NET Standard 1.5, 2.0 and .NET Core 2.0.** FlexCel nuget package contains a .NET Core 2.0 assembly, and .NET standard assemblies. The .NET standard assemblies can be referenced in multiplatform projects and they will be replaced by the corresponding native assembly.
- **Ability to add autoshapes to charts.** Now the existing method [ExcelFile.AddAutoShape](#) works also in chart sheets, and there is a new method [ExcelChart.AddAutoShape](#) that allows to add shapes to charts embedded inside a sheet.
- **FlexCel will now preserve embedded OLE objects in xlsx files.** Now FlexCel will preserve embedded OLE documents (like for example a word document) in xlsx files.
- **Improved performance in reports with thousands of hyperlinks.** Now FlexCel is much faster dealing with thousands of hyperlinks in reports.
- **<#row height> and <#column width> tags in reports now accept expressions.** Now you can write something like <#row height(<#someexpression>)> where expression will be calculated at the moment of running the report.
- **Now FlexCel converts strings with timestamps to dates more like Excel.** In Excel you can write a string with an invalid timestamp like "3:61" (3 hours 61 minutes, which is 4 hours 1 minute) and it will be accepted by the system. FlexCel was rejecting those timestamps, but now it accepts them just like Excel.
- **Support for #GETTING_DATA error in TFlxFormulaErrorValue.** The enumeration [TFlxFormulaErrorValue](#) now contains a new `ErrGettingData` member which corresponds to the type in Excel. Also [Error.Type](#) function will return 8 for this error. Note that Excel doesn't save this error in xlsx files (it saves #N/A instead), but it does save it in xls files. FlexCel preserves it in both.
- **Better support for comments in xlsx file in high dpi.** The size of the comments is preserved better now when `ScreenScaling` is `> 0`
- **Now ExcelFile.RenderObject can render shapes inside groups and use an objectPath parameter to specify the name of the object to render.** There are new overloads in [ExcelFile.RenderObjects](#) and [ExcelFile.RenderObjectAsSVG](#) that take an **objectPath** parameter. This allows you to render an individual shape inside a group instead of the full group, and also to specify directly the name of the shape to render as in

```
xls.RenderObject(1, "@objectname")
```
- **Reduced memory usage when loading fonts for exporting to PDF.** We've optimized the pdf font engine so it uses less memory when loading the fonts.
- **Support for returning arrays with the INDIRECT function.** When doing a *Sum*, *SumIf* or *N* of an *Indirect* function which returned an array, FlexCel worked like Excel 2003 or older and only used the first value of the array. Now it uses the full array in *SumIf* and *N* like Excel 2007 or newer, and in *Sum*, like Excel 2010 or newer. This allows you to write formulas like the ones mentioned here: <https://www.pcreview.co.uk/threads/indirect-function-limitations.1750391/> Note that this formula behavior is exclusive to Excel 2010 or newer: Neither LibreOffice or Google docs implement it.
- **All examples available in Github.** Now besides being available with the setup and at the [documentation site](#) the examples are also available on [Github](#)

New in v 6.16.0.0 - August 2017

- **User defined formats in reports.** Now you can define a function in code that will format the cells depending on complex rules. Take a look at the new [User Defined Formats demo](#) to see an example on how to use them.
- **Now when signing PDFs, FlexCel will mark the generated files as requiring Acrobat 8.** Due to known vulnerabilities in SHA1, signing with SHA1 is deprecated. So now the FlexCel signing demos have been modified to use SHA512. As SHA512 requires Acrobat 8 or newer, now the files will be marked as requiring Acrobat 8 or newer. Note that older acrobat versions will still be able to see the file, but they won't validate the signatures.
- **Added recalculation support for new functions.** Added support for [CEILING.MATH](#) , [FLOOR.MATH](#) functions added in Excel 2013.
- **Improved behavior of CEILING/FLOOR functions.** Now the existing CEILING, FLOOR, ISO.CEILING, CEILING.PRECISE and FLOOR.PRECISE functions are calculated more in the way Excel uses them, with a higher precision to prevent rounding errors like 1.00000...00001 to be rounded up to 2.
- **Breaking Change: Support for double underlines when exporting to pdf and refactored TUIFont.** Now double underlines will be exported to pdf. In order to support this we had to remove the **Underline** and **Strikeout** parameters from the [TUIFont](#) definition, and put them in a separate [TUITextDecoration](#) structure. So now the [DrawString](#) methods in [PdfWriter](#) require a new TextDecoration parameter, and you don't specify the underline or strikeout anymore in TUIFont. Also as now TUIFont is different from GDI+ Font by not having underline into it, now you can't convert automatically between a Font and a TUIFont.

This change can break some code if you are using TUIFont and DrawString directly, but it should break at compile time and be straightforward to fix. Just create the TUIFonts without underline, and specify a text decoration when calling drawstring with those fonts. Take a look at the updated [Creating pdf files with pdf api](#) demo to see how it works now.

- **New methods TUIFont.CreateFromMemory and TUIFont.CreateFromFile.** The new methods [TUIFont.CreateFromMemory](#) and [TUIFont.CreateFromFile](#) allow you to create TUIFonts from fonts not installed in the system.
- **Improved conversion of control points in autoshapes between xls and xlsx files.** Now for some shapes like a roundrect or a smiley face are converted better to xlsx when read from xls files. The default control points for those shapes weren't converted correctly.
- **Now you can enter macros that refer to other files with the API.** Now when you call [AddButton](#) or similar methods, you can use a macro that refers to a different file like file2! macro1. As usual APIMate will report the exact syntax to link to a different file.

- **Breaking Change: Added a new parameter to ExcelFile.RecalcRange.** Now when you call [RecalcRange](#) you need to specify if the formula has relative references (as is the case in conditional formats, data validations and names) or if the references in the formula are absolute (as it is the case in normal spreadsheet formulas). Before this version RecalcRange assumed absolute references, so if you are updating existing code and want to keep the exact behavior just add ", false)" as last parameter. But make sure to review that the formula is not a relative formula.
- **Breaking Change: Bug Fix.** The parameters MaxWidth and MinWidth of the <#column width> and <#row height> tags weren't working properly when autofitting. Now they work according to the docs. If you were using MaxWidth and MinWidth in <row height(autofit...)> or <column width(autofit...)> please review those tags and make sure minwidth and maxwidth are in the correct positions.
- **FlexCel will now check names of tables are valid when you create a table with the API.** Now FlexCel won't let you name a table with an invalid name (like for example a name containing spaces).
- **Bug Fix.** When the print zoom was bigger than 100% the maximum column to print could be calculated wrong.
- **Bug Fix.** When evaluating data validations with [CheckDataValidation](#) introduced in FlexCel 6.15, INDIRECT functions using RC notation were evaluated wrong.
- **Bug Fix.** When doing bidirectional reports with multiple horizontal master detail X ranges, the rows for the vertical ranges could be wrong.
- **Bug Fix.** When a SPLIT tag was used inside a multiple master-detail relationship in a report the results could be wrong.
- **SKIA library updated to the latest.** We have updated the code that uses the SKIA library in .NET Core to the latest version of the library. We removed calls to deprecated methods and replaced them with equivalent methods.
- **Bug Fix.** When exporting arabic rich text with multiple formats in the same cell and a scale factor different from 1 to pdf the results could have the wrong font sizes.
- **Bug Fix.** Row() and Col() functions would return 1 when called from <#Format range> or <#Delete range> tags. Now they return the row and column of the cell where the tag is written.
- **Examples and demos now use SQLite.** The database used in the examples was migrated from SQL Server CE to SQLite. This was because SQL Server CE is deprecated by Microsoft.
- **Bug Fix.** Setup wasn't correctly registering the .NET 4 and newer assemblies in Visual Studio, so those wouldn't appear in the "Add reference" dialog as Extensions. (They were still available if you browsed for them)

New in v 6.15.0.0 - May 2017

- **Ability to check and evaluate data validations.** The new methods [CheckDataValidation](#) , [CheckDataValidationsInSheet](#) and [CheckDataValidationsInWorkbook](#) allow you to check if the values of a cell, sheet or workbook are in conformance with the data validations in those cells. You can also check if a value would be valid when entered into a cell with [CheckDataValidation](#)
- **Improved tagging of PDF Files.** Now repeated columns and rows are tagged as artifacts and not real content. This will also fix a rare bug that could happen when exporting a file with "Columns to Repeat at left" to tagged pdf.
- **Improved HTML5 exporting.** Updated the HTML5 exporting to comply with the latest HTML5 standard.
- **Improved bidirectional reports.** Now you can make a bidirectional report where the vertical range doesn't cross the horizontal range, but just is contained exactly in the top and bottom coordinates of the horizontal range.
- **Improved unicode Bidi Algorithm.** Updated the bid algorithm from version 3.0 to 6.3. Fixed errors that could happen when using invalid unicode characters.
- **Improved compatibility with invalid files.** Now FlexCel won't throw an exception by default when the file has invalid hyperlinks. You can change this behavior by setting the new "ErrorOnXlsxInvalidHyperlink" property in [ExcelFile.ErrorActions](#)
- **Bug Fix.** FlexCel failed to open some encrypted Xlsx files saved in Excel 2013 or newer where the key size in the algorithm to encrypt the file was different from the key size in the algorithm to encrypt the key.
- **Bug Fix.** When exporting to html a merged cell which had columns or rows not hidden but with zero width or height could render wrong.
- **Bug Fix.** Now you can query if a page break at row 0 exists. A page break at row 0 would mean a page break before the first row, and Excel doesn't obey it, but you might end up with such a page break when deleting rows. Now you can check if there is a page break on that row.

New in v 6.14.0.0 - April 2017

- **New documentation center.** We've completely redesigned the documentation, including lots of new code examples, a tips and tricks section and much more. We've manually reviewed all the user guides to make sure they are up to date with the latest information. You can find the new documentation center at [our website](#) **Note:** This version removes support for integrating help in Visual Studio 2012 or older. For those versions you can't use F1 to get help and will have to manually search in the web or in the included chm file. Also, when you press F1 in FlexCel types you will be redirected to the FlexCel online help. Setup won't integrate the offline help anymore for any Visual Studio version.
- **Official support for .NET Core including graphics.** Now FlexCel for .NET Core is out of the beta and fully functional. It now also supports creating pdf files, html, etc., using [SkiaSharp](#) for the graphics support.

- **Support for .NET 4.7.** There is a new FlexCel.dll targeting .NET 4.7.
- **Improved Setup.** Now the setup.exe for Windows will include .NET Core and Xamarin libraries and examples, so there is nothing extra to add. The nuget package flexcel-dnx for .NET Core has been now replaced by FlexCel.nupkg which supports all platforms that FlexCel supports. The setup will now automatically register the FlexCel NuGet package folder so you can just add it from Visual Studio.
- **Breaking Change: Deprecated Xamarin Components.** Now we don't distribute FlexCel anymore via Xamarin Components. Those have been replaced by the **TMS.FlexCel NuGet package**. If your apps used Xamarin Components, remove them and add a reference to the TMS.FlexCel NuGet package.
- **Support for Web Addins, either of Content or Task pane types.** Now FlexCel will preserve Web addins in xlsx files. There are 2 new methods in the API: `ExcelFile.HasWebAddinTaskPanels` and `ExcelFile.RemoveWebAddinTaskPanels` which you can use to know if there are any task pane addins in the file and remove them. The content addins are just objects and you can remove them with `DeleteObject`. You can find if an object is a Web Add-in by calling the new property `TObjectProperties.IsWebAddin`
- **Support for Table Slicers.** Now FlexCel will preserve Table Slicers in xlsx files (xls files don't support them). Note that Pivot Table Slicers were already preserved, this refers to the new Table Slicers in Excel 2013.
- **New static method `CreateKeepingAspectRatio` in `TClientAndhor` allows you to fit an image inside a range of cells maintaining the aspect ratio of the image.** You can either specify the 4 coordinates of the range where you want the image inside, and have the image centered or aligned inside that range, or you can leave one of the coordinates at -1. If you set Row2 or Col2 at -1, then this method will create an image that fits in the other Col1-Col2 or Row1-Row2 respectively, and keeps the aspect ratio.
- **New method `SetTable` allows to modify existing Tables.** While you could modify existing tables by using `RemoveTable` and `AddTable`, now you can modify them directly with `SetTable`.
- **New `DrawBorders` method in `ExcelFile` allows to quickly draw a border around a range of cells.** This new method is just a shortcut for calling `SetCellFormat` but it is a little easier to discover and use.
- **Now you can specify multiple folders with fonts when exporting to pdf.** Now in the `OnGetFontFolder` event of a `FlexCelPdfExport` you can return a list of strings separated by semicolons. So you can return a string like "c:\font1folder;c:\font2folder" and FlexCel will search for the fonts inside font1folder and font2folder.
- **Now `<#includes>` in reports will balance in the containing band.** Now when you include a subreport in a report, the main parent will be balanced as it is with ranges inside ranges.
- **Bug Fix.** When "Precision as displayed" was set to true in the file options, the recalculation engine could calculate some values with a different precision than the one in the cell.

- **Bug fix.** When a file with dates starting in 1900 had a linked formula to another file with dates starting in 1904 the value of the dates in the 1904 file would be considered to be at 1900. Similar for a 1900 file linking to a 1904.
- **Bug fix.** In some rare cases when changing a style an exception could be thrown.

New in v 6.13.2.0 - February 2017

- **New method XlsFile.AddAutoShape to add autoshapes with the API.** The new method allows you to add an autoshape to a file. As usual APIMate will provide you with the needed code.
- **Support for Visual Studio 2017 RC.** Setup now installs in Visual Studio 2017.
- **Support for .NET Core 1.1.** Project.json was changed to a csproj in order to build in 1.1.
- **Support for drawing mirrored images when rendering.** Now when a bitmap is flipped vertically or horizontally, FlexCel will also draw it like this when exporting it.
- **New method TCellAddress.DecodeColumn.** This method is provided for symmetry with the existing TCellAddress.EncodeColumn, but it is normally not required as you can use TCellAddress to get the full cell address string for a row and a column. This method could be used in the rare case you only wanted the column string and not the full cell address.
- **Breaking Change: Support for losslessly rotated JPEG images.** Now when inserting JPEG images that are rotated via the "orientation" attribute in the JPEG file, FlexCel will automatically rotate the image so it appears with the desired orientation in Excel. Before FlexCel behaved as Excel 2010 or older, just entering the image as is, so Excel would show it rotated. Now it works like Excel 2013 or newer, where we rotate the image in Excel to compensate. All orientation values are supported, included mirrors. Note that this change might be breaking if you were manually rotating the images before entering so they would show fine. If you were rotating the images manually, you should remove the code as now it will be done automatically. There is also a new method `ImageUtils.GetJPEGOrientation` which you can use to tell if a JPEG image is rotated or not.
- **Bug Fix.** When autofitting a column which contained multiple lines but the cell was set to not wrap, FlexCel would consider that the cell could wrap and so end up with a smaller column width than needed.
- **Bug Fix.** Xml declarations when writing custom xml parts could be duplicated.
- **Bug Fix.** When replacing hyperlinks in a shape with a report there could be an exception.
- **Bug Fix.** Support for reading xlsx files with custom properties with repeated names or empty names.

New in v 6.13.0.0 - January 2017

- **Support for rendering Right-To-Left sheets.** Now FlexCel can export sheets where A1 is at the right side of the page and the cells grow to the left instead of to the right. A new property `XlsFile.SheetIsRightToLeft` allows you to read or write the right to left state of the sheet directly without needing to use `SheetOptions`. `APIMate` will now also suggest `SheetIsRightToLeft` instead of `SheetOptions` for RTL sheets.
- **Improved right to left support for text.** Now FlexCel will support mixed right to left and left to right text more as the Unicode BIDI algorithm defines it. The **Context** property of the cell is now also used to figure out if it is rtl text embedded in ltr text or ltr text embedded in rtl text.
- **New static properties** `ExcelFile.CompressionLevel` , `FlexCelConfig.XlsxCompressionLevel` and `FlexCelConfig.PdfPngCompressionLevel` . Properties `ExcelFile.CompressionLevel` and `FlexCelConfig.XlsxCompressionLevel` are the same and control the zip compression level used to creating xlsx files. `FlexCelConfig.PdfPngCompressionLevel` controls the compression level for pdf and png files. FlexCel uses "zcDefault" zlib compression level, which normally gives the best ratio between speed and size. But note that Excel itself uses `zcFastest` when saving xlsx files, resulting in faster saves but also bigger files. While you won't probably want to change the defaults, now you can. Note: We require .NET 4.5 or newer for this property to work.
- **New static property `FlexCelConfig.DpiForReadingImages`.** This new property `FlexCelConfig.DpiForReadingImages` allows you to force a resolution in the images you are loading. Normally FlexCel will use the resolution stored in the images to calculate the desired width in inches, but now you can override whatever is saved in the file by changing this property.
- **New implementation of wildcard matching for all functions that use them.** The new algorithm to match patterns like * or ? and used in functions like `MATCH` or `COUNTIF` is now much faster and can use much less memory in pathological cases.
- **New method** `FlexCelReport.Run(Stream templateStream, Stream outputStream, TFileFormats fileFormat)` . This method allows you to specify the resulting file format when running a report to a stream.
- **Better handling of expressions or formats defined both in an included report and the master report.** Now when an included report has the same expressions or formats defined in the config sheet as the master, those local definitions will be used, instead of raising an error of repeated formats/expressions.
- **Breaking Change: Better handling of image resolution in reports.** Now when adding an image to a report and resizing it, FlexCel will take in account the image resolution if it is saved in the image. If the image doesn't have a resolution saved, FlexCel will use the screen resolution. You can revert to the old way of assuming a resolution of 96 dpi for all images by changing `FlexCelConfig.DpiForReadingImages`
- **Bug Fix.** When exporting to HTML and a merged cell covered hidden rows or columns, the resulting html could be wrong.

- **Bug Fix.** When exporting to HTML with embedded SVG images, the fill colors in the SVG images would be wrong if there were gradients.
- **Bug Fix.** When exporting to SVG, text in controls or shapes could go a little lower than it should.
- **Bug Fix.** The formula parser would fail to detect some unicode characters as valid characters for a sheet name or named range.
- **Better display of complex numeric formats.** Now we handle some complex formatting the same as Excel does, handling also invalid formats which Excel doesn't allow better.
- **Bug Fix.** Now FlexCel allows names with spaces as macro identifiers when loading files. While those aren't valid names and Excel won't let you enter them directly, you can enter them with VBA code, and FlexCel was refusing to read those files. Now FlexCel will open them correctly.
- **Bug Fix.** In .NET Core we could fail to read formulas
- **Bug Fix.** When a file had "Precision as displayed" set and there were cell formats including percentage signs, the numbers might be rounded wrong.
- **Bug Fix.** There could be an stack overflow when a camera object rendered a range of cells which included the cells where the camera object was.

New in v 6.12.0.0 - October 2016

- **Improved compatibility with Windows Phone devices.** Some windows 8.1 devices have a bug that doesn't let them read resource files (resx). This means that FlexCel would fail when deployed to those devices, even if it would work in the simulator. To fix it, now FlexCel for Windows Phone and Windows RT doesn't use resource files.
- **Improved performance when creating tens of thousands of names in a file.** Now when creating a file with tens of thousands of names FlexCel will be much faster.
- **Breaking Change: Bug Fix.** In xls files, setting `SheetProtection.Scenarios` and `SheetProtection.Objects` had the reverse effect as in xlsx files. Now xls files behave the same as xlsx files. If you are changing the protection of xls files, **review the calls to `SheetProtectionOptions.Objects` and `SheetProtectionOptions.Scenarios` as they might be reversed.** If you are saving as xlsx files, then there is no need to change anything, as xlsx already worked as expected.
- **Better drawing of labels in charts.** Now the labels inside charts draw more like Excel when exporting xls files to pdf or html. If multiple labels would overlap, now FlexCel tries to separate them. The leader lines in pie charts from the slices to the legends render better too.

- **APIMate will now suggest to use new TSheetProtection(TProtectionType.All/None) instead of new TSheetProtection(true/false).** The constructors using true and false can be confusing, because while they work, they will set all the protection to true and false, and some protections work when the property is true (contents, objects and scenarios) while the others work when the property is false (all the other properties). The constructors using TProtectionType will set some values to true and some to false as needed to have all the sheet protected or unprotected.
- **Bug Fix.** When copying sheets in a file, some conditional formats could raise a null reference exception.
- **Improved compatibility with third-party created files.** Specifically, we now can read spreadsheets created with google docs which contain pivot tables. Those generate invalid xlsx files lacking required attributes, and FlexCel would complain about them missing. Now it will ignore them, and fix the files if you open and save them in FlexCel.

New in v 6.11.0.0 - September 2016

- **Support for Excel tables in xlsx files.** There is partial support for tables in the FlexCel API.
 - Tables are preserved when editing xlsx files. Note that we refer to the tables introduced in Excel 2007: Other tables like "what-if" tables were already preserved.
 - Tables will be copied and modified when you insert or copy ranges.
 - API for reading tables
 - Preview API for writing tables. **Note that this API is not complete yet and might fail in some cases.** APIMate will show you how to add a table with the API.
 - There is no rendering yet (exporting tables to pdf, etc), and no calculation of table references like =SUM(@Table1[column2])
- **New properties FullRecalcOnLoad and FullRecalcOnLoadMode in XlsFile.**

`FullRecalcOnLoad` will tell you if the xlsx file opened with FlexCel had the property "Full Recalc on Load" true. When it is true, normally the file doesn't have the values of the calculated formulas and you need to do a manual `XlsFile.Recalc()` to get the values.

`FullRecalcOnLoadMode` allows you to tell FlexCel how it should mark the files it creates. In the default mode it will mark them as not needing full calculation id they were calculated on save by FlexCel (the default) and mark them as needing recalc on open in other case. Note that those 2 properties only apply to xlsx files: xls files don't have this property and the value returned by those properties will always be false.
- **Some repeated function results are now calculated only once for better recalculation speed.** Now FlexCel can detect repeated subexpressions inside a formula and calculate them only once. So for example if you have a thousand formulas like `=If(A1,2,3... = Sum(B1:E1000),1,Sum(B1:E1000))` then the `Sum(B1:E1000)` will be calculated only once for all the formulas. This can have significant speed improvements if you have formulas with this pattern.
- **Performance improvements in function calculations.** Some of the most used functions like SUM, COUNT, AVERAGE, SUMIF, COUNTIF, etc have been optimized to work at a higher speed.

- **Performance improvements in formula parsing.** Now the formula parser is a little faster and that can lead to faster loading of xlsx files with thousands of formulas.
- **Performance improvements loading xlsx files with thousands of comments.** Now xlsx files with thousands of comments should load much faster.
- **Improved rendering of numbers which don't fit inside a cell.** When a number doesn't fit inside a cell, Excel shows ##### instead. But it will always show at least one #, if it can't fit a complete # into the cell, then it will display it empty. FlexCel was showing part of a # sign when a full # sign wouldn't fit, not it behaves as Excel and shows the cell empty.
- **Bug Fix.** In form objects like checkboxes or listboxes saved by Excel 2007 in xlsx files, the resulting coordinates could be wrong if the value of the anchor took more than one cell. FlexCel would move the anchor to the next cell, but Excel just ignores the extra width or height. Note that this only applies to xlsx files saved by Excel 2007, xls files or xlsx files saved by Excel 2010 or later would be correctly read by FlexCel.
- **Bug Fix.** When deleting sheets with locally stored defined names and you had multiple references to those names in a single formula, FlexCel could fail to update the names.
- **Bug Fix.** When setting a column format for many columns at the same time and reset cells true, some cells might not be reset.
- **Bug Fix.** Now FlexCel won't let you enter formulas with unions ranges of numeric or string values. Before it would allow you to enter a formula like "= 1, 2" and it would be interpreted as the union of the reference "1" and "2". Excel would read it if saved as xls, but would fail to parse the formula when saved as xlsx. So now we don't allow those formulas anymore.
- **Bug Fix.** While it is invalid to write a file with conditional formats or data validations with formulas that refer to other sheets, Excel can load them (but you won't be able to modify them). Now FlexCel can read those too without reporting an error.
- **Bug Fix.** FlexCel could raise an exception when deleting ranges with conditional formats.

New in v 6.10.0.0 - August 2016

- **Support for converting conditional formats between xls and xlsx files.** Together with the support for conditional formats in xlsx files introduced in 6.9, this completes full support for conditional formats in xls or xlsx files. Now the APIs to read and write conditional formats will work in both xls and xlsx, and conditional formats will convert seamlessly between xls and xlsx files, even formats not supported in the original xls97 spec.
- **Support for using formulas in "Text" conditional formats.** Formulas weren't allowed in Excel 2007, and FlexCel 6.9 didn't allowed them either. Now they are fully supported, for Excel 2010 and newer.
- **Fixed small validation issues in the xml generated for xlsx files.** Some xlsx files generated by FlexCel could have xml that would not validate against the xlsx reference files. Excel would still open those files, but the spec wasn't correctly implemented.

- **Breaking Change: Removed "IsPercent" property from Iconset Conditional Format definitions.** IsPercent had no effect in IconSet rules and it is deprecated in Excel. As IsPercent was introduced in the previous version 6.9, it made sense to remove it before it got into wide use.
- **Bug Fix.** Sometimes with very complex groups of conditional format rules, some could be ignored when exporting to pdf.
- **Bug Fix.** Reversed iconsets were exported not reversed to pdf.
- **Bug Fix.** When copying cells with conditional formats from one xlsx file to another, the borders wouldn't be copied.

New in v 6.9.0.0 - August 2016

- **Conditional Format support for xlsx files.** We've added support for conditional format in xlsx files:
 - All xlsx conditional formats including the new ones added to xlsx in Excel 2010 are fully parsed and preserved. When you insert and copy ranges, conditional formats will adapt and be copied too.
 - Full support to read and write them with the API. APIMate also fully supports them and will show you the code you need to enter a conditional format into a file.
 - Full support for exporting all conditional formats to pdf or html. All formats are exported to pdf. When exporting to html, iconsets and databars are not exported, but they aren't exported either when you save as html in Excel. Different from Excel, we export the iconsets to pdf as vectors, so they will look nice in any zoom level.
- **Performance improvements in exporting.** Exporting files to pdf or images is now faster. Depending in your files, you might see a visible improvement.
- **Performance improvements in CSV exporting.** CSV exporting is now up to 2 times faster.
- **Breaking Change: Removed "classic" OSX and iOS components for Xamarin.** Classic iOS and OSX components are no longer supported by Xamarin packaging and they are deprecated, so we aren't shipping them anymore.
- **Font could be wrong in linked shapes.** When you had a shape whose text was linked to a cell and the value of the cell changed, the font of the shape text would be reset.

New in v 6.8.8.0 - August 2016

- **Bug Fix.** When rendering superscripts in multiline cells, the distance between lines could be wrong.

New in v 6.8.7.0 - July 2016

- **Unknown root parts preserved in xlsx/m files.** Now root parts which do not conform to the xlsx spec like ribbon customization or arbitrary parts are preserved when editing xlsx/m files.

New in v 6.8.6.0 - July 2016

- **User Customization parts preserved in xlsx/m files.** Now the buttons that you add to the *Quick access toolbar* in the ribbon for only a specific document will be preserved when you edit the document with FlexCel.
- **Support for latest version of Xamarin Android.** A change in Xamarin now requires a reference to Java.Interop: <http://stackoverflow.com/questions/37788326/c-sharp-xamarin-java-interop-error>
- **Bug fix.** Some invalid png files could cause exporting to pdf to hang.
- **Bug fix.** Fixed issues with resources in .NET Core
- **Bug fix.** A chart with an empty array as range would throw an Exception when saving in xlsx files.

New in v 6.8.5.0 - June 2016

- **Controls and drawings in xlsx now are stored in the same hierarchy and a control can be below a shape.** The xlsx file format introduced in Excel 2007 had controls and drawings in separate parts, so all controls were always above any drawing, and also it wasn't possible to group controls and images. Since Excel 2010 shapes and controls are also stored in a common stream so you can group them or put images above controls. Now FlexCel reads and writes the Excel 2010 parts if available, and correctly puts the controls below the images or grouped if needed. Note that this applies only to xlsx, it was always possible to group controls and images in xls.
- **Bug fix.** Grouped shapes with more than 2,147,483,647 emus height (approximately 38,000 rows with standard row heights) would be truncated in xlsx files (xls files are always truncated anyway since that is a limitation of the file format).
- **Bug fix.** External links could fail to load in xlsx files in Excel 2007. (Excel 2010 and up were already ok)

New in v 6.8.4.0 - June 2016

- **New XlsFile.RenderCells overload which allows to render objects and borders.** Now RenderCells can render also the objects which are in the group of cells.
- **Support for rendering linked images (camera tool) to pdf/html/etc.** Now FlexCel will update the linked images when rendering to show the correct image.

New in v 6.8.2.0 - June 2016

- **Support for preserving ActiveX controls in xlsx files.** Now FlexCel will preserve ActiveX controls when you open, modify and save xlsx/m files. ActiveX objects are not converted between xls and xlsx files.

- **Form controls are now read from and written to the Excel 2010 stream besides the Excel 2007 stream.** FlexCel now reads the Excel 2010 stream for Form controls and uses it if available instead of the Excel 2007 stream. It also now writes both an Excel 2007 and 2010 stream. The Excel 2010 stream is better because it saves the coordinates in device independent units, so controls will look fine when opened in High DPI displays. Excel 2007 on the other hand uses real pixels, which results in different dimensions for the controls when opened in high dpi mode.
- **Files created with NewFile will now not have printer settings, and the locale of them all will be English.** Depending in the Excel version passed to NewFile, FlexCel could add some printer settings to the empty file, and some versions had different locales. Now all locales for files created by NewFile are US English, and there are never printer settings.
- **Support for camera tool (linked images) in xlsx files.** Now "camera tool" pictures will be preserved when saving to xlsx, and converted between xls and xlsx. They will also update when you insert rows or columns. Note that FlexCel won't update camera tool images if the cells change, but they will be updated by Excel when you open the file.
- **New global variables TSmoothingMode.FlexCelDefault and TInterpolationMode.FlexCelDefault.** Those new variables let you decide what is the default antialiasing and interpolation mode used when rendering images.
- **Breaking Change: Autoshapes in xls files are now rendered using the xlsx definition of them if it is stored in the file.** Excel 2007 and newer save the autoshapes in two different places inside an **xls** (not xlsx) file: One that is read by Excel 2003 or older, and the other which is read by Excel 2007 or newer. FlexCel used to read the section of Excel 2003 or older to render the autoshapes, now it is using the section of Excel 2007 or newer. While in general this should improve autoshape rendering, note that this change is potentially breaking if you have files with the correct definition in the xls section and incorrect in the xlsx section. For more information, please see: <https://tmssoftware.com/site/blog.asp?post=347>
- **Recovery mode can now open files with invalid format strings.** Now when `XlsFile.RecoveryMode = true` and the file has invalid format strings, FlexCel will open the file anyway and report the errors in FlexCelTrace.
- **Bug Fix.** In some border cases when opening and saving a file multiple times and adding the same format every time, the format could be added each time instead of detecting it already existed.
- **Bug Fix.** Some autoshapes with holes inside could be rendered as fully filled.
- **Bug Fix.** Improved rendering of custom xls autoshapes
- **Bug Fix.** There could be an error when saving pivot cache slicers or timelines in multiple sheets.
- **Bug Fix.** Some colors in controls or shapes in xls files could be read wrong.
- **Bug Fix.** Improved compatibility when opening invalid xlsx files.

New in v 6.8.1.0 - May 2016

- **Bug Fix.** Some xlsx files manually edited could fail to load in .NET 2, 3.5, pcl or .netcore.
- **Added support for .NET Core RC2.** Now the FlexCel for .NET core version supports (and requires) .NET Core RC2 (.NET Core SDK 1.0 Preview 1)
- **Bug Fix.** Fixed error when exporting images to pdf in UWP 10 apps.

New in v 6.8.0.0 - April 2016

- **Support for Windows 10 Universal Apps.** The dll FlexCelPortable 81 can now be used from Windows 10 Universal apps.
- **Full support of Hyperlinks in autoshapes in xlsx files.** Now hyperlinks in shapes inside xlsx files are fully preserved as they were in xls files. They also will convert between xls and xlsx, and you can change the hyperlinks of the shapes with `xls.SetObjectProperty`. Links are exported to pdf, html and svg.
- **Full support for "Allow users to Edit ranges" in the API.** The new methods `XlsFile.Protection.AddProtectedRange`, `XlsFile.Protection.DeleteProtectedRange`, `XlsFile.Protection.ProtectedRangeCount` and `XlsFile.Protection.ClearProtectedRanges` allow you to read and modify protected ranges. Note that for simple protection you can still just lock or unlock the cells in the cell formatting. APIMate should report now how to enter Protected Ranges too.
- **New `<#Switch>` and `<#IFS>` tags in FlexCel reports.** Those tags behave like the `IFS` and `SWITCH` functions added in Excel 2016 january update. They can eliminate "if chains" and make the expressions simpler. For example `<#ifs(<#value> < 10;<#format cell(red)>><#value> < 20;<#format cell(yellow)>>true;<#format cell(green)>>>`
- **Autosize of chart axis when rendering charts.** Now when exporting xls charts to pdf/html/etc, the axis of the chart will move to fit the data in the axis so it doesn't get cut out.
- **New parameter `convertFormulasToValues` added to `PasteFromXlsClipboardFormat`.** This parameter will allow you to paste the formulas as values from the clipboard. This is useful specially if the formulas you are pasting reference other books, so they won't reference the correct cells when pasted.
- **New parameter `recalcBeforeConverting` added to `ConvertFormulasToValues` and `ConvertExternalNamesToRefErrors`.** This parameter will allow you to convert formulas to values without first having FlexCel recalculating the file (which was the default before). So if your file can't be recalculated by FlexCel because for example it contains links to other files that don't exist anymore, you can still convert the formulas to the latest calculated values.
- **New overload of `FlexCelPdfExport.ExportAllVisibleSheets` taking a filename.** This overload is a shortcut for creating a filestream, calling `BeginExport` on the stream, then calling `ExportAllVisibleSheets` and then calling `EndExport`.

- **Support for ShrinkToFit attribute in cells when exporting to pdf/html/svg/images/printing/previewing.** Now the ShrinkToFit attribute of cells is rendered when exporting, and will show in printing, previewing and exporting.
- **Support for adding horizontal scrollbars with the API.** There is a new property in TSpinProperties which allows to specify if the scrollbar is horizontal. APIMate will report how to do it from a horizontal scrollbar in Excel.
- **<#IF> tag in reports can now omit the false section.** You can now write a tag like <#if(true;hi)> instead of <#if(true;hi)>
- **Better chart rendering for xls files.** Now the labels overflow in a way similar to Excel, and FlexCel calculates the chart axis positions so it won't overflow.
- **The file created by XlsFile.NewFile(n, TExcelFileFormat.v2016) now is in the Excel "January update" format.** The "January update" of Excel 2016 added some new fonts to the themes and changed the build id of a default empty file. Now the empty files that FlexCel generates when you specify v2016 include those new fonts in the themes. The build id had already been updated in a previous FlexCel release.
- **Bug Fix.** When exporting xls bar and column charts with a single data series and "Vary colors per point" = true, FlexCel was not changing the colors on each point.
- **Bug Fix.** When copying sheets with data validations to other files, and the data validations would refer to a list in a different sheet, the data validation would be copied wrong.
- **Bug Fix.** When rendering conditional formats in xls files sometimes the background color could be ignored.
- **Bug Fix.** The constructor of TBlipFill wasn't public.
- **Bug Fix.** TXlsFile.DpiForImages would be ignored in some metafiles.

New in v 6.7.16.0 - March 2016

- **Support for the new functions introduced in the Excel 2016 January Update.** Now FlexCel can recalculate and recognize the 6 new functions introduced in the [Excel 2016 January Update](#) : [TEXTJOIN](#), [CONCAT](#), [IFS](#), [SWITCH](#), [MINIFS](#), [MAXIFS](#).
- **Updated the RecalcVersion for Excel 2016 to the Excel 2016 January Update.** The January update of Excel 2016 changed the RecalcVersion id saved in the xls and xlsx files. This means that xls or xlsx files saved with Excel 2016 "pre-january-update" would ask for saving when opened and closed in Excel 2016 "post January update". Now when you choose the RecalcVersion in FlexCel to be 2016, FlexCel will identify the file as saved by "post-january-update" Excel 2016. This will avoid the save dialog when opening in Excel 2016 with all the updates.

- **New value in TRecalcVersion: "TRecalcVersion.LatestKnownExcelVersion" will identify the file saved by FlexCel as the latest Excel version that FlexCel knows about.** If you set `xls.RecalcVersion` to be `TXlsRecalcVersion.LatestKnownExcelVersion` then FlexCel will identify the file as saved by the latest Excel version it is aware of. Currently this means the files will be identified as saved by Excel 2016 January update. When newer Excel versions appear and FlexCel is updated to support them, then this version will automatically increase to the latest without needing to modify your source code.
- **New property UsedZoom in TOneImgExportInfo.** The property `UsedZoom` will tell you the actual zoom that is going to be used when printing or exporting the sheet. So you now can call `FlexCelImgExport.GetFirstPageExportInfo()` and get the zoom of the pages that will be printed, including the zoom calculated for print to fit if set.
- **Improved compatibility with invalid xls and xlsx files.** Now FlexCel will fix files which have an invalid active sheet stored, and set the active sheet to the first in those cases.
- **Improved compatibility with third party xlsx files.** FlexCel will now understand xlsx files which use absolute references like `A3` in cell value addresses. Note that Excel never writes absolute references in the cell values addresses, but some third parties might. Now you will be able to read those files too.
- **Bug Fix.** COLUMNS DataSet would not work inside a filter.
- **Bug Fix.** When exporting sheets with multiple print ranges, and those print ranges had different zoom (due to having a PrintToFit zoom), FlexCel could raise an error. Now, similar to Excel, it will calculate the smallest zoom needed for all the ranges, and use that in all the print ranges.
- **Bug Fix.** Rendering of images in headers and footers in xlsx files could be wrong if the sizes in the file were in mm.

New in v 6.7.12.0 - February 2016

- **New properties ExcelFile.HeadingRowHeight and ExcelFile.HeadingColWidth.** Those properties allow you to specify the width of the heading column and the height of the heading row when printing headings or exporting them to pdf via FlexCel. The "Custom preview" demo now sets those properties so the sizes are automatic.

New in v 6.7.10.0 - February 2016

- **BugFix.** When using `TFlexCellmgExport.ExportNext` to save to a file without transparency (like JPEG), the background would be black. Now it should be transparent if the format supports Alpha channels, and white otherwise.

New in v 6.7.9.0 - February 2016

- **New static events TUIFont.FontCreating and TUIFont.FontCreated.** Those events allow you to customize the font replacements in your system. For example, if you have Excel files that have a font "MyDeprecatedFont" and you would want to replace it by "MyNewCoolFont" when exporting to pdf, you can use the `FontCreatingEvent` to do so.

You can use the `FontCreated` event to catch fonts where the original wasn't present in the machine and were substituted by the operating system into something else. You can then provide a different substitute font.

New in v 6.7.8.0 - February 2016

- **Experimental support for .NET Core 1.0 and ASP.NET Core 1.0.** There is a new nuget package included which can be used in .net core. As .net core doesn't have a graphics library yet, this package can only deal with xls and xlsx files, and has no graphics capabilities like exporting to pdf or html.
- **New methods `OffsetRelativeFormula` and `RecalcRelativeFormula` in `ExcelFile`.** Those new methods allow you to know the real value of a **relative formula**, such as those returned by **names** and **data validations**. Relative formulas depend on the cell the cursor is, so if the cursor moves the formula changes. As FlexCel doesn't have a cursor, it always returns the formulas considering the cursor at A1. With `OffsetRelativeFormula` you can get how the formula would look like when the cursor is at for example B3, and with `RecalcRelativeFormula` you can recalculate the formula and get the result when the cursor is at B3.
- **Support for quoted column names in reports.** Now you can quote a column name inside a tag in a report, like `<#"db.column) ">` This can be useful if you have column names with for example unbalanced parenthesis. Note that you don't need to quote the name if it has balanced parenthesis.
- **Bug Fix.** In some cases when opening an xls file with existing formats, and calling `AddFormat` for a format already present in the file, FlexCel would fail to realize the format already existed and create a new one. This could lead to having more formats than the number allowed by Excel if you opened a file, added existing formats and save it a lot of times.
- **Bug Fix.** When `<#including>` subreports inside FlexCel Reports with the RC option, empty row formats would be copied to non empty row formats.
- **Bug Fix.** ActiveX controls with a size larger than an `Int32` would raise an Exception when loading.
- **Bug Fix.** Bidirectional reports could fill some wrong cells when using multiple master-details in the rows.
- **Bug Fix.** Xlsx files with autofilters could become invalid if you deleted the range which contained the autofilter.
- **Bug Fix.** `VLookup` and `HLookup` would return a match if you searched for a blank string ("") and the cell was empty. Excel doesn't return a match in those cases, and now FlexCel doesn't either.
- **Bug Fix.** Double bordered lines could render wrong when the zoom was big (about 200% or more)

- **Bug Fix.** Some invalid formulas including more than one "=" sign in a not valid location, like "=1 + =1" didn't throw an Exception when you tried to manually enter them, and would raise the exception later when trying to save. Now FlexCel will report those formulas as invalid when you try to enter them.

New in v 6.7.3.0 - January 2016

- **New JOIN and UNION commands for reports.** Those commands are written in the config sheet and allow you to either *JOIN* the columns of multiple tables into a single table, or to do an *UNION* of the rows of multiple tables into a single one. See the new "Join and Union" demo.
- **Improved bidirectional reports.** Now bidirectional reports can work with rows in master detail and they also will delete empty column bands if none of the columns has records.
- **Improved preservation of timelines in xlsx.** Timelines are a feature introduced in Excel 2013, which allow you to graphically navigate a timeline in a data source. Now FlexCel should preserve timelines for pivot tables.
- **Bug Fix.** Fixed order of records specific for Excel 2010 to workaround a bug in Excel 2010. Some very complex files could raise an error when opened in Excel 2010, even when they were correct by the xlsx spec.

New in v 6.7.2.0 - December 2015

- **Copy to the clipboard now supports html.** Now there is an extra option in the formats to be copied to the clipboard: In addition to native xls (best for copying from one spreadsheet to another) and text (for apps that don't understand anything else), now you can also copy as html, which gives the best results when pasting a spreadsheet in Microsoft Word or PowerPoint. You can copy to html either using `XlsFile.CopyToClipboardFormat` or `FlexCelHtmlExport.ExportToClipboardFormat`.
- **Bidirectional Reports.** Now you can create ranges in shape of a cross that expand to the right and down at the same time. While you could do this before by splitting one of the ranges in 3, now you can directly intersect the ranges and get the correct result. Take a look at the new Bidirectional Reports demo and the documentation in the report designer guide.
- **Changed default fallback fonts in pdf.** Windows 10 doesn't come with MS Mincho or MS Gothic installed by default (you need to manually install the language packs to get the fonts). So now FlexCel looks for both MS Mincho/Gothic (for windows older than 10), and YuMincho/Gothic for Windows 10.
- **The tags <#List>, <#DbValue> and <#Aggregate> can now work inside nested Array/Linq datasets.** Now when you have a master detail relationship where the detail is a property of the master, FlexCel can find the master dataset for the <#List>, <#DbValue> and <#Aggregate> even when they are not added with `AddTable`.

- **New property XlsFile.DocumentProperties.PreserveModifiedDate.** FlexCel by default sets the modified date of the files it saves to the date when the file was saved. But if you want to change this date to an arbitrary date, then you can set PreserveModifiedDate to true.
- **FlexCel will now set the creation and modification date in xls files too.** Now Creation and Modification dates are stored in xls files, same as they already were in xlsx.
- **FlexCel will now allow you to set the file creator for xlsx files.** By default, files created by FlexCel are identified as created by FlexCel in the document properties. Now you can change the application creator by writing


```
xls.DocumentProperties.SetStandardProperty(TPropertyId.NameOfCreatingApplication, "SomeNewCreator")
```
- **Bug fix.** LastModifiedDateTime wasn't returned correctly for xlsx files.
- **Bug fix.** Macros converted from xls files to xlsx could fail to open in Excel 2016 in some border cases.
- **Improved Getting Started document.** Now GettingStarted shows actual code examples on how to do simple tasks and contains links to all documentation.

New in v 6.7.1.0 - November 2015

- **Support for new Excel 2016 features.** While old FlexCel versions will still work fine with Excel 2016 (as expected), FlexCel now provides support for new extra features in Excel 2016. Now XlsFile.NewFile allows to create files like Excel 2016 creates by default. Also XlsFile.RecalcVersion has a 2016 option to tag your files as created by Excel 2016 so Excel 2016 doesn't ask for saving when closing them.
- **Improved support for DataValidations that have lists with cells from other sheets.** DataValidations with lists of cells from other lists were introduced in Excel 2010, and while FlexCel preserved them, it wouldn't modify them when inserting or deleting ranges. They wouldn't either be reported by the API. Now they are modified and also reported by the API, just like all the other data validations.
- **Slicers for Pivot Tables are now preserved in xlsx.** Now FlexCel will preserve the slicers for pivot tables present in xlsx files. This is a feature available only in Excel 2010 or newer, so you won't see them in older Excel versions, but the generated files will still open without errors.
- **Excel 2010 equations are now preserved in xlsx.** Now FlexCel will preserve the new equations in Excel 2010 (Ribbon->Insert->Equation)
- **Center across selection cells are now exported to html.** Now html export will export cells marked as "center across selection", same as exporting to pdf or other exports already did.
- **Improved exporting of superscripts and subscripts to html.** Now superscripts and subscripts are exported better to html files.

- **Full support for formulas attached to textboxes or autoshapes.** Now FlexCel will preserve and convert between xls and xlsx textboxes or shapes which have their text linked to a formula. If you modify the linked cell, the text in the textbox will change.
- **New methods XlsFile.SheetID and XlsFile.GetSheetIndexFromID.** Those new methods can be used to identify a sheet in a FlexCel session and get it back later. Note that as this ID is not saved in the file, it will change every time you load a new file and so it can only be used in a single session.
- **New static property "UseLegacyLookup" in FlexCelReport.** If you set this property to true, FlexCel will use DataViews to do lookups in DataSets instead of the new and faster internal lookup, Set this property to true only if your existing reports rely in bugs in the DataView implementation.
- **Data validations entered manually in xls files could fail to work when opened in Excel.** In some border cases, Excel would report all values as invalid for a data validation entered with FlexCel, even if the values were valid. This only applied to xls files.
- **Now FlexCel can open xlsx files with images with the wrong image type.** If an xlsx file now contains for example a png but it is declared as jpg, now FlexCel will open it as a png anyway. This will only happen with corrupt files or files generated by incorrect third-party products.
- **Error when deleting rows in a pivot table.** When deleting rows in a pivot table in an xlsx file, the rows could go negative creating invalid files.
- **Improved compatibility with third-party tools.** Workaround for some tags not understood by other third-party tools, and now we can read files missing some required records.

New in v 6.7.0.0 - September 2015

- **Support for opening xls versions from 2 to 4.** As FlexCel already supported xls 5 and up and Excel 1 doesn't exist for Windows, this completes the support for all versions of xls. While xls versions from 2 to 4 aren't in wide use, they are still used by other third-party libraries.
- **Enhanced High DPI Support in FlexCelPreview.** Now FlexCelPreview supports High DPI in Windows, besides iOS or OSX as it already did.
- **Breaking Change: Property Resolution in FlexCelPreview has been removed.** The property Resolution of FlexCelPreview has been removed because now FlexCelPreview automatically adjusts to the resolution of the monitor.
- **Full support for background images in a sheet.** XlsFile adds two new methods to deal with background images in a sheet: SetSheetBackground and GetSheetBackground. Background images are now converted between xls and xlsx. ApiMate will also report the code to add a background image to a sheet. A new property ExportSheetBackgroundImages allows you to print or export the background images. (note that Excel never prints the background images, so this property is false by default)

- **Full support for manipulating Custom XML parts with XlsFile.** The new methods `CustomXmlPartCount`, `AddCustomXmlPart`, `GetCustomXmlPart` and `RemoveCustomXmlPart` in `XlsFile` allow for reading and writing the custom xml files of an xlsx files, as explained here: <https://msdn.microsoft.com/en-us/library/bb608618.aspx> ApiMate will now show how to enter custom xml parts in an xlsx file.
- **New property for PDF files: InitialZoomAndView.** The new `InitialZoomAndView` property allows you to specify the initial page and zoom when opening the document.
- **New property for PDF files: PageLayoutDisplay.** The new `PageLayoutDisplay` property allows you to specify if to display one or two pages, and continuous scrolling or one page at a time when opening the document.
- **Two new modes for PDF files PageLayout.** Now generated PDF files can use a `PageLayout` of `TPageLayout.OptionalContent` to show the optional content panel, or `TPageLayout.AttachmentPanel` to show the attachments panel.
- **New property ScreenScaling in XlsFile.** This new property allows you to workaround Excel bugs when working in high dpi displays. For more information read <https://www.tmssoftware.com/site/blog.asp?post=311>
- **Better handling of stored numbers in xlsx.** Now numbers are saved in xlsx with a roundtrip format, which ensures the number we write in the file is exactly the same number that will be read.
- **Ability to <#insert> empty names in reports.** Now when you use the `<#include>` tag in a report, you can leave the name to include empty. This will mean to insert all the used range in the active sheet.
- **New overload for XlsFile.DeleteRange.** There is a new option for `XlsFile.DeleteRange`, which will clear the cells but not the formats on it: It will behave similar to when you press "Delete" in a range of cells in Excel.
- **New property ExportEmptyBands in FlexCelReport.** `ExportEmptyBands` replaces the existing `ExportEmptyRanges` property which has been deprecated. It allows you to choose between 3 possibilities when the data table has 0 records: Delete the range and move cells up, clear the data and format of the range, or clear only the data.
- **Bug Fix.** Now FlexCel will make sure the xml declaration in the custom xml parts added with `AddCustomXmlPart` have the same encoding as the encoding being used to store the file.
- **Bug Fix.** Xlsx files with external formulas referring to other sheets starting with a number weren't quoted, and Excel would report an error when opening those files.
- **Bug Fix.** FlexCel would fail to load files with formulas which pointed to tables in other files with the new table formula syntax.
- **Breaking Change: Improved lookup tag in reports.** The `<#lookup>` tag in reports has been rewritten to be faster and behave better. IF you are defining your own `VirtualDataSets` and overriding the `Lookup` function you might need to rewrite it, as parameters changed. But with the new base lookup implementation that is now available for all, you might just remove the override and use the base.
- **Bug Fix.** Subtotal function could recalculate wrong in border cases.

- **SPLIT Datasets in Reports can now be used as datasets for sheets.** This allows you to overflow a report into multiple sheets. When the data in a sheets reaches the maximum of the split, it will continue in a different sheet. A new sample "Overflow sheets" shows how to do it.
- **Copy to clipboard wasn't working in Excel 2013.** We modified the clipboard format so now it is working.
- **Bug Fix.** When inserting or deleting columns, array formulas located in other sheets might not update to take in account those changed rows or columns.
- **Bug Fix.** Sometimes when moving a range array formulas which pointed to that range might fail to update.
- **Bug Fix.** Some functions with array arguments could not be calculated correctly when the formula was not an array formula.
- **Bug Fix.** The lookup tag introduced in 6.6.32 could fail if the lookup value was a tag in the template
- **Bug Fix.** The functions SumIfs, Averagelfs and CountIfs could give wrong results in some cases.
- **Bug Fix.** When rendering a chart with an image inside, there could be an exception.
- **Bug Fix.** Images inside charts with negative coordinates weren't rendered.
- **Bug Fix.** Now scatter charts behave like Excel, and if any of the x-axis values is a string, it will be rendered as a line chart instead.
- **Bug Fix.** XlsFile.SetAutoRowHeighth wouldn't work if the row was empty.
- **Bug Fix.** Chart rendering now renders charts where all values are 0.
- **Bug Fix.** Chart rendering now respects the label positions next to axis, high and low.
- **Bug Fix.** ExportEmptyBands introduced in 6.6.25 wouldn't work in detail reports.
- **Bug Fix.** In some cases when generating reports and exporting them to pdf directly without saving them as xls/x, there could be a range check error.
- **Bug Fix.** Tabs inside text in autosshapes now will render as 8 spaces. (note that we don't use the tab definitions from the autoshape, so this is an approximation)
- **Bug Fix.** When exporting to bitmaps, the bitmaps where a little bigger than the page size
- **Bug Fix.** Reports using LINQ could raise an Exception in some cases with null values.
- **Improved compatibility with invalid xlsx files generated by third parties.** FlexCel can now read some invalid formulas written in xlsx by other third-party products.

New in v 6.6.23.0 - April 2015

- **Visual Studio 2015 and .NET 4.6 support.** Support has been added for the latest Visual Studio and .NET betas.

- **Fix for the latest Xamarin version.** Xamarin changed how encodings behave when they don't exist: Before they used to raise an Exception and now they return null. This broke the fallback support in older FlexCel versions, and has been fixed now.
- **Bug Fix.** There could be an error when rendering error bars in charts and there were missing values.

New in v 6.6.22.0 - April 2015

- **New property SheetView in XlsFile allows you to set the page view mode and zoom for each mode.** Now you can see or set the page view mode in a sheet (normal, page layout or page break preview). You can also specify the zoom for each of the modes. As usual, APIMate will show you the syntax.
- **New property LinksInNewWindow for FlexCelHtmlExport and FlexCelSVGExport.** When you set LinksInNewWindow to true, both FlexCelHtmlExport and FlexCelSVGExport will export the hyperlinks in the file to open in a new window.
- **Links to local files and to current workbook are now exported in TFlexCelHtmlExport.** Now links to local files or other cells in the current workbook are exported to html. This allows for navigating inside a file. Links in the current workbook work even when exporting to different tabs.
- **Breaking Change: XlsFile.AddImage(row, col, TUIImage) now takes in account the declared image dpi.** Now if you are calling AddImage without specifying the dimensions, FlexCel will use the dimensions corrected by the dpi declared by the image. This is the same way Excel works. In previous FlexCel versions we always assumed a 96dpi image.
- **Rendering of error bars in xls charts.** Now when exporting to pdf/html/etc, FlexCel will draw error bars. All modes (StdErr, StdDev, fixed, percent, custom) are supported.
- **Improved display of line charts.** Now colors and sizes of lines in xls charts will be read from the new xlsx records on it if they exist. This leads to a more faithful rendering, because the xlsx records have extra information, like for example a line width that isn't restricted to 4 sizes.
- **TXlsNamedRange.GetRanges is now public and documented.** GetRanges will return an array with the ranges composing a name. So if you for example have a name with the range "1:1, A:A", GetRanges will return an array with 1:1 and A:A. This method can be used to parse the PRINT_TITLES range.
- **Improved display of markers in charts.** Now markers in charts render much more alike Excel 2013, with the new options for images, etc.
- **Bug fix.** XlsFile.FillPageHeaderOrFooter could return an extra "&" character at the end in some cases.

New in v 6.6.21.0 - January 2015

- **Better Xamarin package for osx.** The Xamarin package used to copy the files in the "osx" folder, now it copies to "mac" to comply with the new naming.

- **New UsePrintScale property in FlexCelHtmlExport.** If you set the new property `FlexCelHtml.UsePrintScale` to true, then the exported html will use the scaling of the printed sheet instead of being exported at 100% zoom.
- **Bug fix.** Some JPEG images weren't recognized as such.
- **Bug fix.** Reports might not read expression values when tags had a default value, like `<#value;0>`
- **Bug fix.** Sometimes FlexCel could fail to load an xlsx file with different images with the same extension but different case (like `image1.png` and `image2.PNG`)
- **New parameters in FlexCelPdfExport.AfterGeneratePage and BeforeGeneratePage events.** The new parameters are the `XlsFile` being exported, the `FlexCelPdfExport` component doing the export, and the current sheet.
- **Improved RecoveryMode.** Now FlexCel can recover more types of wrong files when `RecoveryMode` is true.
- **When drawing xls charts, we now use the options for not plotting empty cells.** This option was introduced in Excel 2007, and FlexCel was ignoring it. Now if you choose not to ignore hidden rows or columns, the chart will render as expected.
- **New method `XlsFile.RemoveUserDefinedFunction`.** `XlsFile.RemoveUserDefinedFunction` allows to unregister a previously registered UDF for recalculation.
- **Breaking Change: Now when drawing chart labels that have N/A! error as result, FlexCel won't draw them.** Excel 2003 or older is different in the way it draws #NA! errors in chart labels from Excel 2007 or newer. In older Excel versions, the label would just draw as #NA!. In newer Excel versions, it doesn't draw. To be consistent with more modern Excel versions, now FlexCel won't draw them either when exporting to pdf or html.
- **FlexCelReport can use also arrays besides `IEnumerable<T>` for detail bands.** When using `IEnumerable` as data source, now you can use a fields which are `array<T>` instead of an `IEnumerable<T>` as detail tables.
- **New `NOGRAPHICS` define.** If you define `NOGRAPHICS` and undefine `GDIPLUS` in the FlexCel project properties, you'll get a build which doesn't depend on any drawing engine.
- **New `DOTNETZIP` define.** If you define `DOTNETZIP` in the FlexCel project properties and add a reference to it, you'll get a build which uses `dotnetzip` instead of `System.IO.Compression`.
- **Bug fix.** In some cases, when pasting a file with autofilters from Excel you could get a range error. This is because Excel copies the filter in its totality and part of the filter might be outside the range copied. Now FlexCel will resize the autofilter if it extends beyond the copied range.

New in v 6.6.11.0 - December 2014

- **Support for recalculating 31 new functions introduced in Excel 2013.** Support has been added for: DAYS, ISOWEEKNUM, BITAND, BITOR, BITXOR, BITLSHIFT, BITRSHIFT, PDURATION, RRI, ISFORMULA, SHEET, SHEETS, IFNA, XOR, FORMULATEXT, COT, ACOT, COTH, ACOth, CSC, CSCH, SEC, SECH, ARABIC, BASE, DECIMAL, COMBINA, PERMUTATIONA, MUNIT, UNICHAR, UNICODE
- **Subtotal command in XlsFile.** There is a new command `xls.SubTotal(...)` which works the same as the command "Subtotal" in the Excel Ribbon, "Data" tab. While you shouldn't use this when creating new files, it can be useful for formatting old files. For new files, it is best to just create the subtotals in place.
- **New option "ExcelLike" in XlsFile.Sort.** Now when doing a `XlsFile.Sort` command you can choose between the correct way to handle formulas (this was the only option before) or the "Excel" way of handling formulas, where references are not updated when a row is moved of place in the sort. The ExcelLike mode doesn't adapt formulas that reference those rows, but it can be much faster for tens of thousands of records.
- **New methods `IsRowMarkedForAutofit` and `IsColMarkedForAutofit` in XlsFile.** The new methods will return true if a row or column was marked for autofit.
- **New property `ExcelFile.AllowEnteringUnknownFunctionsAndNames`.** If you set this property to true, you will be able to enter unknown functions inside formulas, like `"=SomeText()"`. Excel will show the result as a #NAME! error. When this property is false (the default) FlexCel will raise an Exception if the name is not know, which is better to detect misspells.
- **New properties `XlsFile.RecalcVersion` and `FlexCelReport.RecalcVersion`.** This new properties allow you to specify the Excel version that last calculated the file. If you set it to for Example Excel 2010, any Excel newer than Excel 2010 will recalculate the file on open, and ask for saving changes when you close the file. Excel 2010 or older won't recalculate the file on open. If you want every version of Excel to recalculate on open set this property to `AlwaysRecalc` (the default). Look at the API developers guide for more information.
- **Breaking Change: `XlsFile.RecalcForced` and `FlexCelReport.RecalcForced` properties have been removed.** `RecalcForced` used a way to make files recalculate on open which has been deprecated in newer versions of Excel, and will cause validation errors with the generated files. For this reason, `RecalcForced` wasn't doing anything in the last couple of years. Look at the new `RecalcVersion` property if you were using `RecalcForced` and want new not deprecated way to create files which Excel will recalculate on open.
- **Improved Xamarin Unified API support.** Changed the unified API support to compile in the latest beta.
- **Included reports can now reference the formats of the parent report.** Now an included report can reference the formats of the parent report, same way as it can reference the expressions.
- **New overloads of `XlsFile.GetObjectProperties` and `XlsFile.GetObjectAnchor` that take an object path.** This new methods allow you to access the properties and anchor of an object by specifying its name, as in `Xls.GetObjectAnchor(-1, "@MyObject")`

- **Autofitting columns with 90 degree rotation would work always as if the column had "Wrap text" enabled.** When autofitting columns which had a rotation of 90 degrees, FlexCel would always try to wrap the text so it fitted in many lines, even if the cell wasn't set to wrap. Now it will only do this if the cell has "Wrap text" on.
- **Breaking Change: Removed Xamarin Android 2.2 support.** As Froyo (2.2) is now deprecated, we've removed this support in order to avoid deprecated warnings. Now minimum supported is 2.3 (Gingerbread).
- **Bug Fix.** When doing reports with Linq, aggregating a double field could raise Exceptions.
- **Pivot tables in xlsx are now copied when you copy sheets.** Now if you InsertAndCopySheet(...) a sheet with a pivot table from an xlsx file, the table will be copied. (pivot tables in xls were already copied)
- **Unknown names in formulas now return #NAME! instead of #NA!.** Now when a formula references a name that doesn't exist, FlexCel will return #NAME! as the formula result, instead of #NA! as it used to do.
- **Bug fix.** When setting the text of an object using SetObjectText the font might not be preserved.
- **Bug fix.** When changing the font in HtmlFont event in TFlexCelHtmlExport there could be an Exception.
- **Bug fix.** RoundUP and RoundDown functions could return the same number and not the rounded up number in some cases when the number of digits was negative.
- **Bug Fix.** Rendering some files with thousands of hidden columns could take too long.
- **Hidden rows could sometimes count when finding the maximum used column in the sheet.** When printing or exporting an xls/x file, a hidden row with columns outside the printing range could in some cases cause the maximum column to be that in the hidden row, which wouldn't be printed.
- **Improved compatibility with invalid xls files.** Now FlexCel can read some more invalid xls files created by third parties.
- **Sheet names aren't always quoted when returning formula text.** In older FlexCel versions, the sheet was always quoted in formulas. So if you retrieved for example the formula in A1, it could be 'Sheet1!A2. Now we quote the sheet only if needed, same as Excel does. So we would return Sheet1!A2 instead.
- **New convenience constructor for XlsFile which takes a Stream.** Now you can create an XlsFile and open a stream in a single operation, without having to first create the XlsFile and then call xls.Open.
- **Improved error message when opening files with 0 bytes.** Now when opening files with 0 bytes or streams with the position at the end, FlexCel will say a clear message instead of saying that the file format isn't Excel or newer.

- **New properties FlxConsts.MaxRowCount and FlxConsts.MaxColCount.** Those properties return `FlxConsts.Max_Rows + 1` and `FlxConsts.Max_Columns + 1` respectively. `Max_Rows` and `Max_Columns` were zero based, so for example `Max_Rows` return 65535 for xls and not 65536 which is the row count. The new properties return the one-based maximum, which makes it simpler to work in the one-based FlexCel API.

New in v 6.6.2.0 - October 2014

- **Better rendering of text in rotated shapes.** In Excel 2003 or older, text in rotated shapes was shown without rotation and that's how FlexCel would show them. Since Excel 2007 the text can rotate with the shape, using an undocumented record in xls. Now FlexCel can read it and will honor that setting when converting to pdf/html/svg/printing/etc.
- **Bug Fix.** A local link in a pdf to a page that wasn't exported could cause an Exception.
- **Bug Fix.** Exporting "Center on selection" cells could be too slow in border cases.
- **Bug Fix.** `XlsFile.SetCommentRow` could set the wrong comment in some cases.

New in v 6.6.1.0 - October 2014

- **Generic reports using `<#table.*>` can now use user defined functions.** Now you can apply a user defined function to a `<#table.*>` tag.
- **Generic reports using `<#table.*>` can now reference fixed fields in the table.** Now you can mix `<#table.field>` with `<#table.*>` in the same cell.
- **Better compatibility with files created by third parties.** Now FlexCel will load invalid xlsx files with repeated comments.
- **Bug Fix.** Generated xlsx files could be invalid when removing frozen panes from an existing file.

New in v 6.6.0.0 - October 2014

- **Breaking Change: Now the result of `ShapeOptions.Text` is a `TDrawingRichString` instead of a `TRichString`.** In order to allow more customizability in the text of shapes and objects, we had to move the text property from a `TRichString` (which is used for cells, and in xls was also used for objects) to a `TDrawingRichString` (Which in xlsx offers more possibilities to customize the text). As there is an automatic conversion from a `TDrawingRichString` to a `TRichString`, most code will just keep working. But there might be some cases (like functions where you pass a var parameter) where you will need to change the types from `TRichString` to `TDrawingRichString` in order to compile.
- **Better support for preserving autoshape text in xlsx.** Now when you change the text of an autoshape in xlsx, the existing properties of the text will be preserved.
- **Support for reading and writing a cell's text direction (RTL, LTR or Context).** Now you can specify the text direction in a cell, and APIMate will show you how to do it. The FlexCel rendering engine also now supports better RTL code (still without providing official RTL support, it is better in this version and usable in most cases)

- **Support reading the number of horizontal and vertical page breaks in a sheet.** Two new properties: `XlsFile.HPageBreakCount` and `XlsFile.VPageBreakCount` return the count of page breaks in a sheet.
- **Bug Fix.** `XlsFile.LastFormattedCol` returned the last formatted column - 1. Now it is returning the correct number.
- **Bug Fix.** Rendered xls charts could show an extra line in some corner cases with missing data.
- **Bug Fix.** TOPN datatables didn't inherit their relationships with master datasets.
- **Bug Fix.** Macro references in buttons could be copied wrong when copying sheets.
- **Bug fix.** When copying a range of cells to another sheet which included formulas introduced in Excel 2007 or newer there could be an error when saving as xls.
- **Bug Fix.** FlexCel enforced a maximum of 1023 manual page breaks for xls but not for xlsx. Now We also check that the generated xlsx files don't have more than 1023 manual page breaks, since that would crash Excel.

New in v 6.5.0.0 - September 2014

- **PDF/A support.** FlexCel can now export to PDF/A-1, PDF/A-2 and PDF/A-3 files. A new property `FlexCelPdfExport.PdfType` determines if the file is a standard PDF or the version of PDF/A.
- **Xamarin Unified API Support.** FlexCel now includes two new dlls compiled against `Xamarin.Mac.dll` and `Xamarin.iOS.dll` instead of `XamMac.dll` and `monotouch.dll`, in order to support the new Unified API (<http://developer.xamarin.com/guides/cross-platform/macios/newstyle/>) This means you can now compile 64 bit iOS and OSX applications with FlexCel.
- **Breaking Change: Generated PDF files are now tagged by default.** The files generated by FlexCel are now tagged by default, as tagging is an accessibility requirement. Tagged PDF files are bigger than normal files so in order to try to get smaller files FlexCel uses now features available only in Acrobat 7 or newer. To go back to generating untagged files you can set `FlexCelPdfExport.TaggedPdf = false`. To go back to creating files compatible with Acrobat 5 or newer, set `FlexCelPdfExport.PdfVersion = TPdfVersion.v14`
- **Breaking Change: Generated PDF files are now compatible with Acrobat 7 or newer.** In order to reduce the size of the tagged pdf files that FlexCel now creates by default, FlexCel now generates files that need Acrobat 7 or newer to open. To go back to creating files compatible with Acrobat 5 or newer, set `FlexCelPdfExport.PdfVersion = TPdfVersion.v14`. Note that as PDF/A-1 requires compatibility with Acrobat 5 or newer, when exporting PDF/A-1 FlexCel will use v14 automatically. PDF/A-2 and 3 don't require v14, so it isn't used by default for those formats.
- **Breaking Change: Generated PDF files now embed the fonts by default.** Now the default value of `FontEmbed` in `FlexCelPdfExport` and `PdfWriter` is `TFontEmbed.Embed`. While this will create slightly bigger files, they will show fine everywhere, including mobile devices which might not have the fonts. You can revert to the old behavior by changing `FontEmbed` to be `TFontEmbed.None`.

- **Breaking Change: FlexCel will throw an Exception when trying to embed a font that doesn't have a license allowing embedding.** FlexCel will now check that the embedded fonts in PDF have a license that allows embedding. You can revert to the old behavior by setting `UnlicensedFontAction = TUnlicensedFontAction.Ignore`, in case you have an agreement with the Font author. You can also set `UnlicensedFontAction = TUnlicensedFontAction.Replace` to replace the unlicensed fonts with a fallback font. FlexCelTrace will alert when replacing or ignoring a font that is not licensed.
- **Ability to embed files inside the PDF.** Now you can embed arbitrary files inside the pdf. This allows for example to ship the original xls/x file inside the pdf. This is supported also in PDF/A-3.
- **Ability to set the language of the PDF files.** You can now set a `FlexCelPdfExport.Properties.Language` to specify the language of the generated PDF file. Note that the language will be used by text-to-speech engines to read text out loud, so it is recommended to set this property.
- **PDF properties are now saved in XMP format.** PDF properties (like Author, Title, etc.) are now saved in XMP xml format besides the PDF format. XMP is a requirement for PDF/A, and allows other tools that don't understand PDF to read the metadata. Note that the files generated by FlexCel will be now a little bigger due to this metadata, because it can't be compressed.
- **Ability to embed a Color Profile inside the generated pdf files.** You can now set a `FlexCelPdfExport.EmbedColorProfile` property to embed a color profile in the generated files. Note that as a color profile isn't required and it increases the size of the generated files, this option is false by default. But as it is required by PDF/A, a color profile will be embedded in PDF/A files.
- **Breaking Change: Now if you don't specify properties for pdf files in FlexCelPdfExport (like Author, Title, etc.), those will be read from the Excel file being exported.** If you want to revert to the old behavior, you can set `UseExcelProperties = false` in `FlexCelPdfExport`.
- **New structure StandardMimeType returns the mime types for xls, xlsx, xlsxm, pdf, etc.** You can use `StandardMimeType` where you need to specify a mime type for a file generated with FlexCel, instead of having to manually search for the type.
- **Support for <#DBValue> tag in LINQ Reports.** DBValue used to work only with datasets, now you can use it also with LINQ data providers like arrays or lists.
- **Breaking Change: VirtualDataTable doesn't have the MoveToRecord virtual method anymore, and instead it has a new GetValue(row, column).** In order to provide an efficient support for dbvalue in LINQ reports, we needed to change the way to move to a random record. So we don't use `MoveToRecord` anymore and use `GetValue(row, column)` instead. This change is unlikely to affect you, unless you are writing your own `VirtualDataTable` descendent. If you are, then the compiler will point at the missing method implementation.
- **Improved Search and Replace.** Now FlexCel preserves better the format of the cells being replaced. A new overload of `XlsFile.Replace` allows you to specify the format or the values of the replaced cells in a cell by cell basis.

- **Support for entering names referring to other files using Excel notation.** A normal reference to another file has the filename inside brackets, like "[file1.xlsx]Sheet1!A1". But in the case of global names, Excel uses the notation "file1.xlsx!name1", without brackets, which makes it impossible to know if you are entering a name reference to another file (file1.xlsx) or a name reference to the same file, in a sheet named file1.xlsx. FlexCel didn't allow this way to specify the names, and it used to ask for brackets always so you would have to write [file1.xlsx]!name1 to enter the name. Now you can use the same notation as Excel, and FlexCel will allow it as long as you setup a TWorkbook before which includes file1.xlsx.
- **Support for format strings that specify fractions.** Now when using a format string like "??/??" the numbers will be displayed as fractions. For example 0.75 will show as 3/4. All Excel formats for fractions are fully supported.
- **New constructor for XlsFile allows to specify the Excel version in one step.** Now you can create a new file in for example Excel 2010 file format by writing `XlsFile xls = new XlsFile(1, TExcelFileFormat.v2010, true);` in C# or `xls := XlsFile.Create(1, TExcelFileFormat.v2010, true);` in Delphi.
- **New enumeration TExcelFileFormat.v2013.** We now provide a specific `TExcelFileFormat.v2013` enumeration to create Excel2013 files.
- **iOS and OS/X previewer compatibility improved.** Some xlsx files generated by FlexCel that wouldn't show in iOS/OSX previewer will display now. Xlsx charts now update their caches so the previewer will show them correctly.
- **TFlxApplyFont as a new StyleEx property that allows for fine control of which styles are applied.** Before this release you could only apply the full style of the font or nothing by changing the Style property. Now you can specify individual styles like bold or italics by changing the StyleEx property.
- **XlsFile.Sort does a stable sort.** Now when you sort a range of cells, order will be preserved for items with the same values.
- **Bug Fix.** Local named ranges could lose their sheet when inserting sheets from other file.
- **Shapes inside charts are now preserved in xlsx files..** Now xlsx charts will preserve the shapes inside.
- **Ability to preserve modification date in xlsx files.** By default, FlexCel will set the modification date to the date the file was saved. But if you are modifying an existing file and want to preserve the original creation date, you can now do it by setting `XlsFile.DocumentProperties.PreserveCreationDate` to true.
- **Better support for Excel 4.0 macro sheets.** Files with Excel 4.0 macros should load better.
- **Bug Fix.** `XlsFile.Replace` might not keep existing cell formats when replacing dates.
- **Bug Fix.** `Chart.DeleteSeries` could break the format of the remaining series when called in a serie at the middle of the chart.
- **Bug Fix.** There could be an exception when deleting some ranges of cells with hyperlinks.

- **Bug Fix.** Negative dates when in 1904 mode used to display as ####. Now they display as in Excel (see <http://support.microsoft.com/kb/182247>). Note that this is not a logical way to display dates, that is -1 doesn't mean 12/31/1903, but it means "-1/2/1904". Negative dates actually increase as the number get smaller.
- **Support for UTF16 surrogates when exporting to pdf.** Now when exporting to pdf, FlexCel will correctly display UTF16 surrogates. FlexCel already was surrogate-aware in the rest of the codebase.
- **Support for space (" ") named styles.** While Excel won't let you enter a cell style named " ", it will allow you to use it if you manually edit an xlsx file and create it there. To be able to deal with those files, FlexCel will now support reading and writing styles named with a space.
- **Now when adding controls with linked cells, the linked cells will be modified to match the initial value of the control.** Now when adding comboboxes, listboxes or checkboxes linked to a cell, the cell will be modified to match. Note that the change applies to newly created objects, if you change the value of an existing control, the linked cell was always updated in all FlexCel versions that supported changing control states.
- **Bug Fix.** Some xlsx files with charts could enter an infinite loop when loading.
- **Bug Fix.** When replacing rich strings, the rtf runs could be wrong in border cases.

New in v 6.3.1 - May 2014

- **Improved image rendering.** Some files created by third parties could display the images in the wrong position.

New in v 6.3.0.0 - April 2014

- **Windows Phone 8.1 and Windows Store 8.1 Support.** Now FlexCel includes a Portable Class Library which will let you create Windows Phone 8.1 and Windows Store 8.1 apps. Note that in order to see it, you need Visual Studio 2013 Update 2 (currently in RC with a go-live license)
- **SVG Exporting.** A new component, FlexCelSVGExport allows to export xls/x files to SVG. A new example "Export SVG" is included too. SVG files are now supported by all major browsers, desktop and mobile, so you can use them when exporting to html to have resolution independent images.
- **HTML 5 Exporting.** Now FlexCelHtmlExport can export to HTML5/CSS3 as one of the options. When exporting to HTML5 some new capabilities are available.
- **Improved support for document properties.** Full support for setting standard or custom properties in xlsx files. (xls is still read only). A new method `XlsFile.DocumentProperties.RemoveAllProperties` will allow you to remove all properties in the xls or xlsx files, so you can be sure it doesn't contain unwanted information. A new method `XlsFile.DocumentProperties.GetUsedStandardProperties` will return a list of the used standard properties in the file. A new method `XlsFile.DocumentProperties.SetStandardProperty` allows to modify standard properties in

xlsx. New methods `SetCustomProperty`, `GetCustomProperty` and `GetAllCustomProperties` allow to manage custom properties. Document properties will be preserved when opening xls or xlsx files and saving as xlsx, or when opening and saving as xls. They won't be preserved when opening xlsx and saving as xls.

- **Embed images in HTML files.** A new property: `FlexCelHtmlExport.EmbedImages` allows to embed the images as Data Uris (<http://tools.ietf.org/html/rfc2397>) inside the html file. When setting this property to true the generated html file can be self contained, and you don't have to deal with managing the external images.
- **SVG images in HTML files.** Now `FlexCelHtmlExport.SavedImagesFormat` allows a new possibility: `THtmlImageFormat.Svg`. SVG is a standard vector image format supported by most modern browsers, and using svg means that high-dpi/retina devices can zoom smoothly in the vector assets of the file, like charts or autoshapes.
- **TExcelChart.RemoveLegend method.** A new method in `TExcelChart` allows to remove a legend. Note that currently this only works in xls charts, not xlsx.
- **Improved chart rendering.** Now charts will have the frame box with rounded corners if you specify so in Excel.
- **Improved performance in VLookup.** The implementation of the function `VLookup` is now much faster when searching in an unsorted range.
- **Improved performance in pdf/html exporting.** Many optimizations in the rendering engine.
- **Improved APIMate.** `APIMate` now shows how to autofit a comment box to the text size, and also how to set properties in xlsx.
- **New property DisableSQLValidation in FlexCelReport.** Now `FlexCelReport` has a new property "DisableSQLValidation" which will allow to send arbitrary SQL commands from the config sheet. It will allow for example to execute stored procedures in SQL Server. If you decide to disable the SQL validation, please take a look at the remarks in `DisableSQLValidation` documentation.
- **Bug fix.** Some cells with automatic background colors in xls files could be saved with the wrong color when converting to xlsx.
- **Bug fix.** `XlsFile.Find` could keep returning the same values in corner cases.
- **Bug fix.** When Inserting an empty sheet locally defined ranges wouldn't update the sheet where they were defined. This happened only with empty sheets, if you inserted a sheet with data it would work fine.
- **Bug fix.** `FlexCelPreview` could show text as underlined in some third-party generated xlsx files.

New in v 6.2.1.0 - March 2014

- **Improved default font in Headers and footers.** In previous versions if no font was specified for the headers or footers, FlexCel would default to Arial. Now it defaults to the normal font in the file.

- **iOS and Android pdf encoding handling.** Now FlexCel will create pdf files without using Win1252 encoding, which isn't included by default in Xamarin for iOS or Android.

New in v 6.2.0.0 - February 2014

- **Support for preserving PowerPivot tables in xlsx.** Now PowerPivot tables will be preserved in xlsx.
- **Improved Excel 2013 support.** FlexCel could fail to open some complex Excel 2013 xlsx files.
- **Improved handling of dates between 1900-1-1 and 1900-2-28.** Excel considers 1900 to be a leap year, even when it wasn't. (look at <http://support.microsoft.com/kb/214326>) As FlexCel uses the .NET DateTime which correctly assumes 1900 wasn't a leap year, dates between 1900-1-1 and 1900-2-28 would appear in FlexCel as one day before the dates in Excel. Now FlexCel corrects those dates so they look as in Excel, but the DateTime datatype still doesn't have Feb-29-1900, so that date will still be wrong. It is still advised to not use dates before march-1-1900 when working in Excel.
- **Support for running in machines with FIPS 140 enabled.** Now FlexCel can be used in machines with FIPS 140 policies enforced. (see <http://support.microsoft.com/kb/811833>)
- **New static events in TPdfWriter.** There are 3 new static events: GetFontDataGlobal, GetFontFolderGlobal, OnFontEmbedGlobal. Those work like the already existing GetFontData, GetFontFolder and OnFontEmbed, but being static, they work an application level. If you set them, you don't need to set them for every TPdfWriter or TFlexCelPdfExport instance you create.
- **Improved rendering of formatted numbers.** Some formatting strings (for example "general;-general") were not rendered like Excel.
- **Support for forcing codepage in Excel95.** Now you can force a codepage when opening an Excel 95 file with xls.Open(..., Encoding). While normally you don't need to specify a codepage for xls95 since it is specified in the file, if the file doesn't have a codepage record or has it wrong, you can now specify it here.
- **Support for displaying numbers in Engineering notation.** When displaying numbers in Scientific notation, FlexCel would always use normalized notation (http://en.wikipedia.org/wiki/Scientific_notation#Normalized_notation). Now it can also use Engineering notation if the format string specifies it (http://en.wikipedia.org/wiki/Engineering_notation)
- **SheetProtection is copied when copying sheets from one file to another.** Now FlexCel will copy the sheet protection when you are copying sheets from other file object.
- **Bug fix.** The rendering engine could fail to draw the top gridline in pages after the first when PrintGridLines was true and you were repeating rows at the top.
- **Bug fix.** When opening xls files with data validations and saving them as xlsx, some relative ranges could point to an incorrect cell range in the xlsx file.
- **Bug fix.** Charts in xlsx files didn't preserve textures.
- **Bug fix.** There was an error when recalculating the =LARGE and =SMALL functions in non contiguous ranges of cells.

- **Bug fix.** Sometimes AddFormat() could repeat a format twice in the file.
- **Bug fix.** In certain cases when a macro name had a dot "." on it, FlexCel could fail to open the file.
- **Improved 3rd party compatibility.** Improved generated xls files so they can be loaded by some 3rd party tools.

New in v 6.1.0.0 - September 2013

- **Improved chart rendering.** While the chart engine still can only draw charts in xls files at the moment, it can now read some embedded xlsx records in the xls file so the chart will display like Excel 2007 or newer, not like Excel 2003.
- **Improved xlsx autoshape rendering and conversion.** Now autoshapes in xlsx files are converted better to xls, and also render more faithfully.
- **Improved rendering in iOS and Android.** Now texture bitmaps are also supported in iOS and Android besides Windows.
- **iOS7 support.** Changes to better support iOS7. Some records in xlsx files have been changed so the iOS7 viewer can show the files.
- **Visual Studio 2013 RC support.** VS 2013 RC is now supported.
- **Improved mobile documentation.** New demos added for Android and iOS. Reviewed and improved the documentation.
- **Bug fixes.** Small bugfixes.

New in v 6.0.0.0 - August 2013

- **Cross Platform support.** FlexCel has gone through a big review to make it cross platform. It now runs in Xamarin.iOS, Xamarin.Android and Xamarin.Mac. Read, write, modify and export to pdf or html your Excel files from any iOS, Android or OSX device.
- **Support for the new Excel 2013 xlsx encryption.** Xlsx files encrypted with Excel 2013 can now be opened.
- **Reduced Memory usage.** FlexCel 6 will use from about 1/2 to 1/4 of the memory FlexCel 5 used. We've done a big rearchitecture of the code to ensure it runs fine in memory-limited devices, and this improvement is also available for Windows.
- **More conformant xls files.** All xls files created by FlexCel now pass the Microsoft Office validator if the original file passed it. Note that not all xls files created by Excel pass the Office validator.
- **In OSX and iOS the pdf engine doesn't need more access to the "Fonts" folder.** Now the pdf engine in OSX and iOS can get the fonts directly from memory.
- **Support for changing how FlexCel displays the internal numeric formats.** Excel has some internal numeric formats that aren't stored in the file, and different versions of Excel in different languages might show them different. For example format 37 is defined in some Excel versions as "#,##0 \$;-#,##0 \$" while in others is defined as "#,##0 \$;(#,##0) \$",

(showing negative numbers in parenthesis instead of with a minus sign). The best is not to use those formats that will display different depending on the Excel version, but if you need to make FlexCel behave like one specific localized version, you can change those formats with the static method `XlsFile.SetBuiltInFormat(...)`

- **Support for recalculating XIRR and XNPV functions.** XIRR and XNPV are now recalculated.
- **New `startPageToExport` and `totalPagesToExport` parameters in `TFlexCelPdfExport.ExportSheet`.** Those methods allow to control how many pages are exported.
- **Many improvements and small bug fixes.** There are too many small changes to be mentioned here, but there is hardly any aspect of the library that hasn't been improved.
- **Breaking Change: Compact framework is no longer supported.** Due to all the work to support the new platforms, we had to do some cleaning. Compact framework required a lot of effort to maintain because it lacked too many features, and it isn't being developed anymore by Microsoft.
- **Breaking Change: `System.Drawing` classes have been replaced by internal `TUIClasses` in the API.** This means for example that `System.Drawing.Color` is now `TUIColor`. This change shouldn't break much code since those classes weren't used much in the API, and also because there is an implicit conversion between `System.Drawing.Color` and `TUIColor`. We've made a lot of effort to try to minimize code changes, and chances are that you won't need to change a line of code, but some corner cases might still happen. This change was necessary because most of the newer .NET variations don't include a `System.Drawing` namespace, or it is incomplete.

New in v 5.7.18.0

- **Bug fix.** Images could become invalid when copying sheets between different files.

New in v 5.7.17.0

- **Improvements in `FlexCelPreview`.** New `AutofitPreview` property automatically resizes the preview so it fits to width, height or page. See the "Custom Preview" demo for more information. Now by default the preview scrolls past the last page, enough to allow you to select any page. You can go back to old behavior by setting the `"EndPreviewAtLastPage"` property to true. New `AutofitPreviewOnce` method allows you to autofit the preview just once. New method `MaxPageSize` will return the maximum width and height of the pages in the preview.
- **Bug fix.** Sometimes when changing some printoptions with the API, number of copies might be undefined. It will be 1 now in those cases.
- **Bug fix.** Now FlexCel can read xlsx files with invalid timestamps.

New in v 5.7.16.0

- **Support for xltx and xltm files.** Now you can save xltx and xltm "template" files as you could save xlt. The property `TExcelFile.IsXltTemplate` will tell you if the file you opened was or not a template and you can change it in order to save templates. Note: FlexCel will automatically save the files as templates when you save to a file with extension xlt, xltx or xltm, no matter the value of the `IsXltTemplate` property. You only need to set it when saving to streams.
- **New method `TExcelFile.RecalcRange`.** `RecalcRange` can take any formula that evaluates to a range of cells and return the array of rectangular ranges.
- **New overload for `GetDataValidation`.** The method `TExcelFile.GetDataValidation` can now also return the range where the data validation is applied. You can use this range as an offset to call the new `TExcelFile.RecalcRange` method and convert the text formula in a data validation to a range of cells.

New in v 5.7.14.0

- **New properties in FlexCel preview.** `FlexCelPreview` has new properties for customizing how the page looks like: `ShowThumbsPageNumber`, `PageShadowSize`, `PageShadowColor`, `PageBorderColor`, `PageBorderWidth`, `PageBorderStyle`, `PageNumberBgColor`, `PageNumberSelectedBgColor`, `PageNumberTextColor`, `PageNumberSelectedTextColor`, `Resolution`.
- **New methods `GetSheetSelected` and `SetSheetSelected` in `XlsFile`.** These two new methods allow you to know or set which sheets are selected in a file. Different from `XlsFile.ActiveSheet`, you can select multiple sheets in a single file.
- **Small improvements.** `RoundUp` and `RoundDown` functions now ignore roundtrip digits like Excel. Names that evaluate to rectangular coordinates now can show top, left, right and bottom coordinates even if they are formulas that evaluate to rectangles, and not direct ranges.
- **Bug fixes.** The preview rectangle representing the page was a little larger than what it should be. Html exporting bug when trying to export some files with 16384 columns. Improved compatibility with xlsx files created by third parties.

New in v 5.7.10.0

- **Performance improvements.** Rendering a file with thousands of merged cells is much faster now.

New in v 5.7.9.0

- **Bug Fix.** Fixed a bug in the preview component. Bug was introduced in 5.7.6.0.

New in v 5.7.8.0

- **Bug Fix.** References to external files when saving complex xlsx files which were converted from xls or manually created could be invalid.

New in v 5.7.6.0 - May 2012

- **Improved virtual mode.** Virtual mode now can skip sheets you don't need to read, and also stop reading the file as soon as you have read all the values you need. Look at the improved "Virtual Mode" demo for more information.
- **New Recovery Mode.** A new property in XlsFile, "RecoveryMode", tells FlexCel to try to ignore many common errors in corrupt files so you might be able to open them.
- **Buf fixes.** Improved compatibility with complex and third party created files, fixes in the preview component and many other small fixes and improvements.

New in v 5.7.2.0

- **Bug fix.** TRangeCopyMode.OnlyFormulasAndNoObjects would copy objects when copying from one file to another.
- **Bug fix.** Offset function would return error when recalculating if you used missing arguments in the 2 last parameters.

New in v 5.7.1.0

- **Bug fix.** Some complex merged/span cells could cause wrong results when exporting to HTML.
- **Bug fix.** XlsFile.AddSheet added a sheet before the last position, not after the last one. Now it behaves as expected.

New in v 5.7.0.0 - March 2012

- **Replaced System.IO.Packaging by fully managed native implementation.** The new classes are faster than System.IO.Packaging and also fix some bugs present on it. For example, it doesn't use IsolatedStorage, so it is thread safe for xlsx files bigger than 10 mb and doesn't require special permissions to access IsolatedStorage. Also it fixes many issues in the Mono System.IO.Packaging implementation, allowing FlexCel to officially support xlsx under Mono.

IMPORTANT: If you are creating very big xlsx file in threads, please make sure to update to this version.

- **Support for xlsx in mono and .NET 2.0.** Now xlsx is fully supported in both Mono and .NET 2.0.

- **ATLEAST tag for the config sheet in reports.** AtLeast ensures a datasource has at least n records, and if it hasn't it will return a default value for the records between Count(Datasource) and n. See the documentation in "Using FlexCel Report" pdf.
- **Master detail reports can now be defined with 2 ranges that expand to the same range.** Now if you need for example a **Master** range and a **Detail** range in the same row, you can make the **Master** range bigger than **Detail** (for example **Master** = A1:B1 and **Detail** = A1). While both ranges are the same (both A1:XFE1), FlexCel now can realize the bigger range is the master and use it to know the master-detail relationship. Before you would get a "ranges intersect" message if **Master** wasn't actually bigger than detail.
- **New "Text Qualifier" parameter when importing csv files.** Now when using XlsFile.Import(...) to import a text file, you can define a text qualifier different from a double quote ("). The text qualifier is used when the delimiter is part of the field and so it must be quoted.
- **New property ExcelFile.PrintLandscape.** Allows for setting landscape printing easier than with PrintOptions.
- **Bug fixes.** When exporting to PDF and dpi wasn't 96 and using "XP Style dpi scale" the resulting PDF might be wrong. Issues when printing all sheets in a workbook and skipping the first pages. Allowed to read duplicated row records so some invalid files can be read.
- **Bug Fixes.** Now recalculation works more like in Excel in Lookup and Match functions.
- **Bug Fixes.** Now FlexCel can read invalid xls files which have wrong strings. PdfExport could raise an Exception rendering 0 pixel width metafiles.
- **Bug Fix.** Some numbers near the maximum possible in Excel (1e308) could be stored with reduced precision.
- **Bug fixes.** Small issues in chart preserving. Speed issues when rendering big metafiles.

New in v 5.6.0.0 - November 2011

- **Chart preserving in xlsx.** Now xlsx charts are preserved and updated when you insert or copy rows, copied when you copy ranges, etc. Charts aren't converted between xls and xlsx file formats, so in order to use xlsx charts you need to start from an xlsx file.
- **Support for reading Excel 5 and 95 xls files.** While Excel 95 is not in mainstream use, many third party libraries produce xls 95 files today, and now you can read those files directly with FlexCel. After opening them, they must be saved as xls 97 or xlsx.
- **Support for accessing nested properties when using LINQ in reports.** Now when using LINQ in reports you can access nested properties from the template. If for example you have a class "Orders" and this class has a nested class "Customer", you can write in the template "<#Orders.Customer.Name>". You can use as many nesting dots as needed, as long as the properties have a single value.

- **Support for calculating circular references.** Now FlexCel can calculate iterative workbooks. The new properties "OptionsRecalcCircularReferences", "OptionsRecalcMaxIterations" and "OptionsRecalcMaxChange" in XlsFile allow you to control the iterative recalculation. As always, the APIMate tool will show how to set those properties in a sheet.
- **APIMate improvements.** APIMate will now show the schema of the fonts when using themed fonts (Excel 2007).
- **Bug fix.** There could be an error when manually copying formulas that included named ranges from a workbook to another.
- **Bug fixes.** <#delete range> tags could work wrong in nested reports. Xlsx files could be invalid after copying cells from other workbook. Better compatibility with xls files generated by other 3rd party tools.
- **Bug fix.** .NET 3.5 XMLReader can hang when reading some malformed xlsx files. Now FlexCel will throw an Exception when reading those files in .NET 3.5. Note that .NET 4 already worked fine and keeps doing so.
- **Bug fix.** Xlsx files could save the same image more than once when used in many places. Now only one copy of every unique image is stored.
- **Bug fix.** Xlsx files without printer information could default to landscape instead of portrait.
- **Bug fix.** Problem when saving xlsx files with more than one pivot table in different sheets.
- **Performance enhancements.** When reading 2007/2010 xls files with thousands of manual styles.
- **Performance enhancements.** Improved xls saving performance.

New in v 5.5.1.0 - August 2011

- **Bug fixes.** Manual page breaks in xlsx could be ignored by Excel. Improved compatibility with invalid xls files.
- **Performance improvements.** Exporting to html/mhtml now is faster for some files.
- **Support for rendering non contiguous print areas.** Now when exporting to Pdf/Html/Images/etc. FlexCel will honor print areas that have many different sections, like "=Sheet1!\$A\$1:\$B\$4,Sheet1!\$D\$5:\$F\$7".
- **Bug fix.** In some cases, after copying a chart from one sheet to another, you wouldn't be able to select the chart anymore from FlexCel to continue working with it.
- **Performance improvements.** Exporting to pdf now is faster for some particular files. A new property "IgnoreFormulaText" in XlsFile allows you to ignore the formula text when reading the cells in a file, speeding up the reading. Look at the "Performance" pdf for more information.
- **Bug fix.** Malformed hyperlinks now can be read. FlexCel can now read xlsx files with malformed hyperlinks.

- **Added a new ErrorAction TExcelFileErrorActions.OnXlsxMissingPart.** A new ErrorAction member has been added to allow reading corrupted xlsx files that lack some parts. This is off by default, you need to explicitly call XlsFile.ErrorActions &= ! TExcelFileErrorActions.OnXlsxMissingPart for it to work. As with all other errors, when it happens it will be logged to FlexCelTrace.
- **Bug fixes.** Fixed problem that could rarely happen with nested relationships inside ADO.NET tables in a report.
- **Bug fixes.** Support for reading [this row] tokens in formulas inside Excel 2007 tables. We don't still process them and they will be imported s #REF!, but at least you can read the file.
- **Bug fixes.** Issue when copying data validations and conditional formats by columns.
- **Bug fixes.** When sorting a cell range formulas referring to that range could be offset by one.

New in v 5.5.0.0 - May 2011

- **Breaking Change: Native support for LINQ and Entity Framework as data sources for reports.** Now you can use any IQueryable iterator to create a report, besides DataSets. Breaking Change: VirtualDataTable and VirtualDataTableState objects had to be modified to allow the best performance when using IQueryable. If you have defined your own VirtualDataTable/State descendants, you will have to make some changes in the code to compile.
- **New "Virtual Mode" for reading xls and xlsx files on demand.** The new "Virtual Mode" allows you to read huge xls/xlsx files on demand, without loading the full file into memory. If you are importing big files with FlexCel, this new mode can make a big difference. Look at the "Virtual Mode" demo for more information.
- **Support for reading and writing encrypted xlsx files.** Reading and writing encrypted xlsx files is fully supported, both Excel 2007 (Standard Encryption) and Excel 2010 (Agile Encryption)
- **Support for protected xlsx files.** Full support for protected xlsx workbooks and sheets, including password protection.
- **Pivot Table preservation in xlsx files.** Pivot tables are now preserved when saving xlsx files, and they can be used in reports, and copied between sheets or files.
- **Macro preservation in xlsx files.** Macros are now preserved when when opening xls files and saving as xls or xlsx, or when opening xlsx files and saving as xlsx.
- **Full support for R1C1 formulas.** Now you can use the R1C1 notation besides A1 when entering or reading formulas. You can change the cell reference mode with the property XlsFile.FormulaReferenceStyle.
- **Full support for all objects in the Forms palette (radio buttons/group boxes/comboboxes/listboxes/spins/labels/scrollbars/buttons) in the API, rendering, and xlsx.** The new methods allow you to add and modify any object in the forms toolbar. Also now all those objects will be printed/exported to pdf/html/images, and they are fully supported in xlsx too.

- **Support for buttons/checkboxes/readio buttons/group boxes/comboboxes/listboxes/spins/labels/scrollbars in APIMate.** Now APIMate will show how to add/modify checkboxes and all objects in the forms palette.
- **Support for Autoshapes in xlsx.** Now autoshapes in xlsx are preserved, converted between xls and xlsx, and rendered.
- **Header and footer images support in xlsx files.** Images in headers and footers are now fully supported in xlsx.
- **Exporting named ranges to html.** A new property "ExportNamedRanges" in FlexCelHTMLExport allows you to export the names in the sheet as span ids that you can use later to modify those cells with javascript. A new event, "NamedRangeExport" allows you to customize how those names are exported.
- **XlsFile now implements IEnumerable.** You can now loop through the cells in an Excel file with a foreach loop.
- **New overload of XlsFile.** NewFile allows to specify the version of Excel used to create the file. Now you can specify the version of the blank xls or xlsx files created by FlexCel. Different versions of Excel have different default fonts, column widths, etc, and now you can specify exactly which version you are creating. This is specially useful for APIMate, since in older FlexCel versions it would always create a 2003 xls file, and modify it with code to match the newer versions. Now it creates the correct file, and code needed is much less.
- **Fixed length text exporting now exports merged cells and cells that span to the right.** When exporting Excel files to fixed length text, now merged cells and cells spanning to the right will use the full length available instead of cutting at the cell end.
- **Breaking Change: Deprecated Get/SetCheckboxLinkedCell methods in the API.** While those methods will still work, we have introduced a generic Get/SetObjectLinkedCell method that should be used instead.
- **Bug fixes and performance optimizations.** Xlsx files now load and save much faster, and performance was improved also for xls files. Pdf files are faster too. A new document in how to get the best performance in FlexCel is also included.
- **Experimental MonoTouch support.** FlexCel can now be compiled for MonoTouch. With it, you can read, write and recalculate xls files in iPhone apps. Note that this support is basic and not fully tested, even when it looks to be working fine. Xlsx file format and rendering (exporting to pdf/html/printing) are not supported. Search for "MonoTouch.sln" in the distribution.

New in v 5.3.0.0 - August 2010

- **Support for Recalculation of 49 built in functions new to Excel 2007.** Includes support for: Averagelf, Averagelfs, Bin2Dec, Bin2Hex, Bin2Oct, Convert, Countlfs, CoupDayBs, CoupDays, CoupDaysNc, CoupNcd, CoupNum, CoupPcd, Dec2Bin, Dec2Hex, Dec2Oct, Delta, DollarDe, DollarFr, Duration, EDate, Effect, EoMonth, FactDouble, Gcd, GeStep, Hex2Bin, Hex2Dec, Hex2Oct, IfError, IsEven, IsOdd, Lcm, MDuration, MRound, MultiNomial, NetworkDays, Nominal, Oct2Bin, Oct2Dec, Oct2Hex, Quotient, RandBetween, SeriesSum,

SqrtPi, SumIifs, WeekNumn, WorkDay, YearFrac. Some of the functions are new to Excel 2007 (like Averagelf), and others were previously available in Add-ins. Look at SupportedFunctions.xls in documentation for more details.

- **Support for Recalculation of 8 built in functions new to Excel 2010.** Includes support for: NETWORKDAYS.INTL, WORKDAY.INTL, AGGREGATE, CEILING.PRECISE, ISO.CEILING, FLOOR.PRECISE, PERCENTILE.EXC, QUARTILE.EXC Look at SupportedFunctions.xls in documentation for more details.
- **Support for all Excel 2010's "Renamed Functions".** Now you can enter any of the Excel 2010 renamed functions in FlexCel, and those functions whose previous name was previously recalculated in FlexCel (and have the same paramters) will also recalculate now in FlexCel. Renamed functions: BETA.DIST, BETA.INV, BINOM.DIST, BINOM.INV, CHISQ.DIST.RT, CHISQ.INV.RT, CHISQ.TEST, CONFIDENCE.NORM, COVARIANCE.P, EXPON.DIST, F.DIST.RT, F.INV.RT, F.TEST, GAMMA.DIST, GAMMA.INV, HYPGEOM.DIST, LOGNORM.DIST, LOGNORM.INV, MODE.SNGL, NEGBINOM.DIST, NORM.DIST, NORM.INV, NORM.S.DIST, NORM.S.INV, PERCENTILE.INC PERCENTRANK.INC, POISSON.DIST, QUARTILE.INC, RANK.EQ, STDEV.P, STDEV.ST.DIST.2T, T.DIST.RT, T.INV.2T, T.TEST, VAR.P, VAR.S, WEIBULL.DIST, Z.TEST Look at SupportedFunctions.xls in documentation for more details.
- **New "BALANCED COLUMNS" mode for reports.** Now you can do parallel column reports where all columns stay balanced and cells are automatically added to pad the columns with less records. Look at the new "Balanced Columns" demo for an example in how to use it.
- **New FIXEDN ranges for reports.** FixedN ranges will behave as "FIXED" ranges for the first n records, and then behave as normal "__" ranges. For example the name "__db__FIXED2" will overwrite the 2 first records in the template, and then insert the rest. Look at the new "Balanced Columns" demo for an example in how to use it.
- **New ROWS function for reports.** Allows to create datasources in the fly from the template with a defined number of rows. Look at the new "Balanced Columns" demo for an example in how to use it.
- **New TCopyRangeMode.Formats to copy formats from a block of cells to another.** Now you can call InsertAndCopyRange with TRangeCopyMode.Formats to copy the cell formats from one place to another.
- **Improved Medium trust support.** Our obfuscation tool was having issues when running in Medium Trust, now it should be fixed. This allows for deployment in shared hosting like godaddy.
- **Many small fixes and enhancements.** As always, a lot of small fixes and improvements have been done.

New in v 5.2.0.0 - April 2010

- **Breaking Change: Support for .NET 4.0.** Includes support for the new 4.0 security model.

- **Breaking Change: Deprecated support for .NET 1.1.** In order to move faster to the new technologies, we had to deprecate .NET 1.1 support for this version.
- **Full Comment support in xlsx.** Now comments are fully supported in xlsx besides xls. You can also set extended properties like the comment color directly from the API.
- **Full Data Validation support in xlsx.** Now data validation is fully supported in xlsx besides xls.
- **Full Hyperlink support in xlsx.** Now Hyperlinks are fully supported in xlsx besides xls.
- **Full Checkbox support in the API, rendering, and xlsx.** The new methods: Get/SetCheckboxState, Get/SetCheckboxLinkedCell and AddCheckbox allow you to add and modify checkboxes states and linked cells. Also now checkboxes will be printed/exported to pdf/html/images, and they are fully supported in xlsx too.
- **New "DateFormats" parameter supported when opening or importing CSV files, and also when setting cells from string.** This parameter allows you to specify only a subset of supported datetime formats when importing, to ensure .NET won't interpret invalid strings as dates. For example, calling: `xls.Open("test.csv", TFileFormats.Text, ',', 1, 1, null, new string[] { "d/M/yyyy", "hh:mm" }, Encoding.Default, true);` will only import dates in format "d/m/yyyy" or times in format "hh:mm".
- **New "FirstSheetVisible" property in XlsFile.** This property controls what is the first sheet tab that is shown in the sheet bar at the bottom of Excel.
- **New "CenteredPreview" property in FlexCelPreview.** When true, previews will render centered in the window, as they do in Excel.
- **Added support for new functions in recalculation.** Added support for FREQUENCY.
- **Performance Improvements and bug fixes.** The move away from .NET 1.1 allowed us to switch to generics much more of the code, with up to 10% speed up. Together with other performance improvements, 5.2 can be up to 30% faster in some cases.
- **Database in all demos migrated from Access to SQL Server compact.** As Microsoft still doesn't support the JET driver in 64 bits, we changed the demos to use SQL Server Compact Edition instead. This way you will be able to test the database demos in pure 64 bits.
- **Tested against Office 2010 RTM.** Generated files have been tested against the release version of Office 2010.

New in v 5.1.0.0 - January 2010

- **Excel 2010 XLS file support.** Excel 2010 introduced a new "Protected View" that will flag old FlexCel xls files as "unsafe". This release fixes this so files will open in 2010 without warnings. It is important to update to this version as soon as possible, so when Excel 2010 is released your files will keep on working without warnings.
- **BASIC IMAGE SUPPORT IN XLSX.** Simple images are now fully supported in xlsx besides xls. They will be converted and preserved when you open an xls file and save as xlsx or viceversa, and also rendered to pdf, etc. Grouped images and autoshapes are still not supported in xlsx, but coming soon.

- **THEME SUPPORT.** Now the rendering engine will use other themes besides the standard office theme, and you are also able to modify the themes in a sheet.
- **BETTER INDEXED COLOR SUPPORT.** The new method "OptimizeColorPalette" will modify the Excel 97-2003 color palette in an xls file so it includes the colors used in the sheet. Excel 2007 or newer don't need this as they support RGB colors.
- **MISC IMPROVEMENTS IN XLSX FILE SUPPORT.** Support for autofilters, selections, printer driver settings, showing gridlines/headers and many small window properties when loading or saving xlsx. Macros are preserved when reading an xls file and saving as xlsx.
- **IMPROVED MEDIUM-TRUST SUPPORT.** Improved fallback in Exceptions when running in Medium Trust. As before, FlexCel can be compiled with "FULLYMANAGED" conditional define to not only be 100% safe but also 100% managed code. But now even the dlls compiled without "FULLYMANAGED" will work fine in medium trust.
- **NEW METHODS IN XLSFILE FOR EXPORTING AND IMPORTING FROM TEXT FILES.** The new methods XlsFile.Import and XlsFile.Export provide more flexibility when working with text files than the existing XlsFile.Open/XlsFile.Save methods. Now you can specify a "fixed length" file besides a text delimited file, and you can also import a text file in the middle of an existing file.
- **NEW COPY MODE ALLOWS TO COPY OBJECTS MARKED AS "DON'T COPY" when copying ranges or sheets.** TRangeCopyMode.AllIncludingDontMoveAndSizeObjects will copy everything when used in InsetAndCopyRange. InsetAndCopySheets will use this mode now by default.
- **NEW METHOD GETUSEDNAMEDRANGES IN THE API.** Returns which ranges are being used in formulas inside the sheet and which aren't.
- **NEW METHOD CELLRANGEDIMENSIONS IN THE API.** Returns the dimensions a range of cells would use when rendered. Can be used when rendering to a bitmap to calculate the size of the bitmap what will hold the cells.
- **TOOLS ARE NOW PRECOMPILED WITH .NET 3.5.** The tools like ApiMate and FlexCelDesigner used to come precompiled with .NET 1.1, so you can use them no matter which .NET version you have in your development machine. But as .NET 1.1 doesn't support xlsx, now they come with 3.5.
- **BUG FIXES.** Small fixes and improvements.

New in v 5.0.1.0 - October 2009

- **IMPROVED PERFORMANCE INSERTING ROWS WITH MANY IMAGES.** This will also speed up reports with lots of images too.
- **NEW FUNCTION SUPPORT FOR RECALCULATION.** PercentRank is implemented now too.
- **IMPROVED SUPPORT FOR OTHER THIRD PARTY EXCEL GENERATED FILES.** While we support virtually every xls file Excel generates (and we are not aware of any file we can't retrieve if it has been saved with Excel and it is in xls 97 or up), some third party apps

create files with wrong information, that rely in bugs (and sometimes even in buffer overflows) in Excel to work. In this release we implemented support for many of those files, including files generated by SAP

- **BUG FIXES.** This is primary a maineneance release, and there are many bug fixes and small improvements, mainly in the Excel 2007 support (both xlsx and "Excel 2007 xls") It is recommended that you update from 5.0.

New in v 5.0.0.0 - October 2009

- **EXCEL 2007/2010 SUPPORT.** Included support for new features in Excel 2007/2010:
 - Basic support for reading and writing xlsx file format. Note that due to framework limitations, you need .NET 3.5 for xlsx support.
 - Expanded the rows to 1048576 and the columns to 16384. A compatibility mode still lets you work with the smaller grid should you need to do so.
 - Support for Excel's 2007 true color and themes. Breaking change: ColorIndex properties don't exist anymore and now are just Color. You can still access the color indexes with Color.Index.
 - Support for gradients in cell backgrounds; to get/set them or to export them to pdf/images/print.
 - Support for a different header and footer for the first page and for even pages; to get/set them or to export them to pdf/images/print.
 - Support for comments in named ranges.
 - Cell indentation can go up to 250 characters instead of the old 15.
 - Methods OptionsMultithreadRecalc, OptionsForceFullRecalc, OptionsAutoCompressPictures, OptionsBackup, OptionsCheckCompatibility in XlsFile class allow to configure the corresponding settings in an Excel file.

Please note that xlsx file format support in 5.0 doesn't include objects/charts/images or conditional formats. That will be added along the 5.n series.

Except for the bigger number of rows and columns (which can't be retrofitted to xls), all new properties will be saved even to the old xls files. They won't be available when opening the files in Excel 2003, but they will show in Excel 2007. And they will be used by FlexCel when printing or exporting to pdf/html, etc.

Take a look at the new section "Considerations about Excel 2007 support" in the API Guide for more information about updating to xlsx support.

- **NEW HTML 3.2 SIMPLE EXPORTING MODE.** This new exporting mode won't use CSS or floating images, and most settings will be done through simple tags. While some style tags are still used when there is no other option, they are mostly not used either. This mode isn't as faithful reproducing Excel files as the existing ones, and it doesn't validate either, but it can be very useful when you need simple HTML more than exact representation of the xls file. It can be used with devices or browsers that don't support CSS, or in places where you can't change the existing CSS definitions (for example if you are adding a table in a blog post, where you can't change the page headers to include other CSS file).
- **WHAT-IF TABLES.** Now FlexCel can recalculate What-if tables, and you can add or read the What-if tables in a file. APIMate also supports What-if tables now, and will show the syntax to create them.

- **ADDED RECALCELL METHOD TO THE API.** This method allows you to calculate only a cell and its dependencies, not the whole workbook. It can be useful if you are using FlexCel as a calculator and making thousands of recalculations where you are only interested in the value of one cell.
- **ADDED RECALCEXPRESSION METHOD TO THE API.** With this method you can calculate any formula that is not in the file. For example, if you want to know the sum of the cells in column a of a worksheet, you can use `xls.RecalcExpression("=sum(A:A)")`.
- **SUPPORT FOR ENTERING MULTICELL FORMULAS WITH THE API.** Now you can not only enter array formulas with the API as you could before, but also enter array formulas that span over more than one cell. We only added the ability to add them from the API, FlexCel was already fully aware of multicell array formulas and could recalculate them too. ApiMate will show you the syntax to enter them.
- **IMPROVED SUPPORT FOR DATE AXIS IN CHARTS.** Now date axis in charts behave exactly the same way they do in Excel.
- **IMPROVED SUPPORT FOR NUMERIC FORMATS.** Now "*", "_" and "?" characters in format strings are fully supported when rendering files, and will show exactly as they do in Excel.
- **IMPROVED RENDERING.** Support for diagonal borders. A new "Linespace" property in XlsFile object allows you to fine tune the linespace between 2 lines in multiline cells.
- **IMPROVED COLOR MATCHING ALGORITHM FOR CONVERTING TO INDEXED COLORS.** Now NearestColorIndex will use the Euclidean distance in L*a*b* color space instead of RGB, for improved color matching.
- **NEW SAVEFORHASHING METHOD.** This method will save the file in a file format that will remain the same if the file didn't change, ignoring the timestamps present in the xls file format. So you can hash this value and use the hash to compare it to a new file, and know if something changed. Cell selections and sheet selections are not saved by default, but they can be included.
- **ADDED ABILITY TO READ AND WRITE "SHARED WORKBOOK" PROTECTION OPTIONS.** Now you can change the shared workbook protections options in any file, xls orxlsx.
- **IMPROVED PERFORMANCE.** FlexCel 5 has been a big rewrite that allowed use to tweak many places for even better performance.

New in v 4.9.6.2

- **IMPROVED AUTOFIT OF MERGED CELLS.** Now when autofitting rows and a merged cell has more than one row, you can select which one of the rows from the merged cell will be updated. Same for autofitting columns and merged cells with more than one column. This applies to both reports and API. See "Autofitting Merged Cells" section in the API Guide for more information.
- **SYNTAX HIGHLIGHT WHEN DEBUGGING REPORTS.** Now when in debug mode, strings will be maroon, booleans blue and errors red.

New in v 4.9.6.0 - November 2008

- **DELPHI PRISM SUPPORT.** All demos have been converted to Delphi Prism, installation now installs into Delphi prism, and APIMate can generate Delphi Prism code.
- **ABILITY TO MODIFY CHART SERIES FROM THE API.** Now you can directly modify chart series from the API.
- **FULL SUPPORT FOR WORKING WITH NAMED STYLES FROM THE API.** Now you can create, modify or remove named styles in the Excel file. Also apply styles or find out which styles are applied to a cell.
- **TTC FONT SUPPORT WHEN EXPORTING TO PDF.** Now TTC (True Type Collection) fonts are fully supported when exporting to PDF. This includes subsetting.
- **NEW SETEXPRESSION METHOD IN FLEXCEL REPORT.** Now you can use the SetExpression method in FlexCelReport to dynamically add formulas to a report. For example, you might have an edit box where the user enters an expression like "<#evaluate(<#Order.Amount> * <#Order.Vat>)>", and this expression will be used in the final report. With this method you can reuse the same template to evaluate different formulas.
- **IMPROVED HTML RENDERING.** Fixed small browser incompatibilities. Chart sheets now are exported too. Now FlexCelViewer renders by default in XHTML 1.1, to be compatible with the designer.
- **IMPROVED SPEED RENDERING CONDITIONAL FORMATS.** Applies if you have thousands of conditional formats defined in a sheet, speed of rendering will be much faster.
- **IMPROVED SPEED IN FORMULA RECALCULATION.** Now formula recalc is faster if you are using full sheet ranges (like A:IV).
- **SMALL BUG FIXES.** Autofilters now are updated when inserting or deleting columns.

New in v 4.9.5.0

- **FONT SUBSTITUTION IN PDF.** A new Property "FallbackFonts" in FlexCelPdfExport allows you to specify a list of "Fallback" fonts that FlexCel will use when the character to print is not in the main font. See "Dealing with missing fonts and glyphs" in UsingFlexCelPdfExport.pdf for more information.
- **NEW DBVALUE TAG FOR REPORTS.** Allows you to know the value of any record of a data table, for example to merge similar cells. See the new "Merging Similar Cells" demo.
- **IMPROVED NAMED RANGE SUPPORT.** New methods DeleteNamedRange and ConvertExternalNamesToRefErrors will let you delete a range or convert all ranges with external references in a file to #REF! errors.
- **NEW TRYTOCONVERTSTRINGS PROPERTY IN FLEXCELREPORT.** When your data is stored as strings in your database, this property will make FlexCel enter the correct datatype for the contents into the cells.

New in v 4.9.2.0 - June 2008

- **NEW LIST TAG FOR REPORTS.** The `<#List(>` tag allows to aggregate a dataset into a list that can be dropped into a single cell, or also to use other tags without being inside a named range.
- **NEW SEMIABSOLUTEREFERENCES properties in the API and in Reports.** Now you can control how to change absolute references in formulas referring to cells inside the block being copied. For example in Excel, if you have Cell A1: 1, Cell B1: = $\$A\1 , and copy the row down, the new row will be Cell A2: 1, Cell B2: = $\$A\1 . If you set this new property to true, Cell B2 will be = $\$A\2 , since A2 is inside the block being copied. You can use this on the API when copying blocks with absolute references you would like to change, or in multi master detail reports, to ensure absolute references point to the right place.
- **SHEET TAB COLOR SUPPORT.** Now you can read or set the color of a sheet tab using the new `SheetTabColorIndex` property in `XlsFile`. (This feature is supported in Excel XP or newer)
- **SHEET TAB COLOR EXPORTED TO HTML.** Now by default if a sheet has a tab color, it will be shown in the resulting HTML file. You can change this by setting the new "UseSheetTabColors" property in the `StandardSheetSelector` class to false.
- **BUG FIXES.** Fixed an error when subsetting complex true type fonts when exporting to pdf. Fixed an error when entering bmp image in compact framework.

New in v 4.9.1.0

- **Breaking Change: SUPPORT FOR EMBEDDING FONT SUBSETS IN PDF.** Now FlexCel can embed only the subset of characters being used from a font into a PDF file, allowing smaller PDF files when embedding unicode fonts. NOTE: This is a BREAKING change, since font subsetting is enabled now by default. If you want to keep the old behavior (for example to have editable PDF files) you need to set `FontSubset` property to `DontSubset` in the PDF export components.
- **GLOBAL ERROR HANDLER FOR NON-FATAL ERRORS.** There is a new `FlexCelTrace` global class where you can hook a listener to get notified of all non-fatal errors while working with FlexCel. You can use it for example to know when a font is not installed in the system and is being replaced by other in a pdf file, or when a character is not present in a font and so it will show as a blank square. See the new "Error Handling" demo and the PDF documentation in the PDF export guide.
- **ADDED TOP(N) FILTER FOR REPORTS.** With this new filter you can get the top n items from a table directly from the template without touching the code. See the modified "Fixed Forms With Datasets" demo.
- **IMPROVED HTML RENDERING.** Improved how exported HTML files are generated.
- **IMPROVED PDF EXPORT.** Added a new event allowing to control whether to embed or not and individual font. See the modified "Export PDF" demo.
- **BUG FIXES.** Fixed bug with some functions when recalculating linked files. Fixed overflow exception when creating charts with very large values.

New in v 4.9.0.0

- **RECALCULATION OF LINKED FILES.** Now FlexCel can recalculate across linked files, even files with circular links. See the new section about Workspaces in the PDF API Guide.
- **AGGREGATE SUPPORT IN REPORTS.** The new tag "Aggregate" allows to sum, average or find the minimum or maximum value in a dataset from the template. You can use it when you can't modify the data layer. See the new "Aggregates" demo.
- **SUPPORT FOR HTML TAGS WHEN REPLACING TEXT IN AUTOSHAPES.** Now you can use html inside autoshapes as you could use inside normal cells.
- **IMPROVED RENDERING.** Fixed small issues when rendering Excel spreadsheets.

New in v 4.8.0.1

- **IMPROVED RECALCULATION.** Fixed a bug that might cause a file not to be recalculated in the second time you call recalc with complex files. Small performance improvements.
- **IMPROVED FORMAT DISPLAY.** Added support for [mm], [hh] and [ss] format specifiers for elapsed time.
- **IMPROVED MONO COMPATIBILITY.** Changed internal compression routines in pdf so they work under mono

New in v 4.8.0.0

- **PDF SIGNING.** Now you can digitally sign the generated pdfs, with both a visible or non visible signature.
- **NEW APIMATE TOOL.** This new tool can convert an Excel file to code, so you can see how to call the FlexCel APIs. Code can be generated in C#, VB.NET or Delphi.NET. A flash demo showing how to use it is available at <http://www.tmssoftware.com/flexcel/tutorial.htm>
- **IMPROVED HTML GENERATION.** Includes the ability to export headers and footers as blocks above and below the spreadsheet, and fixes to workaround internet explorer bugs. Exporting headers and footers to HTML is off by default, but you can turn it on. (Look at the Export to HTML demo)
- **BUG FIXES.** Small fix with formulas in Data Validation, and support for [>n] tags in numeric formatting expressions.

New in v 4.7.0.1

- **FLEXCEL DESIGNER BUG FIX.** FlexCel designer could raise an Exception when started.

New in v 4.7.0.0

- **INTELLIGENT PAGE BREAKS.** Even when there is no direct support for widow/orphan lines in Excel, FlexCel now provides a way to keep rows and columns together avoiding page breaks in the middle of important data. You tell FlexCel which rows you want to keep together, and it will automatically add page breaks at the needed points in the file so it prints as you want to. This new feature can be used both from the API or from the reports. For more information, look at the "Intelligent Page Breaks" demos in the Report and API sections.
- **BETTER ERROR HANDLING OF PAGE BREAK ERRORS.** In previous FlexCel versions you could choose whether to raise an Exception or silently ignore errors when trying to insert more than the maximum allowed number of manual page breaks (1026). In this version you can insert as many Page Breaks as you want, and the error or silent ignore will be done at save time. This allows to have more than 1026 manual page breaks when exporting to PDF without saving as xls.
- **<#DEFINED FORMAT> TAG FOR REPORTS.** Allows to know if a user defined format is defined or not. Look at the Intelligent Page Breaks demo in the report section.
- **NEW <#IMGPOS> AND <#IMGFIT> <#IMGDELETE> TAGS FOR REPORTS.** You can use ImgPos to center or align an image dynamically inside a cell. ImgFit will resize the rows and columns below the image so the image is fit in one cell, and ImgDelete will delete an image. Take a look at the modified Images demo or at the new Features Page demo.
- **IMPROVED AUTOFIT IN API AND <#AUTOFIT> TAG FOR REPORTS.** Now you can Autofit rows and columns setting a maximum and a minimum height/width for the autofit.
- **IMPROVED <#IMGSIZE> TAG FOR REPORTS.** Now the ImageSize tag can do a "Best Fit" resize. You define the maximum size of the image in the template, and ImageSize will resize the image so it is as big as possible keeping the aspect ratio and inside the bounds you select. Look at the modified Images demo.
- **IMPROVED <#FORMAT RANGE>, <#DELETE RANGE> AND <#MERGE RANGE> TAGS FOR REPORTS.** Now you can use named ranges instead of strings like "a1:a2" to define the ranges to format, merge or delete. It is recommended that you use this new way, so the ranges will adapt when you insert rows in your template. (A tag <#FORMAT RANGE(a1:a3;blue)> will not change to A2:A4 if you insert a new row at A1. A tag using a named range will.)
- **NEW IMAGEBACKGROUND PROPERTY WHEN EXPORTING TO HTML.** Allows you to define a background color like white for images in html files, so they are not transparent and they show fine in Internet Explorer 6 without needing to use the FixIE6TransparentPngSupport to true or using gif images instead of png.
- **BUG FIXES.** Small bug fix for border cases when inserting columns. AND function now can AND over a range of cells. Report Expressions now can use named ranges.

New in v 4.6.0.0

- **SUPPORT FOR VISUAL STUDIO 2008 AND .NET 3.5 FRAMEWORK.** Also updated Setup with the option for installing into VS2008.

- **NATIVE DEMOS FOR VISUAL BASIC.NET.** All more than 50 demos have been converted to Visual Basic .NET, allowing for easier study for vb users.
- **SUPPORT FOR CUSTOM EXCEL FORMULA FUNCTIONS.** Now you can define your own classes that implement Excel custom formula functions (like for example the ones in the Analysis Toolpack Addin, or any function you define using a macro). You can read formulas using those functions from Excel, write them or calculate them. See the "Custom Excel Formula Functions" demo for more information.
- **SUPPORT FOR AUTOFILTERS IN API.** Now you can read or write autofilters in a sheet using the API.
- **IMPROVED COPYING FROM ONE FILE TO ANOTHER.** Now when copying between files Charts will be copied too, and all external references will be converted to the new file. Also Autofilters will be copied when there are no autofilters in the destination sheet.
- **IMPROVED GENERIC REPORTS.** Now you can write expressions in a cell with a `<#DataSet.*>` tag, leading to much more powerful generic reports. Also now if the cell with the `<#DataSet.**>` tag has an autofilter, the autofilter will be propagated to the following columns. See the improved Generic Reports demo for more information.
- **MORE CUSTOMIZATION IN THE <#INCLUDE> TAG.** Now you can include files in your reports without running a report on the include, and also specify if you want to copy the column widths and row heights from the included report into the parent. See the documentation in the Include tag.
- **BUG FIXES.** Small bug fixes and optimizations.

New in v 4.5.1.0 - October 2007

- **DEBUG MODE IN FLEXCELREPORT.** A new property in FlexCelReport, DebugExpressions, allows to output the full stack trace of the report tags to the cells where they are, instead of the tag value. Other property, ErrorsInResultFile allows to log the error messages to the generated file instead of raising exceptions. See the new Debugging Reports demo and the Debugging Reports section in the pdf user guide for more information.
- **DATA VALIDATION SUPPORT.** Added methods to add, delete, change or get information about the Data Validation of a cell.
- **IMPROVED OUTLINE SUPPORT.** Added methods to collapse or expand the outlines in a sheet to a specified level, to collapse and expand individual nodes, or to find out if a row or column is an outline node (contains a "+" sign).
- **IMPROVED XLS COMPATIBILITY WITH EXCEL 2007.** Some really complex image manipulations could cause Excel 2007 to fail to load the generated xls files. It has been fixed now.
- **IMPROVED COMPATIBILITY WITH MONO.** Tested with .NET 2.0 implementation of Mono and workarounded MONO issues.
- **NEW OBJECT EXPLORER AND ADVANCED API DEMOS.** The first shows how objects in a sheet are nested, and the second shows how to use different methods in the API.

New in v 4.5.0.0 - September 2007

- **HTML EXPORTING ENGINE.** A new component TFlexCelHtmlExport can export Excel files to html, in HTML 4.01 strict or XHTML 1.1 and fully standards compliant, with the quality you have come to expect from us. Most things, like images, charts, merged cells, conditional formats, wordart, etc. are exported. Multiple sheets can be exported in tabs or as a single file. Support for ie 6/7, Firefox, Opera and Safari.
- **FLEXCELASPVIEWER.** Allows viewing Excel files as html directly from any ASP.NET application. Just drop the component in a WebForm, and assign it to an xls file. (only ASP.NET 2.0 supported)
- **META TEMPLATES.** The new <#PREPROCESS> tag allows a template to modify itself before creating a report. You can now for example create reports that will automatically delete a column from the report template if the dataset does not have the field. See the "Meta Templates" demo for more information.
- **PARTIAL FORMAT DEFINITIONS IN REPORTS.** Now you can define formats in the config sheet that will apply only a part of the cell format. For example, if you name a format "Header(background)" it will only apply the background of the cell and not all the other properties. See EndUserGuide.pdf for more information, and the Multiple Sheet Report demo for an example.
- **API IMPROVEMENTS.** New method XlsFile.RenderObject allows exporting any image/chart/autoshape into an image. New method XlsFile.RenderCells allows exporting a range of cells (without objects) into an image. You can still export full spreadsheets to images with FlexCellmgExport, but these methods provide lower level access.
- **FULL TEXT SEARCH IN DEMOS.** Now you can easily find the demo that shows a feature you are interested in by typing in the Search box in MainDemo.
- **BUG FIXES AND SPEED IMPROVEMENTS.** Small fixes in rendering and overall speed enhancements.

New in v 4.0.0.3 - May 2007

- **BUG FIXES.** Locales like Turkish where I (uppercase) is not the same as i (lowercase) had problems. When printing/exporting to pdf more than one sheet, sometimes the "print to fit" size for the second sheet could be calculated wrong. Rotated transparent shapes exported to pdf could be exported wrong.

New in v 4.0.0.2 - April 2007

- **IMPROVED CSV IMPORT/EXPORT.** Now supporting Locales.
- **BUG FIXES.** Error in border cases when adapting formulas after inserting rows in different sheets.

New in v 4.0.0.0 - December 2006

- **CHART RENDERING.** Now most 2d charts are printed and exported as images or pdf. 3d charts are exported as their 2d equivalents. Bubble, Surface and Radar charts are not exported.
- **NEW HTML CAPABILITIES.** Now you can directly enter HTML formatted strings into an Excel cell, using `TRichString.FromHtml()`. Also you can convert the rich text in an Excel cell into an HTML string. A new property "HtmlMode" on `FlexCelReport` allows you to do reports from HTML data. Also the new `<#HTML>` tag allows to select which cells you want to use html and which ones you don't no matter the "HtmlMode" value. See the "HTML Reports" demo for more information.
- **SUPPORT FOR WRITING PXL 2.0(POCKET EXCEL) FILES.** Now you can not only read but also create native Pxl 2.0 files.
- **READING DOCUMENT PROPERTIES.** Now you can read the Author, Title, etc. of any xls document.
- **IMPROVED FLEXCELIMAGEEXPORT.** The new method "SaveAsImage" allows creating multipage tiffs, fax, png, gif and jpg images of any xls file with just a method call. See the "PRINT PREVIEW AND EXPORT" demo for more details.
- **IMPROVED PRINTING AND EXPORTING.** New properties `AllVisibleSheets` and `ResetPageNumberOnEachSheet` on `FlexCellmgExport` and `FlexCelPrintDocument`, allow printing all sheets on a workbook, keeping the "page n /m" headers or footers correlative. A method on `FlexCelPdfExport` allows the same thing. See the exporting demos.
- **AUTOFITTING SUPPORT.** Now you can autofit rows or columns with `XlsFile.AutoFitRow`, `XlsFile.AutoFitCol` and `XlsFile.AutoFitWorkbook` methods. On reports, the new tags `<#Row Height>` and `<#Column Width>` allow to change the row height / column width in a report, and to hide, show or autofit columns and rows. See the new Autofitting demo on the reports section.
- **VIRTUAL DATASETS.** Now you can use any data you like as source for your reports, not only Datasets. See the new Virtual DataSets demo.
- **SUPPORT FOR MANUAL FORMULAS IN REPORTS.** New report tags `<#Formula>` and `<#Ref>` allow replacing tags inside formulas, creating customizable formulas depending on the report data. (See the new "Manual Formulas" demo).
- **NEW ENTERFORMULAS PROPERTY ON FLEXCELREPORT.** Allows to enter any text starting with "=" as a formula instead of text.
- **FIXED BANDS ON REPORTS.** Now, by defining "`__band__FIXED`" ranges you can have bands that don't insert cells when moving down. See Fixed Forms With Datasets demo.
- **IMPROVED TAG REPLACE.** Report Tags are replaced now also on WordArt objects and Screen tips inside Hyperlinks.
- **SPLIT TAG ON REPORTS.** Allows splitting a datatable every n rows. See Split demo for more information.

- **USER TABLE TAG ON REPORTS.** Allows defining the datasets you want to use directly on the template. See User Tables demo for more information.
- **ADDED RECALCULATION FOR FUNCTIONS.** DCount, DSum, DAverage, DMin, DMax, DProduct, DCountA, DGet, DVar, DVarP, DStDev, DStDevP, Large, Small, MinA, MaxA, Var, VarP, VarA, VarPA, WeekDay, Product, SumSq, CountBlank, Roman, AverageA, Days360, FV, PV, NPV, DB, DDB, Syd, Sln, PMT, IMPT, PPMT, NPer, NormDist, NormsDist, LogNormDist, NormInv, NormsInv, LogInv, ExponDist, Poisson, Binomdist, NegBinomDist, HypGeomDist, Standardize, GeoMean, HarMean, Rank, GammaDist, Gammaln, GammaLn, ChiDist, ChiInv, IRR, MIRR, Rate, Areas, Rows, Columns, SumX2mY2, SumX2pY2, SumXmY2, Transpose, MMult, ZTest, ChiTest, Weibull, Kurt, Skew, AveDev, DevSQ, Steyx, Rsq, Pearson, Slope, Fisher, FisherInv, Median, Quartile, Percentile, Mode, Intercept. 210 functions supported (see the new SupportedFunctions.xls spreadsheet).
- **IMPROVED ARRAY FORMULA SUPPORT.** Now you can enter array formulas with the API (for example "{=Average(if(a1:a3=3;1))}"). And now FlexCel can calculate array formulas too, including array formulas that cover more than one cell.
- **ADDED BOOKMARKS TO PDF.** Now you can automatically add a bookmark on each sheet when exporting to pdf, or manually modify the bookmarks too. See the "Custom Preview" demo, on the button to export to pdf when "All Sheets" is selected.
- **IMPROVED RENDERING.** Added support to print and Export to PDF more than 70 Autoshapes: From block arrows to FlowCharts to basic shapes. See the new file SupportedAutoshapes.xls for more information.
- **IMPROVED RENDERING.** Now FlexCel can print and export to pdf basic WordArt text. Not all effects or types of WordArt are supported, but text is shown.
- **IMPROVED RENDERING.** Improved shadow support for autoshapes, and also gradient, texture, pattern and image fills supported.
- **ADDED MOVERANGE TO THE API.** Allows moving a range of cells in a sheet the same way Excel moves them, adapting all formula references as needed. The same as the existing InsertAndCopyRange and DeleteRange methods, this method is fully optimized to perform thousands of moves by second.
- **ADDED Find, Replace and Sort METHODS TO THE API.** While you could always do this by code, now it is easier to search inside, replace or sort a range.
- **ADDED XlsFile.Protection.WriteAccess PROPERTY.** Lets you know which user has a file opened in Excel. See the modified Getting started demo.
- **NEW READING FILES DEMO.** Showing how to import an Excel file into a DataSet / Datagrid.
- **IMPROVED MONO PDF SUPPORT.** Now FlexCel will try to automatically find the fonts when running on Linux. Also added a section on UsingFlexCelPdfExport.pdf explaining how to create pdf files from MONO.
- **1904 DATES SUPPORT.** Full support for 1904 based dates, allowing interoperability with xls files created on Apple computers.
- **PRECISION AS DISPLAYED SUPPORT.** Now recalculation in files where "Precision as Displayed" is true will honor that setting.

- **IMPROVED PERFORMANCE.** Performance improvements in Reporting, rendering and the API.
- **OPTIMIZED PDF FILE SIZE.** Now Pdfs are smaller.
- **IMPROVED HELP FILES.** Help files are now created with SandCastle, and can be integrated inside the VS IDE.
- **BUG FIXES.** Small bug fixes.

New in v 3.7.0.0 - December 2005

- **SUPPORT FOR READING PXL (POCKET EXCEL) FILES.** Now you can read both Pxl 1.0 and 2.0 files.
- **COMPACT FRAMEWORK 2.0 SUPPORT.** A new project FlexCelCF20 is included for CF 2.0. A new project for .NET 20 (FLEXCEL20.csproj) is included too.
- **DELPHI 2006 (.NET).** Support and demos.
- **SUPPORT FOR RENDERING MOST USED AUTOSHAPES.** Now rectangles, textboxes, ellipses, lines, triangles, arrows will be printed/previewed/exported to pdf. Support for semi-transparent fills, rich text inside, shadows, etc.
- **NEW FUNCTIONS SUPPORTED FOR RECALCULATION.** Cell, Lookup, Address, Fact, Combin, Permut, SinH, CosH, TanH, ASinH, ACosH, ATanH, Fixed, Dollar, Code, T, N, Hyperlink, StDev, StDevP, StDevA, StDevPA, Correl, Covar
- **NEW CALCULATED COLUMNS ADDED TO REPORTS.** Now you can use <#DataSet.#RowPos> and <#DataSet.#RowCount> inside reports to access the actual position and record count of a band.
- **IMPROVEMENTS ON PDF API.** Now you can write transparent text with the API, for example to superimpose a watermark to a FlexCel generated file. (See PDF Export demo)
- **IMPROVEMENTS ON RENDERING.** Watermark images on headers/footers now print right.
- **IMPROVEMENTS ON FORMULA PARSING.** There were some issues when entering complex formulas with the API, the formulas will be entered correctly but Excel would not calculate them.
- **NEW UTILITY API FUNCTIONS.** Added 2 overload to SetCellFormat allowing to set the format to a range of cells and change only one attribute on a range of cells (for example change only the line style, keeping the existing fonts)
- **NEW FEATURED DEMO.** FlexCel Image Explorer allows you to see and extract the images you have inside an Excel file.

New in v 3.6.0.0 - August 2005

- **DIRECT SQL ON REPORTS.** Now you can write sql against a connection on the server directly from the template. Note that for security reasons, if you don't add any connection to the report, no SQL can be executed. See "Direct SQL" demo for details.

- **RELATIONSHIPS ON THE TEMPLATE.** Now you can add data relationships directly on the template, allowing for example to "split" a dataset into master/detail and relate the 2 new datasets. See "Master Detail on one Table" demo for more information.
- **DISTINCT FILTER.** Now you can use a "DISTINCT()" filter to filter unique values on a dataset. See "Master Detail on one Table" demo for more information.
- **MERGE RANGE TAG.** With it you can conditionally merge cells on a band. See "Master Detail on one Table" demo for more information.
- **LOTS OF SMALL IMPROVEMENTS ON THE RENDERING ENGINE.** Now Conditional formats are printed and exported to pdf, cell patterns are exported to pdf and print better, Printer hard margins are considered, Subscripts/Superscript show on text, grouped and arbitrary angle rotated images are shown, etc.
- **PARAMETERS ON REPORT EXPRESSIONS.** Now you can use parameters when defining report expressions. For more information, see Expression Parameters demo.
- **IMPROVED CF SUPPORT.** Better support for CF packages (Registered version only)
- **NEW FEATURED DEMOS.** Showing how to access a web service from FlexCel or how to export an AdvWebGrid. Modified the Print/Preview demo to show how to export Multipage tiff files.
- **NEW DEMO ON HOW TO DIRECTLY OPEN THE GENERATED FILES.** The "Getting Started" and "Getting Started Reports" demos have been modified to show how to directly open the generated files without asking the user to save the file. (note that for this to work, the user must have excel on his machine).
- **FLEXCEL API.** New methods FreezePanes and GetFrozenPanes to freeze/unfreeze or get information about frozen panes. New methods SplitWindow / GetSplitWindow to split a sheet.
- **PERFORMANCE IMPROVEMENTS.** Almost 30% faster on some tests.
- **IMPROVED FRAMEWORK 2.0 SUPPORT.** Now when FRAMEWORK20 symbol is defined the new compression classes on System.IO.Compression will be used.
- **BUG FIX.** A call to StringFormat.SetMeasurableRanges could cause a deadlock when running hundreds of threads at the same time.

New in v 3.5.0.0 - May 2005

- **NATIVE PDF EXPORT.** Now you can export your reports to pdf, all on 100% native and managed code, without needing to have Excel or Adobe Acrobat installed.
- **OPTIMIZED PRINTING ENGINE.** Much faster and with lots of new features. (like repeating rows/columns, brightness/contrast adjustments on images, printing column and row headers, printing different types of borders and much more)

- **IMPROVED RECALCULATION.** More than 100 functions supported, and now much more like Excel. New methods include: Find, Proper, Concatenate, Exact, Rept, Clean, Search, Substitute, Text, Index, Match, RoundUp, RoundDown, Even, Odd, Subtotal, CountA, Value, Sumproduct. Also now supported intersect and union of ranges while recalculating, and arrays as parameters of formulas.
- **BASIC AUTOSHAPES SUPORT.** Now you can retrieve all their values and change their text. FlexCelReport will replace text inside Autoshapes also.
- **SUPPORT FOR READING/WRITING HEADER AND FOOTER IMAGES.** Now you can access images on headers and footers, and they will be printed and exported to pdf too. Note that images on headers and footers are only supported on Excel XP and newer, older Excel versions will open the file but will not display the graphics.
- **NEW COMPONENT TO PREVIEW WITH THUMBNAILS AND WITHOUT PRINTERS INSTALLED. NEW COMPONENT TO EXPORT TO IMAGES.** See Custom preview demo for more information on both.
- **CELL SELECTIONS.** Now you can read and write cell selections on a file. A new property on FlexCelReport, "ResetCellSelections", allows you to reset all selections on all sheets to "A1", so you do not need to worry about selection positions when saving the template.
- **SET NAMED RANGES.** Now you can add or modify named ranges, including ones as the print area.
- **OPTIMIZED FOR .NET 2.0.** If you define the "FRAMEWORK20" conditional define, a lot of 2.0 ONLY features (like Generics) will be used in places where they can improve performance.
- **REGULAR EXPRESSIONS ON REPORTS.** You can use the new <#REGEX()> tag to perform regular expression replaces on the reports. See "Regular Expressions" demo for more information.
- **COMPACT FRAMEWORK ASSEMBLIES.** While you can still use the same dll for both compact and full framework (as before) now we include a special FlexCelCF solution that will create an assembly specifically targeted to CF.
- **CODENAMES.** Now you can read the codenames of the sheets. This is useful because codenames never change, while sheet names can be changed by the user.
- **NEW METHODS.** ConvertFormulasToValues allows you to quickly remove formulas form a sheet. RecalcAndVerify() allows to verify you are using supported functions on your template. (See Verify Recalc demo)
- **HYPERLINK SUPPORT ENHANCED.** Now empty hyperlinks on report will not show, and also the syntax for tags is changed to "*.tag.*" for Excel2003 compatibility. (old <.tags> still work, but it is recommended to use the new syntax for new development)

New in v 3.1.0.1 - January 2005

- **FORMULA RECALCULATION.** Now most used formulas are recalculated before saving the files. (If the new RecalcMode property is not manual). So you can see formula results on any Excel viewer, such as Microsoft Excel viewer or FlexCelPrintDocument.

- **FOUR RECALCULATION MODES.** Recalculation can be manual (similar to v3.0), Forced (always recalculate before saving), Smart (Recalculate before saving only if file has been modified) or OnEveryChange (recalculate after changing any cell). Also an new RecalcForced property allows you to specify Excel not to recalculate on open.
- **PROTECTION AND ENCRYPTION.** There is a new property "Protection" on XlsFile that allows to both read and write protected and encrypted xls files. See Protect demo for more details.
- **ENHANCED DELPHI.NET SUPPORT.** Added a new demo with Delphi.Net, and now all BDP datatypes are supported.
- **NEW FUNCTIONS FOR REPORTING.** Now you can use Sum, Average, Round, Abs, Ceiling, Floor, Exp, Int, Ln, Log, Log10, Pi, Power, Rand, Sign, Sqrt, Trunc, Count, Radians, Degrees, Sin, Cos, Tan, ASin, ACos, ATan, ATan2, SumIf, CountIf, Date, DateValue, Day, Month, Year, Time, TimeValue, Hour, Minute, Second, Now, Today, Error.Type, IsBlank, IsErr, IsError, IsLogical, IsNA, IsNonText, IsNumber, IsREF, IsText, Type, Na, Choose, Offset, HLookup and VLookup when creating expressions for a report. All of those functions will be evaluated when recalculating formulas too. For a complete list of supported functions, see EndUserGuide.pdf, "Evaluating Expressions"
- **SMALL PERFORMANCE IMPROVEMENTS.** FlexCel 3.1 is even faster than 3.0. Not much, because 3.0 was quite fast, but a little faster.

New in v 3.0.0.5 - August 2004

- **First public FlexCel for .NET version.** Based in the FlexCel for Delphi 2.x version, this initial version includes reports and an extensive API to read or write xls files.

Supported Excel functions

This is a list of all the standard Excel functions up to Excel 2021.

Functions marked with ~~strikeout~~ are not implemented yet. To know if a specific spreadsheet has all the formulas supported by FlexCel, you can use the [Validate Recalc](#) demo

Formulas on column "Array Enabled" mean that you can enter them inside an Array formula with a range argument (for example "if(a1:a10 < 3;...;....)") All formulas can be used inside Array Formulas, but only the ones on this list can be used with array arguments where they would expect a single value. By "Array formulas" we mean formulas you enter with Shift-Ctrl-Enter in Excel

Supported built-in functions

- **Total built-in functions:** 455
- **Implemented:** 353 (78%)

Built-in functions in 2003

Financial

Name	Implemented	Array enabled
DB	✓	
DDB	✓	
FV	✓	
IPMT	✓	
IRR	✓	
ISPMT		
MIRR	✓	
NPER	✓	
NPV	✓	
PMT	✓	
PPMT	✓	
PV	✓	
RATE	✓	
SLN	✓	

Name	Implemented	Array enabled
SYD	✓	
VDB		

Date & Time

Name	Implemented	Array enabled
DATE	✓	✓
DATEVALUE	✓	✓
DAY	✓	✓
DAYS360	✓	
HOUR	✓	✓
MINUTE	✓	✓
MONTH	✓	✓
NOW	✓	✓
SECOND	✓	✓
TIME	✓	✓
TIMEVALUE	✓	✓
TODAY	✓	✓
WEEKDAY	✓	✓
YEAR	✓	✓
DATEDIF	✓	✓

Math & Trig

Name	Implemented	Array enabled
ABS	✓	✓
ACOS	✓	✓
ACOSH	✓	✓
ASIN	✓	✓
ASINH	✓	✓
ATAN	✓	✓

Name	Implemented	Array enabled
ATAN2	✓	✓
ATANH	✓	✓
CEILING	✓	✓
COMBIN	✓	✓
COS	✓	✓
COSH	✓	✓
DEGREES	✓	✓
EVEN	✓	✓
EXP	✓	✓
FACT	✓	✓
FLOOR	✓	✓
INT	✓	✓
LN	✓	✓
LOG	✓	
LOG10	✓	✓
MDETERM		
MINVERSE		
MMULT	✓	✓
MOD	✓	✓
ODD	✓	✓
PI	✓	✓
POWER	✓	✓
PRODUCT	✓	✓
RADIANS	✓	✓
RAND	✓	✓
ROMAN	✓	
ROUND	✓	✓
ROUNDDOWN	✓	✓
ROUNDUP	✓	✓

Name	Implemented	Array enabled
SIGN	✓	✓
SIN	✓	✓
SINH	✓	✓
SQRT	✓	✓
SUBTOTAL	✓	
SUM	✓	✓
SUMX2MY2	✓	✓
SUMX2PY2	✓	✓
SUMXMY2	✓	✓
SUMIF	✓	✓
SUMPRODUCT	✓	✓
SUMSQ	✓	✓
TAN	✓	✓
TANH	✓	✓
TRUNC	✓	

Statistical

Name	Implemented	Array enabled
AVEDEV	✓	✓
AVERAGE	✓	✓
AVERAGEA	✓	✓
BETADIST		
BETAINV		
BINOMDIST	✓	
CHIDIST	✓	
CHIINV	✓	
CHITEST	✓	✓
CONFIDENCE	✓	
CORREL	✓	

Name	Implemented	Array enabled
COUNT	✓	✓
COUNTA	✓	✓
COUNTBLANK	✓	✓
COUNTIF	✓	✓
COVAR	✓	
CRITBINOM		
DEVSQ	✓	
EXPONDIST	✓	
FDIST		
FINV		
FISHER	✓	✓
FISHERINV	✓	✓
FORECAST		
FREQUENCY	✓	✓
FTEST		
GAMMADIST	✓	
GAMMAINV	✓	
GAMMALN	✓	✓
GEOMEAN	✓	
GROWTH		
HARMEAN	✓	
HYPGEOMDIST	✓	
INTERCEPT	✓	
KURT	✓	
LARGE	✓	✓
LINEST		
LOGEST		
LOGINV	✓	
LOGNORMDIST	✓	

Name	Implemented	Array enabled
MAX	✓	✓
MAXA	✓	✓
MEDIAN	✓	✓
MIN	✓	✓
MINA	✓	✓
MODE	✓	✓
NEGBINOMDIST	✓	
NORMDIST	✓	
NORMINV	✓	
NORMSDIST	✓	✓
NORMSINV	✓	✓
PEARSON	✓	✓
PERCENTILE	✓	✓
PERCENTRANK	✓	
PERMUT	✓	✓
POISSON	✓	
PROB		
QUARTILE	✓	✓
RANK	✓	
RSQ	✓	✓
SKEW	✓	✓
SLOPE	✓	✓
SMALL	✓	✓
STANDARDIZE	✓	✓
STDEV	✓	
STDEVA	✓	
STDEVP	✓	
STDEVPA	✓	
STEYX	✓	✓

Name	Implemented	Array enabled
TDIST		
TINV		
TREND		
TRIMMEAN		
TTEST		
VAR	✓	
VARA	✓	
VARP	✓	
VARPA	✓	
WEIBULL	✓	
ZTEST	✓	

Lookup & Reference

Name	Implemented	Array enabled
ADDRESS	✓	✓
AREAS	✓	
CHOOSE	✓	✓
COLUMN	✓	✓
COLUMNS	✓	✓
HLOOKUP	✓	
HYPERLINK	✓	
INDEX	✓	✓
INDIRECT	✓	✓
LOOKUP	✓	
MATCH	✓	✓
OFFSET	✓	✓
ROW	✓	✓
ROWS	✓	✓
TRANSPOSE	✓	✓

Name	Implemented	Array enabled
VLOOKUP	✓	

Database

Name	Implemented	Array enabled
DAVERAGE	✓	
DCOUNT	✓	
DCOUNTA	✓	
DGET	✓	
DMAX	✓	
DMIN	✓	
DPRODUCT	✓	
DSTDEV	✓	
DSTDEVP	✓	
DSUM	✓	
DVAR	✓	
DVARP	✓	
GETPIVOTDATA		

Text

Name	Implemented	Array enabled
ASC	✓	✓
BAHTTEXT	✓	✓
CHAR	✓	✓
CLEAN	✓	✓
CODE	✓	✓
CONCATENATE	✓	✓
DOLLAR	✓	
EXACT	✓	
FIND	✓	

Name	Implemented	Array enabled
FIXED	✓	
#S		
LEFT	✓	✓
LEN	✓	✓
LOWER	✓	✓
MID	✓	✓
PHONETIC		
PROPER	✓	✓
REPLACE	✓	✓
REPT	✓	
RIGHT	✓	✓
SEARCH	✓	✓
SUBSTITUTE	✓	
T	✓	
TEXT	✓	
TRIM	✓	✓
UPPER	✓	✓
VALUE	✓	✓

Logical

Name	Implemented	Array enabled
AND	✓	✓
FALSE	✓	✓
IF	✓	✓
NOT	✓	✓
OR	✓	✓
TRUE	✓	✓

Information

Name	Implemented	Array enabled
CELL	✓	
ERROR.TYPE	✓	✓
INFO		
ISBLANK	✓	✓
ISERR	✓	✓
ISERROR	✓	✓
ISLOGICAL	✓	✓
ISNA	✓	✓
ISNONTEXT	✓	✓
ISNUMBER	✓	✓
ISREF	✓	
ISTEXT	✓	✓
N	✓	
NA	✓	✓
TYPE	✓	✓

- **Total:** 238
- **Implemented:** 214 (90%)

Added functions in Excel 2007

Name	Implemented	Array enabled
ACCRINT	✓	
ACCRINTM	✓	
AMORDEGRC		
AMORLINC		
AVERAGEIF	✓	✓
BESSELI		
BESSELJ		
BESSELK		
BESSELY		

Name	Implemented	Array enabled
BIN2DEC	✓	
BIN2HEX	✓	
BIN2OCT	✓	
COMPLEX		
CONVERT	✓	
COUPDAYBS	✓	
COUPDAYS	✓	
COUPDAYSNC	✓	
COUPNCD	✓	
COUPNUM	✓	
COUPPCD	✓	
CUMIPMT	✓	
CUMPRINC	✓	
DEC2BIN	✓	
DEC2HEX	✓	
DEC2OCT	✓	
DELTA	✓	
DISC		
DOLLARDE	✓	
DOLLARFR	✓	
DURATION	✓	
EDATE	✓	
EFFECT	✓	
EOMONTH	✓	
ERF		
ERFC		
FACTDOUBLE	✓	
FVSCHEDULE		
GCD	✓	✓

Name	Implemented	Array enabled
GESTEP	✓	
HEX2BIN	✓	
HEX2DEC	✓	
HEX2OCT	✓	
IFERROR	✓	✓
IMABS		
IMAGINARY		
IMARGUMENT		
IMCONJUGATE		
IMCOS		
IMDIV		
IMEXP		
IMLN		
IMLOG10		
IMLOG2		
IMPOWER		
IMPRODUCT		
IMREAL		
IMSIN		
IMSQRT		
IMSUB		
IMSUM		
INTRATE		
ISEVEN	✓	
ISODD	✓	
LCM	✓	✓
MDURATION	✓	
MROUND	✓	
MULTINOMIAL	✓	✓

Name	Implemented	Array enabled
NETWORKDAYS	✓	
NOMINAL	✓	
OCT2BIN	✓	
OCT2DEC	✓	
OCT2HEX	✓	
ODDFPRICE		
ODDFYIELD		
ODDLPRICE		
ODDLYIELD		
PRICE		
PRICEDISC		
PRICEMAT		
QUOTIENT	✓	
RANDBETWEEN	✓	
RECEIVED		
SERIESSUM	✓	
SQRTPI	✓	
TBILLEQ		
TBILLPRICE		
TBILLYIELD		
WEEKNUM	✓	
WORKDAY	✓	
XIRR	✓	
XNPV	✓	
YEARFRAC	✓	
YIELDDISC		
YIELDMAT		

- **Total:** 94
- **Implemented:** 52 (55%)

Added functions in Excel 2010

Name	Implemented	Array enabled
NETWORKDAYS.INTL	✓	
WORKDAY.INTL	✓	
AGGREGATE	✓	
CEILING.PRECISE	✓	✓
ISO.CEILING	✓	✓
CHISQ.DIST		
CHISQ.INV		
CONFIDENCE.T		
COVARIANCE.S		
ERF.PRECISE		
ERFC.PRECISE		
F.DIST		
F.INV		
FLOOR.PRECISE	✓	
GAMMALN.PRECISE		
MODE.MULT		
PERCENTILE.EXC	✓	
PERCENTRANK.EXC		
QUARTILE.EXC	✓	
RANK.AVG	✓	✓
T.DIST		
T.INV		

- **Total:** 22
- **Implemented:** 9 (41%)

Added functions in Excel 2013

Name	Implemented	Array enabled
DAYS	✓	✓
ISOWEEKNUM	✓	✓

Name	Implemented	Array enabled
BITAND	✓	✓
BITLSHIFT	✓	✓
BITOR	✓	✓
BITRSHIFT	✓	✓
BITXOR	✓	✓
IMCOSH		
IMCOT		
IMCSC		
IMCSCH		
IMSEC		
IMSECH		
IMSINH		
IMTAN		
PDURATION	✓	✓
RRI	✓	✓
ISFORMULA	✓	✓
SHEET	✓	
SHEETS	✓	✓
IFNA	✓	✓
XOR	✓	✓
FORMULATEXT	✓	✓
ACOT	✓	✓
ACOTH	✓	✓
ARABIC	✓	✓
BASE	✓	✓
CEILING.MATH	✓	✓
COMBINA	✓	✓
COT	✓	✓
COTH	✓	✓

Name	Implemented	Array enabled
CSC	✓	✓
CSCH	✓	✓
DECIMAL	✓	✓
FLOOR.MATH	✓	✓
ISO.CEILING	✓	✓
MUNIT	✓	✓
SEC	✓	✓
SECH	✓	✓
BINOM.DIST.RANGE		
GAMMA		
GAUSS		
PERMUTATIONA	✓	✓
PHI		
SKEW.P		
NUMBERVALUE	✓	✓
UNICHAR	✓	✓
UNICODE	✓	✓
ENCODEURL		
FILTERXML		
WEBSERVICE		

- **Total:** 51
- **Implemented:** 35 (69%)

Added functions in Excel 2016

Name	Implemented	Array enabled
FORECAST.LINEAR		
FORECAST.ETS		
FORECAST.ETS.SEASONALITY		
FORECAST.ETS.CONFINT		
FORECAST.ETS.STAT		

Name	Implemented	Array enabled
SWITCH	✓	✓

- **Total:** 6
- **Implemented:** 1 (17%)

Added functions in Excel 2019

Name	Implemented	Array enabled
AVERAGEIFS	✓	✓
COUNTIFS	✓	✓
SUMIFS	✓	✓
TEXTJOIN	✓	✓
CONCAT	✓	✓
IFS	✓	✓
MINIFS	✓	✓
MAXIFS	✓	✓

- **Total:** 8
- **Implemented:** 8 (100%)

Added functions in Office 2021

Name	Implemented	Array enabled
FILTER	✓	✓
RANDARRAY	✓	✓
SEQUENCE	✓	✓
SORT	✓	✓
SORTBY	✓	✓
UNIQUE	✓	✓
XLOOKUP	✓	✓
XMATCH	✓	✓
LET	✓	✓
SINGLE	✓	✓

- **Total:** 10

• **Implemented:** 10 (100%)

Added functions in Office 365

Name	Implemented	Array enabled
ARRAYTOTEXT	✓	✓
VALUETOTEXT	✓	✓
BYCOL	✓	✓
BYROW	✓	✓
FIELDVALUE		
ISOMITTED	✓	✓
LAMBDA	✓	✓
MAKEARRAY	✓	✓
MAP	✓	✓
REDUCE	✓	✓
SCAN	✓	✓
STOCKHISTORY		
TEXTBEFORE	✓	✓
TEXTAFTER	✓	✓
TEXTSPLIT	✓	✓
VSTACK	✓	✓
HSTACK	✓	✓
TOROW	✓	✓
TOCOL	✓	✓
WRAPROWS	✓	✓
WRAPCOLS	✓	✓
TAKE	✓	✓
DROP	✓	✓
CHOOSEROWS	✓	✓
CHOOSECOLS	✓	✓
EXPAND	✓	✓

• **Total:** 26

- **Implemented:** 24 (92%)

Renamed functions

This is a list of the functions renamed in Excel.

- **Supported** means that FlexCel understands the function and you can add it with the API.
- **Implemented for recalc** means that FlexCel knows how to calculate the function.

NOTE

Normally, if a function renamed is implemented for recalc in FlexCel under the old name then it will also be implemented for recalc in the new name.

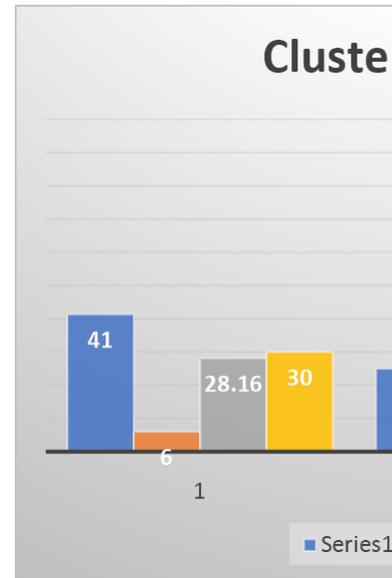
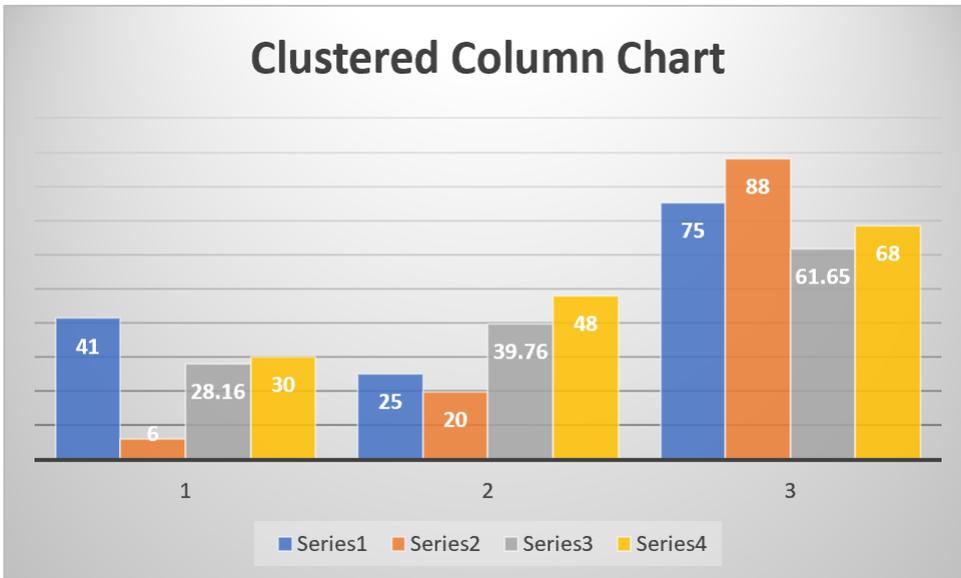
But some renamed functions like **Norm.S.Dist** have a different list of parameters. Those might not be implemented for recalc.

Renamed functions in Excel 2010

New name	Old name	Supported	Implemented for recalc
BETA.DIST	BETADIST	✓	
BETA.INV	BETAINV	✓	
BINOM.DIST	BINOMDIST	✓	✓
BINOM.INV	CRITBINOM	✓	
CHISQ.DIST.RT	CHIDIST	✓	✓
CHISQ.INV.RT	CHIINV	✓	✓
CHISQ.TEST	CHITEST	✓	✓
CONFIDENCE.NORM	CONFIDENCE	✓	✓
COVARIANCE.P	COVAR	✓	✓
EXPON.DIST	EXPONDIST	✓	✓
F.DIST.RT	FDIST	✓	
F.INV.RT	FINV	✓	
F.TEST	FTEST	✓	
GAMMA.DIST	GAMMADIST	✓	✓
GAMMA.INV	GAMMAINV	✓	✓
HYPGEOM.DIST	HYPGEOMDIST	✓	
LOGNORM.DIST	LOGNORMDIST	✓	
LOGNORM.INV	LOGINV	✓	✓

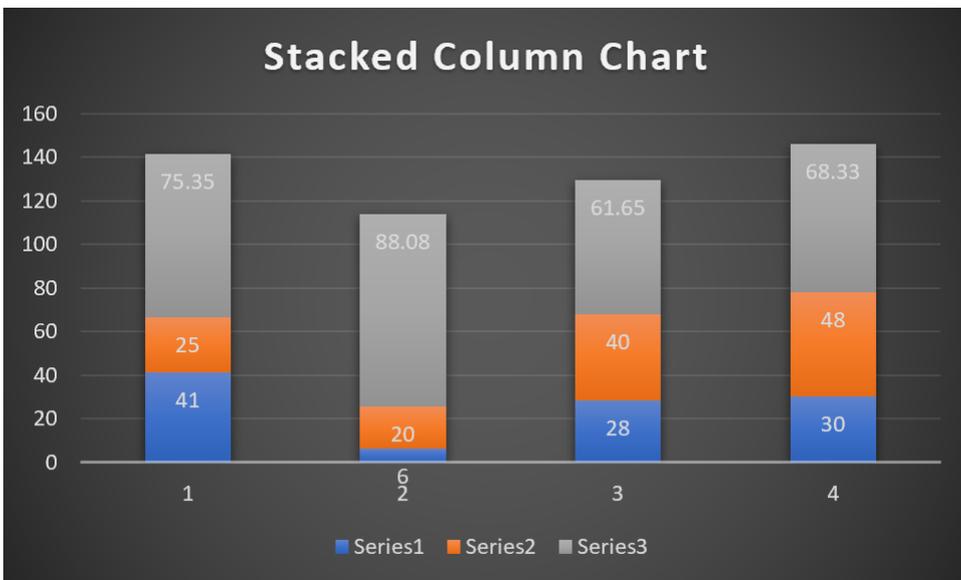
New name	Old name	Supported	Implemented for recalc
MODE.SNGL	MODE	✓	✓
NEGBINOM.DIST	NEGBINOMDIST	✓	
NORM.DIST	NORMDIST	✓	✓
NORM.INV	NORMINV	✓	✓
NORM.S.DIST	NORMSDIST	✓	
NORM.S.INV	NORMSINV	✓	✓
PERCENTILE.INC	PERCENTILE	✓	✓
PERCENTRANK.INC	PERCENTRANK	✓	✓
POISSON.DIST	POISSON	✓	✓
QUARTILE.INC	QUARTILE	✓	✓
RANK.EQ	RANK	✓	✓
STDEV.P	STDEVP	✓	✓
STDEV.S	STDEV	✓	✓
T.DIST.2T	TDIST	✓	
T.DIST.RT	TDIST	✓	
T.INV.2T	TINV	✓	
T.TEST	TTEST	✓	
VAR.P	VARP	✓	✓
VAR.S	VAR	✓	✓
WEIBULL.DIST	WEIBULL	✓	✓
Z.TEST	ZTEST	✓	✓

- **Total:** 39
- **Implemented:** 25 (64%)

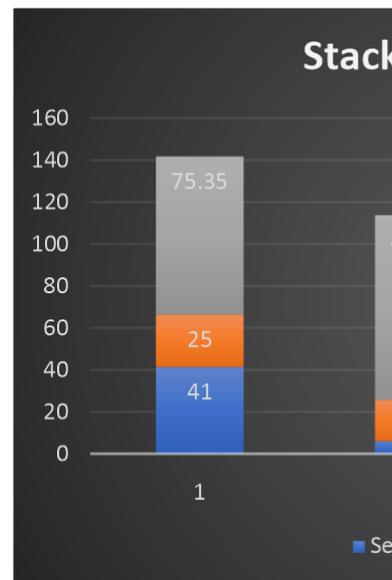


Stacked Column

Excel



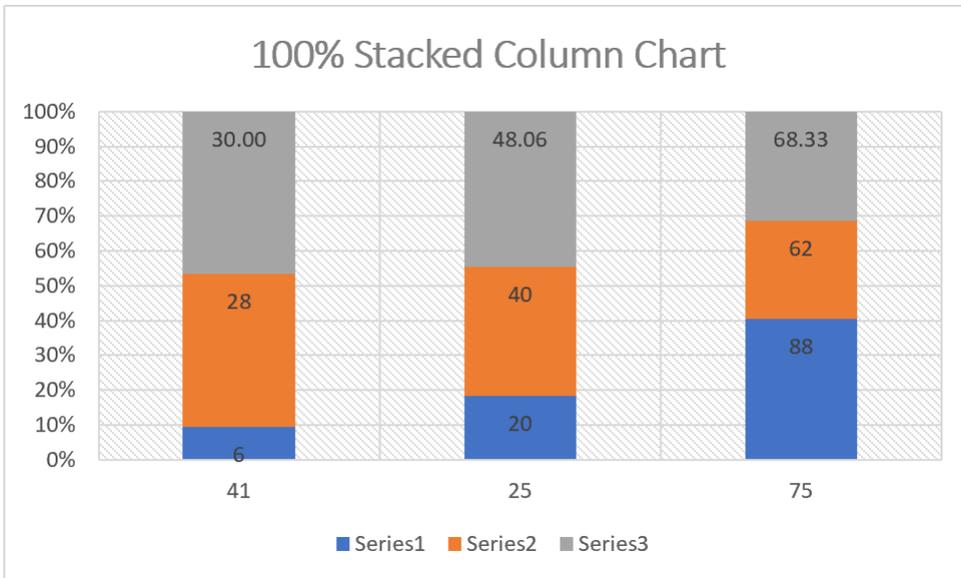
FlexCel



100% Stacked Column

Excel

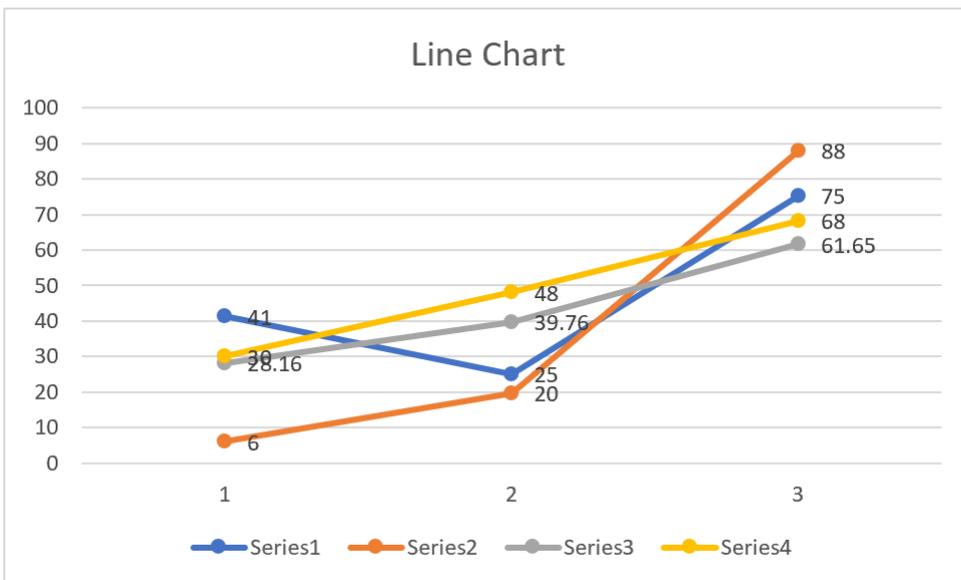
FlexCel



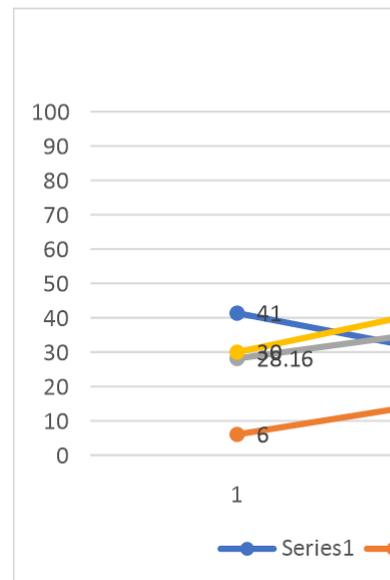
Line

Line

Excel



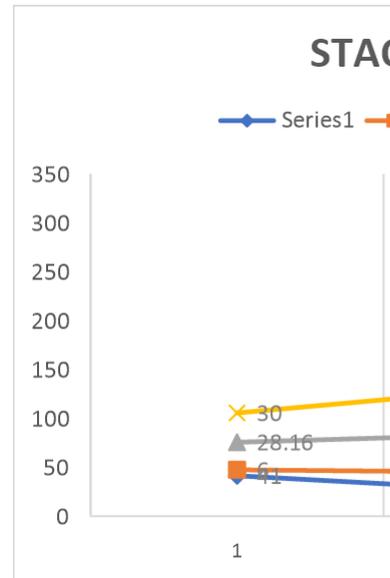
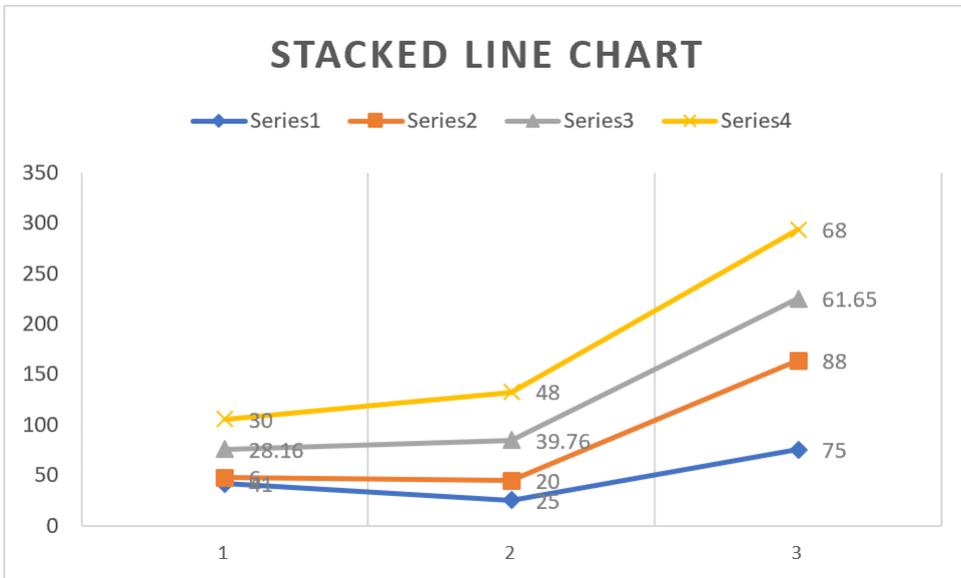
FlexCel



Stacked Line

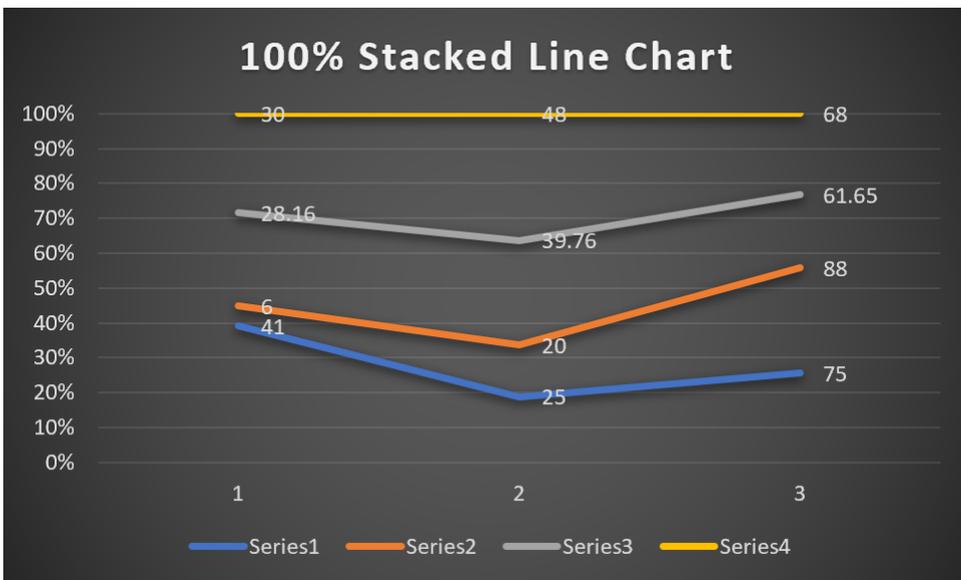
Excel

FlexCel

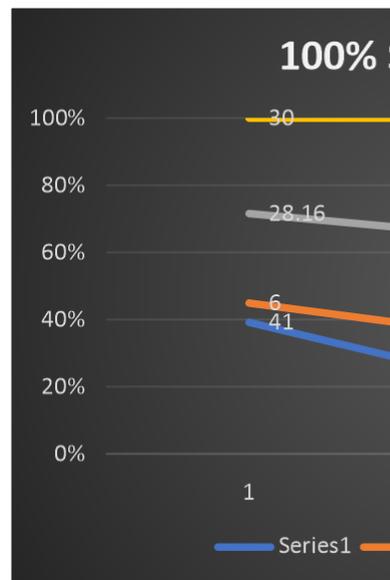


100% Stacked Line

Excel



FlexCel

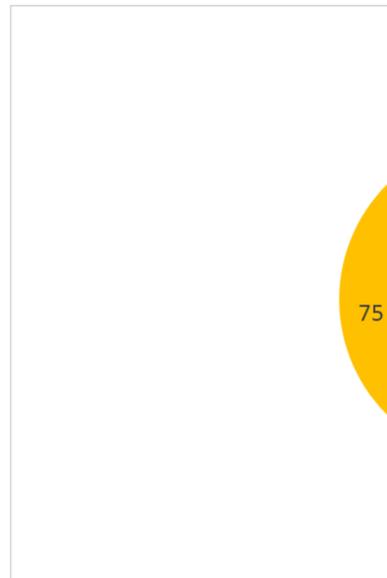
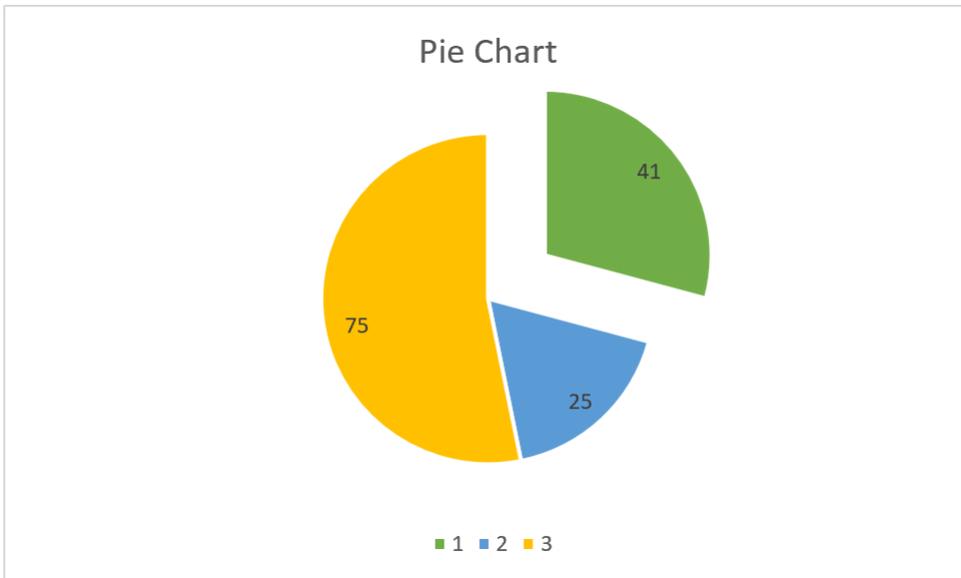


Pie

Pie

Excel

FlexCel



3-D Pie

Not Supported

Pie of Pie

Not Supported

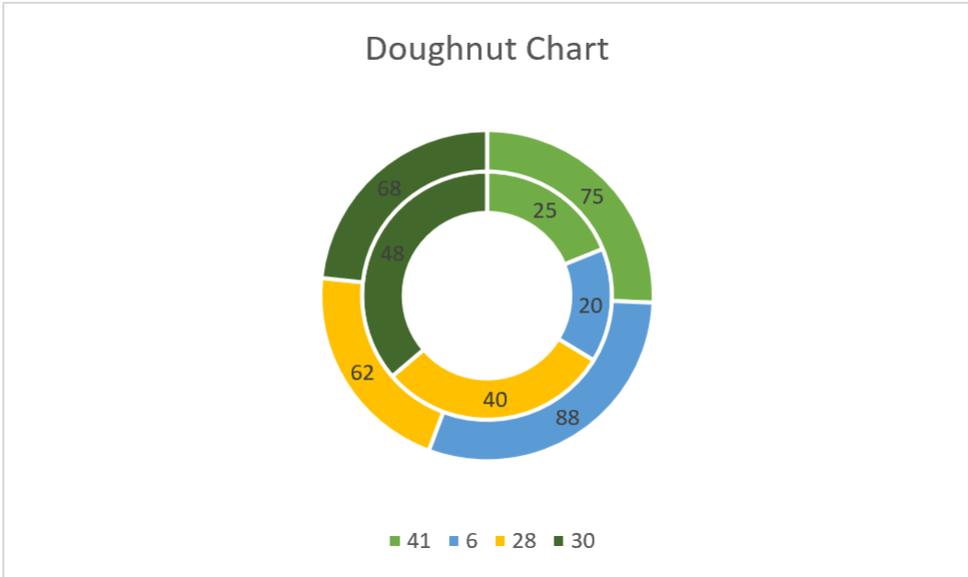
Bar of Pie

Not Supported

Doughnut

Excel

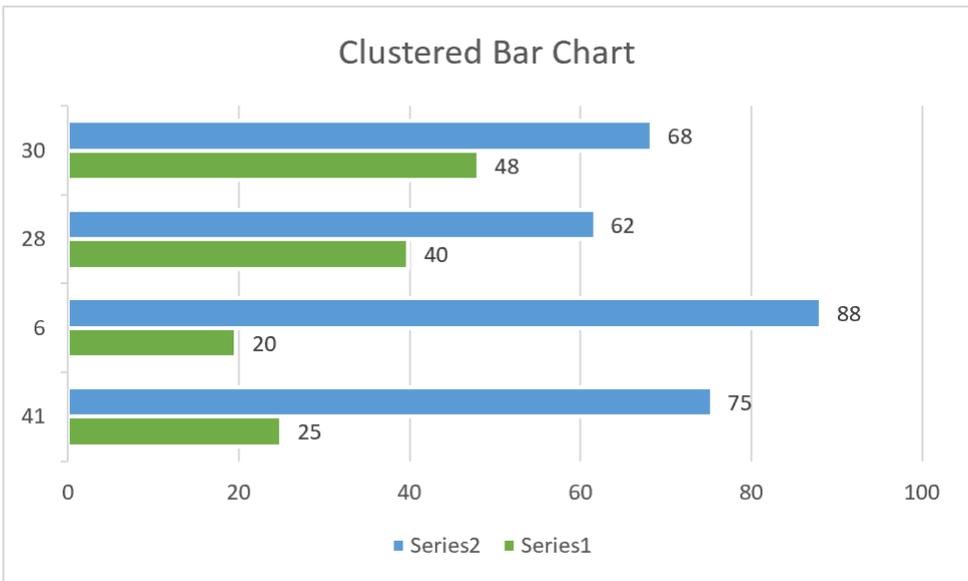
FlexCel



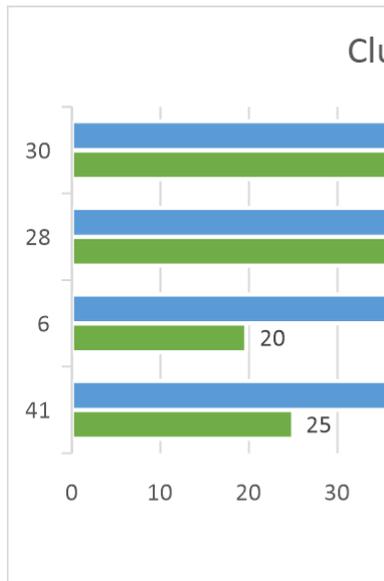
Bar

Clustered Bar

Excel



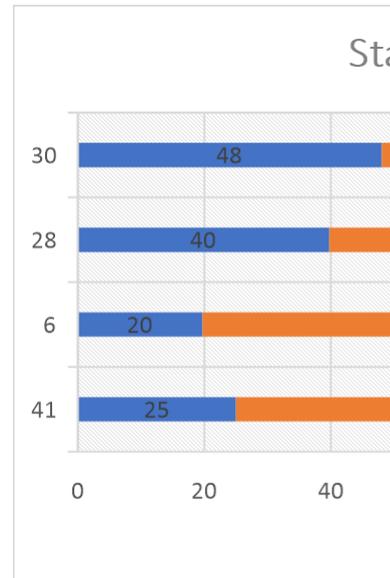
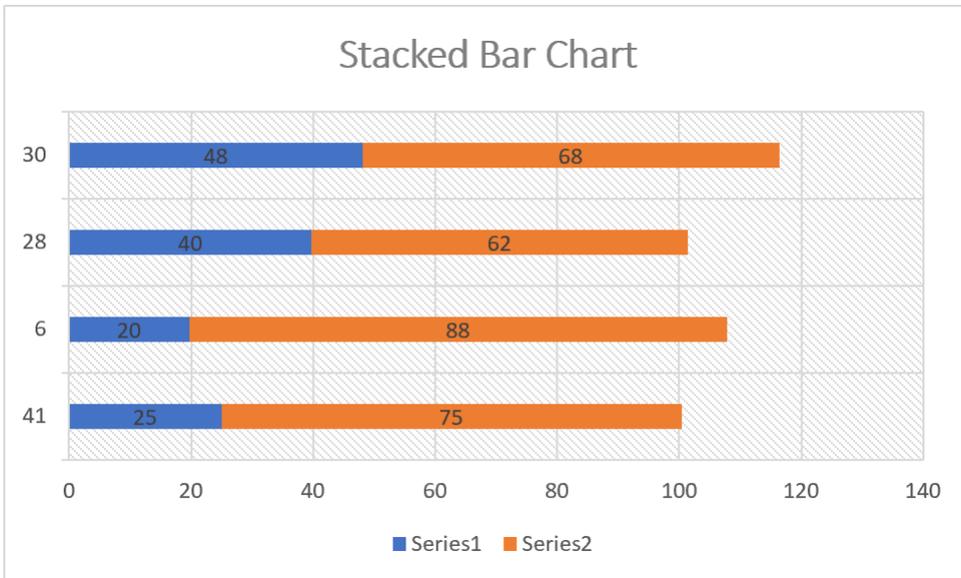
FlexCel



Stacked Bar

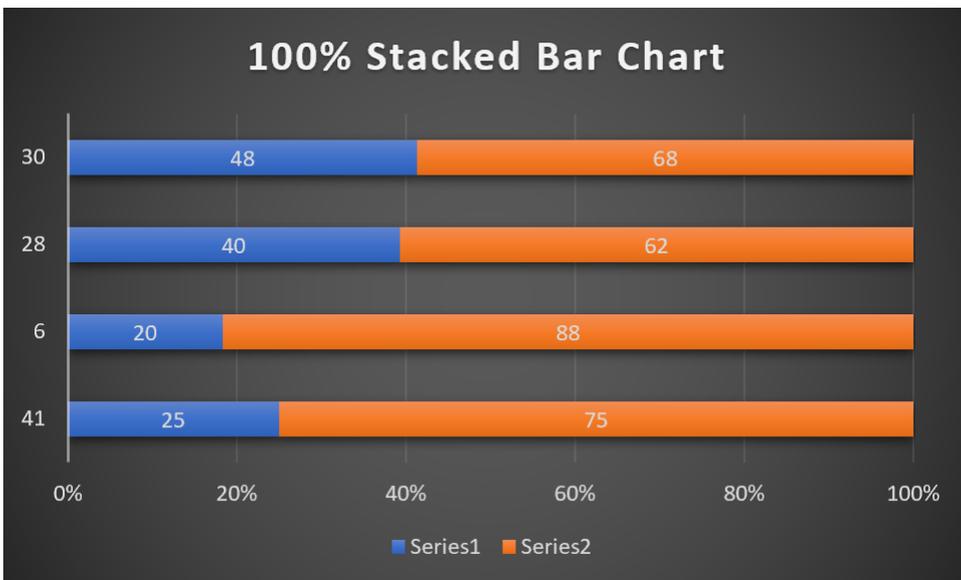
Excel

FlexCel

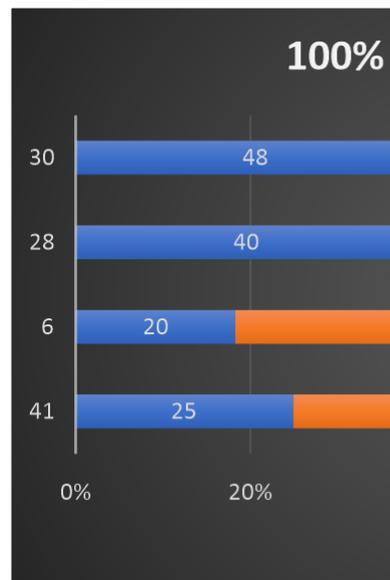


100% Stacked Bar

Excel



FlexCel

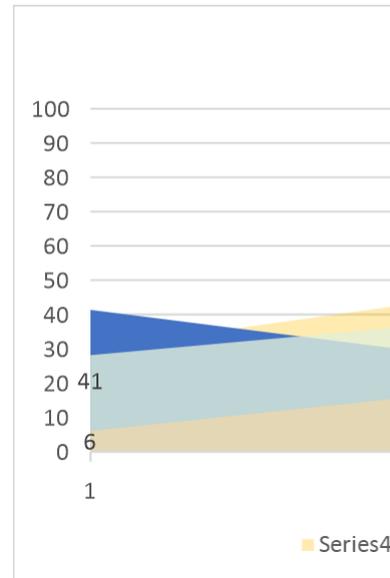
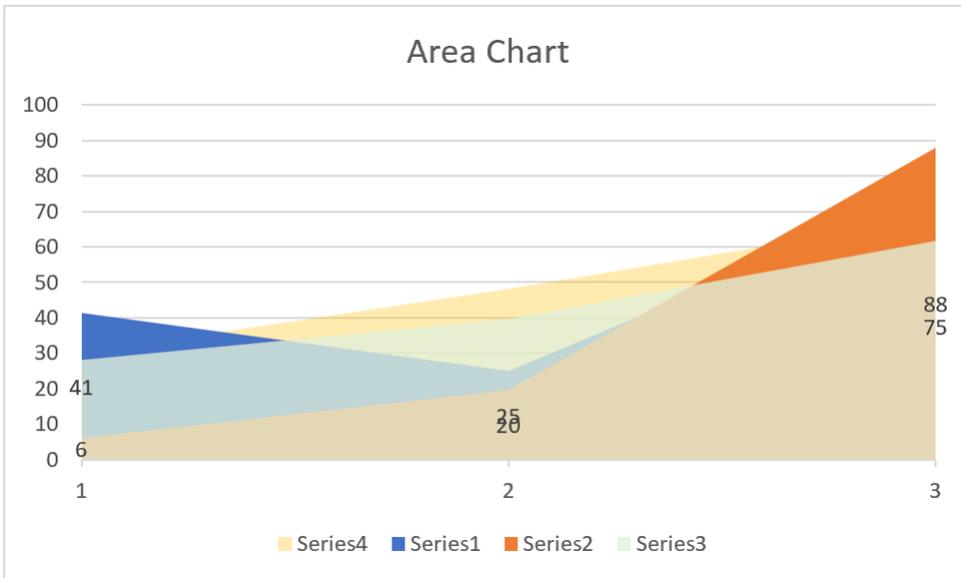


Area

Area

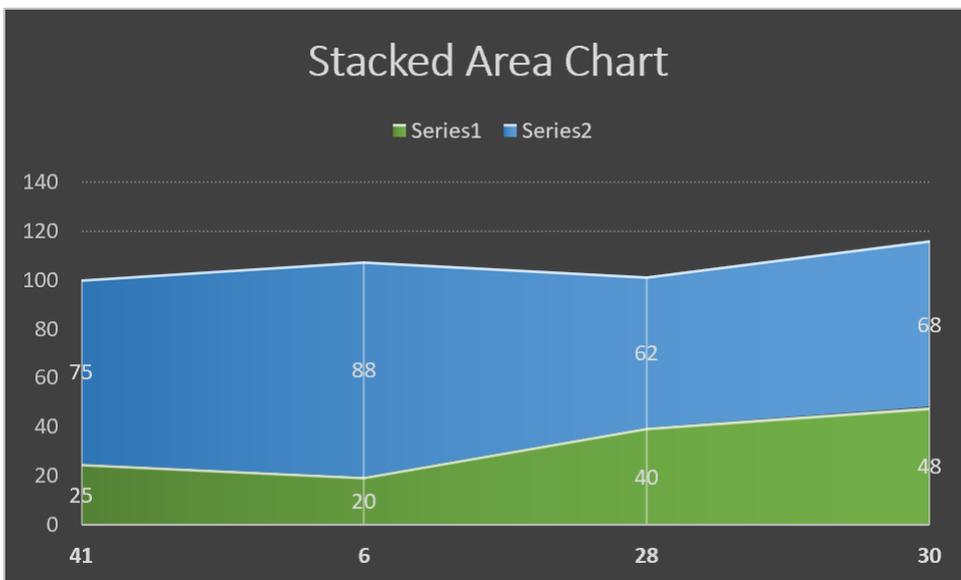
Excel

FlexCel

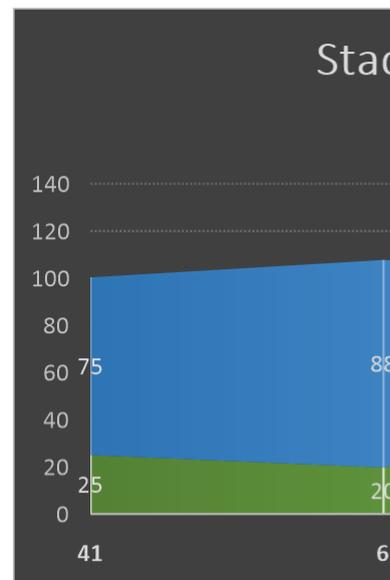


Stacked Area

Excel



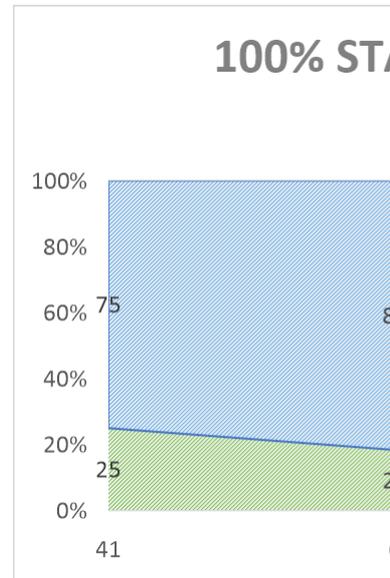
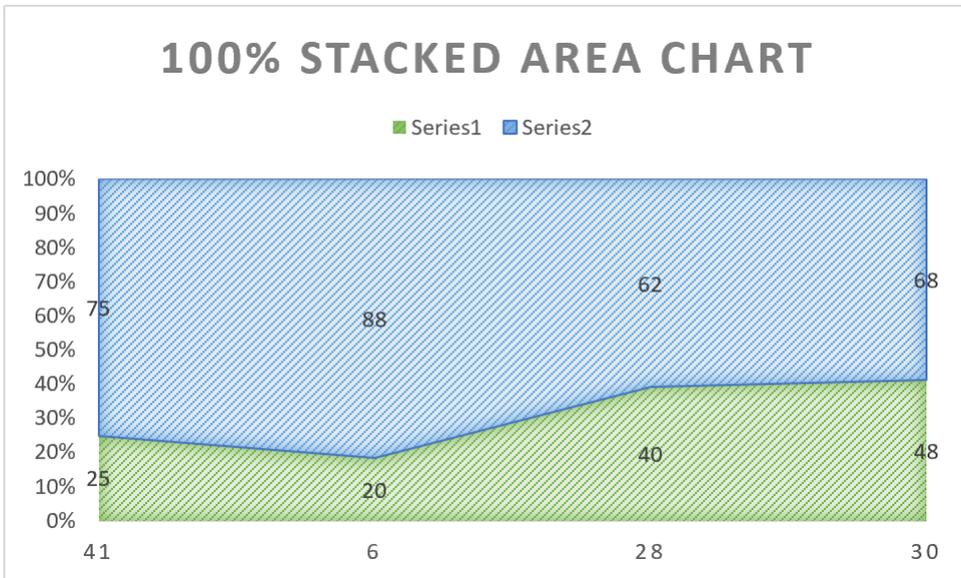
FlexCel



100% Stacked Area

Excel

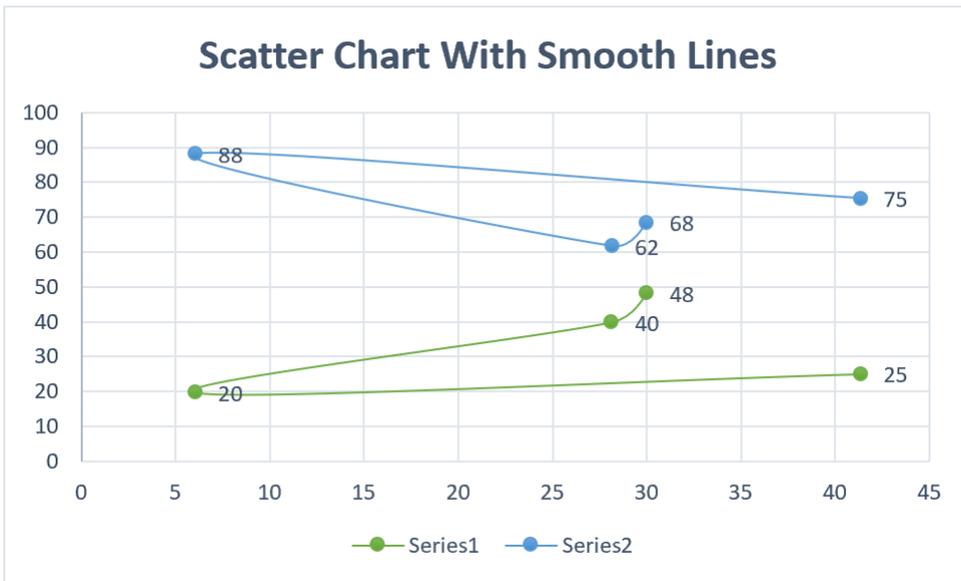
FlexCel



Scatter

Scatter with smooth lines

Excel



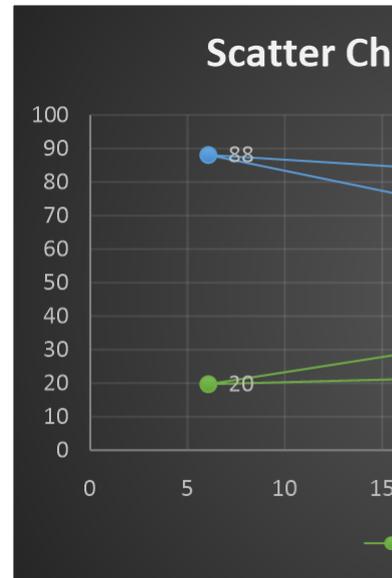
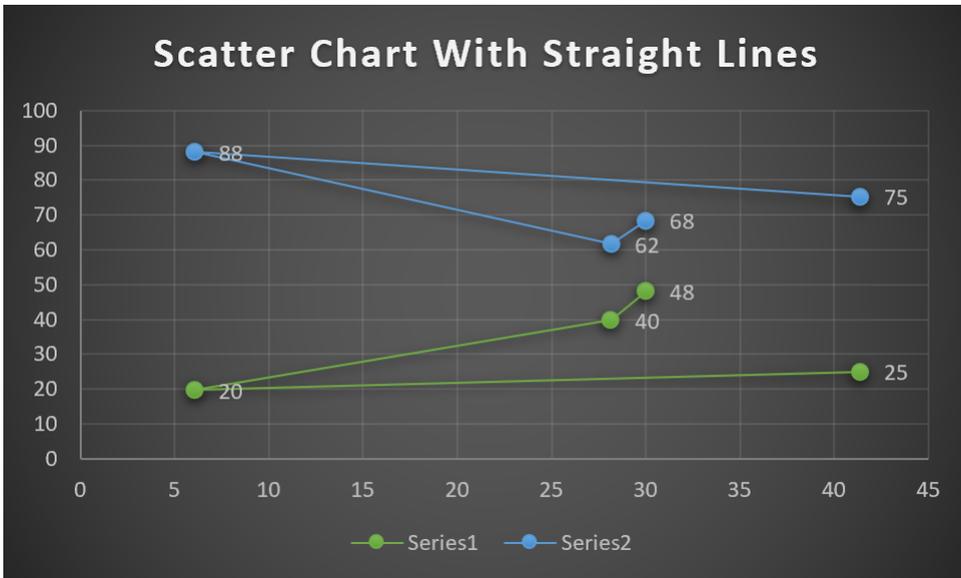
FlexCel



Scatter with straight lines

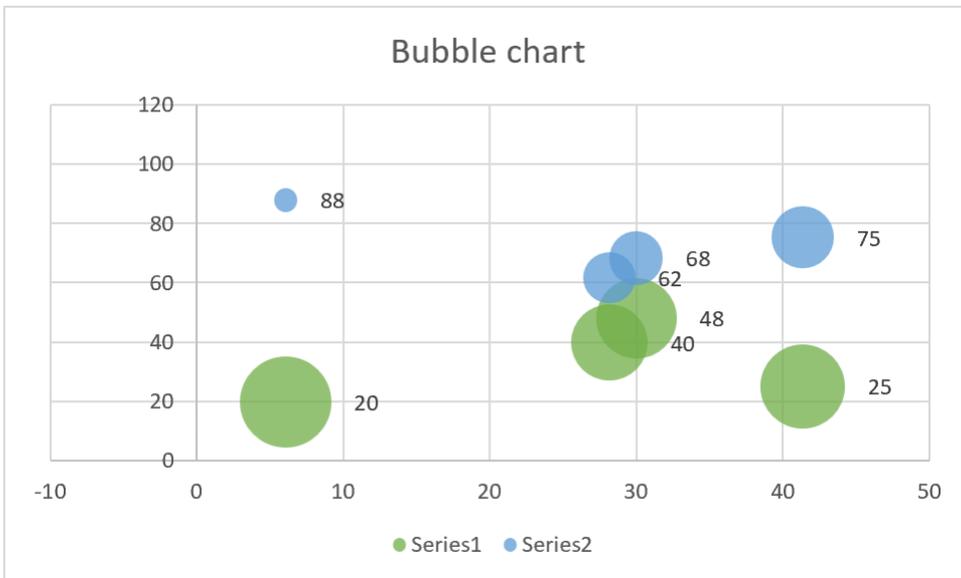
Excel

FlexCel

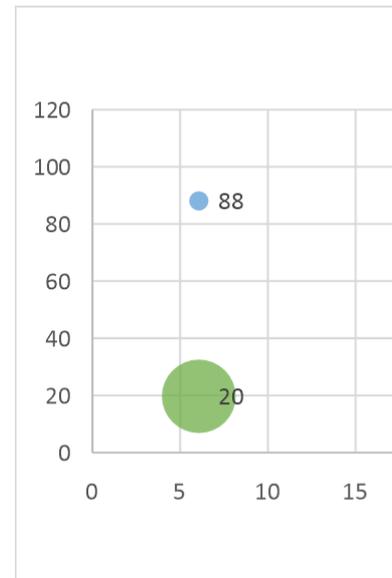


Bubble

Excel



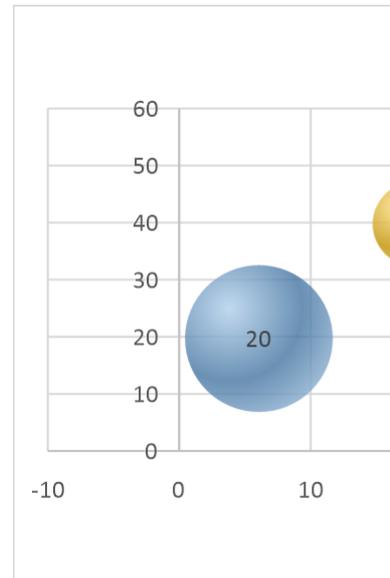
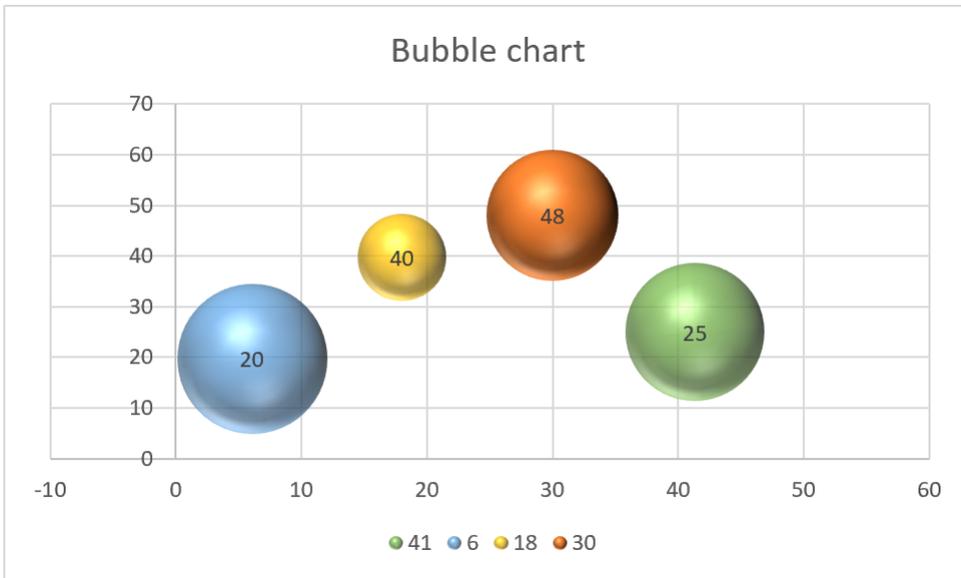
FlexCel



3D Bubble

Excel

FlexCel



Map

Not Supported

Stock

Not supported

Surface

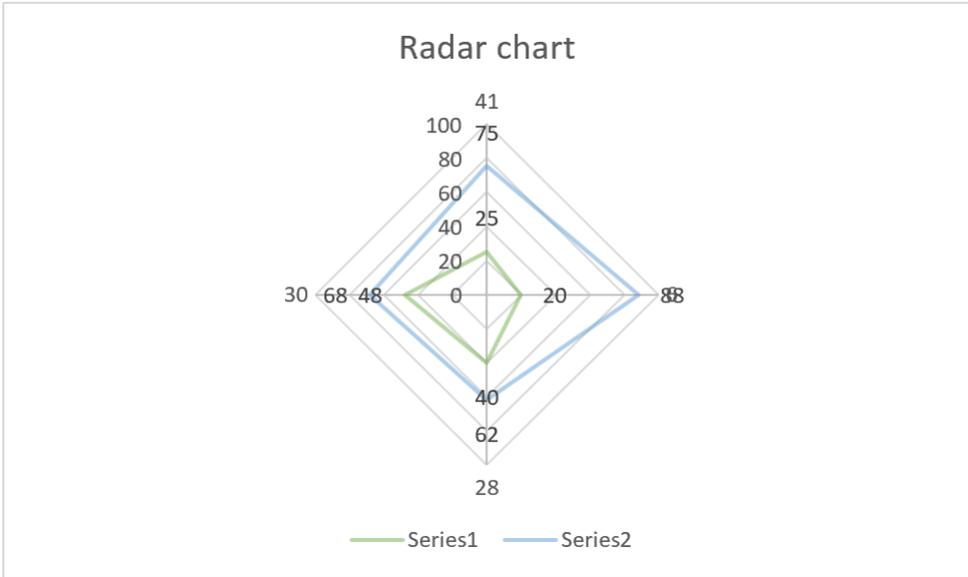
Not Supported

Radar

Radar

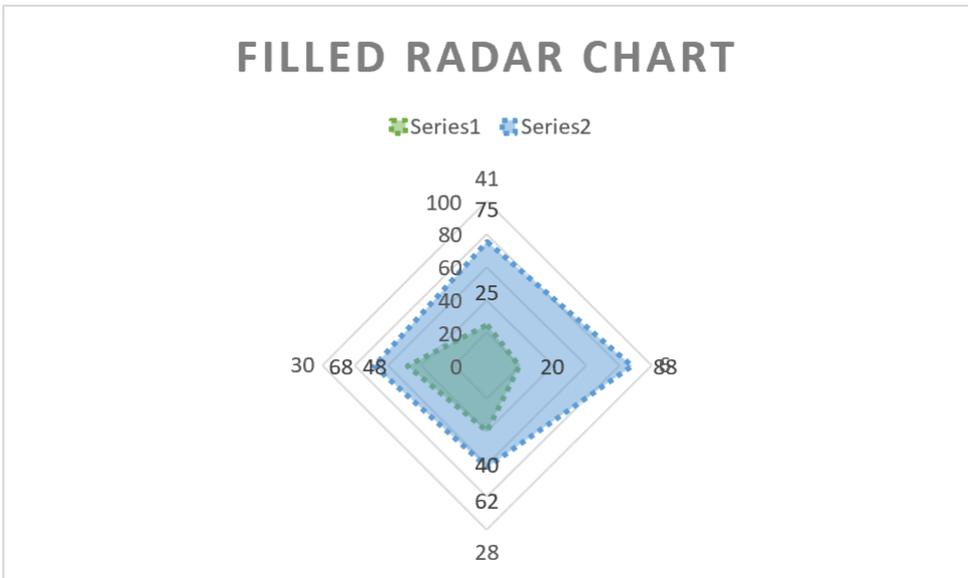
Excel

FlexCel



Filled Radar

Excel



FlexCel



Treemap

Not Supported

Sunburst

Not Supported

Histogram

Not Supported

Box & Whisker

Not Supported

Waterfall

Not Supported

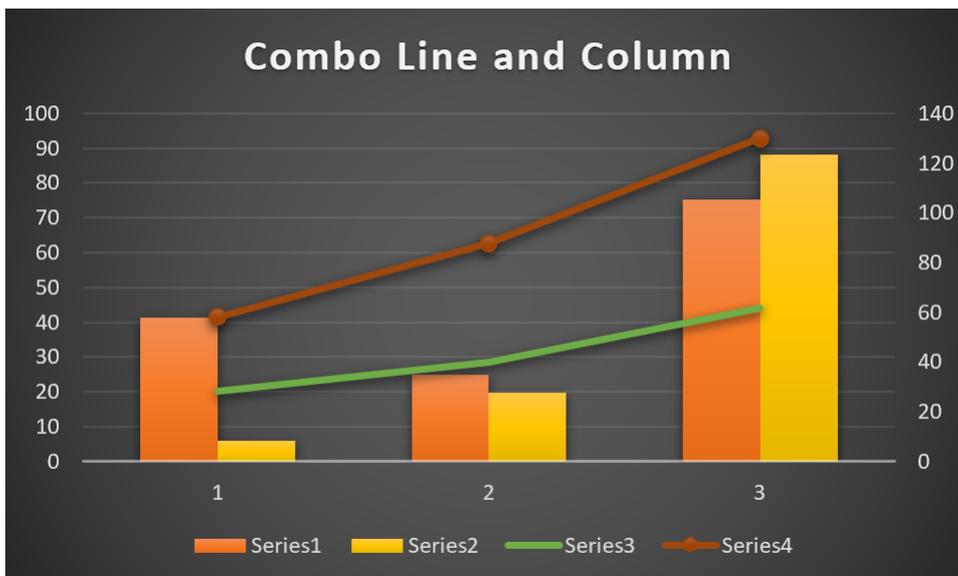
Funnel

Not Supported

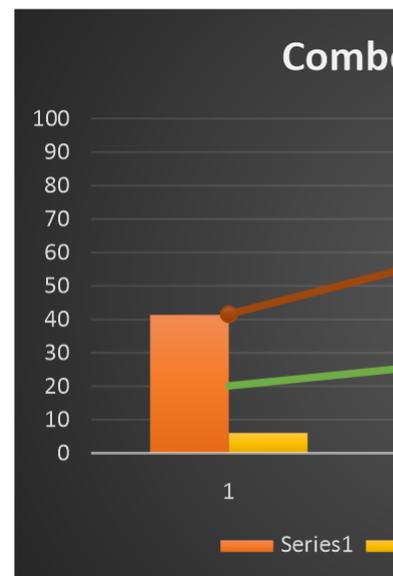
Combo

All combo combinations are supported.

Excel

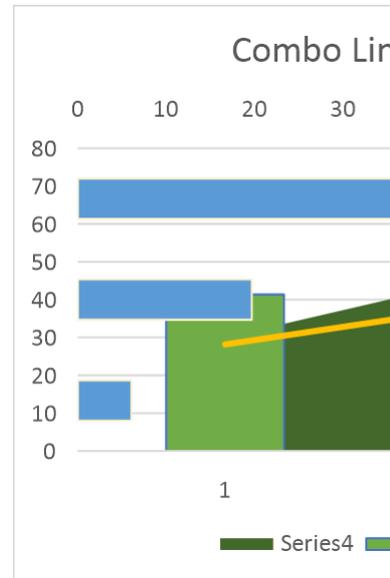
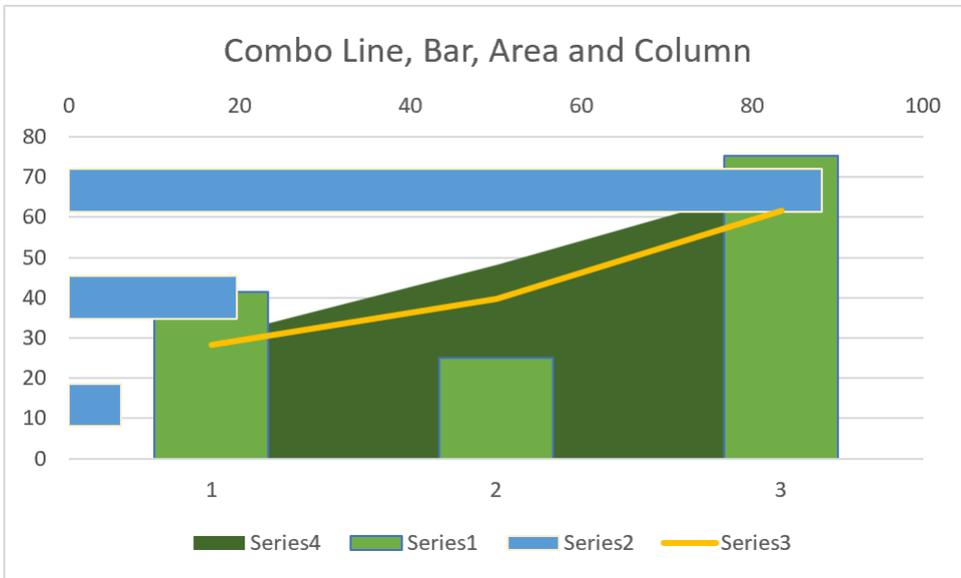


FlexCel

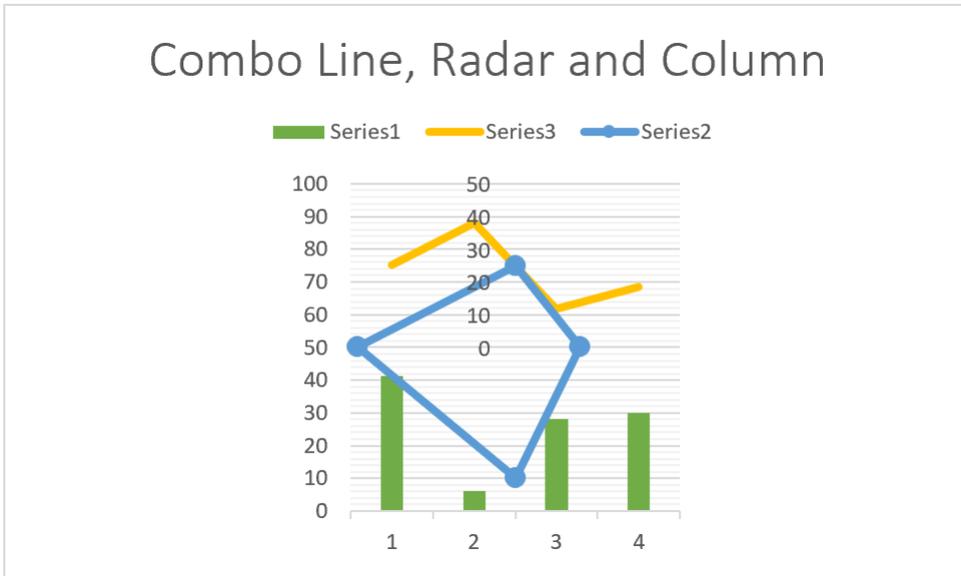


Excel

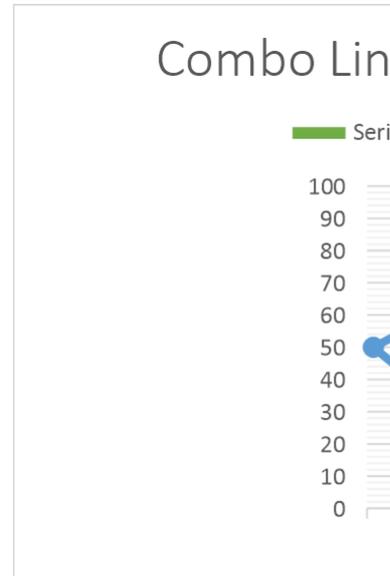
FlexCel



Excel



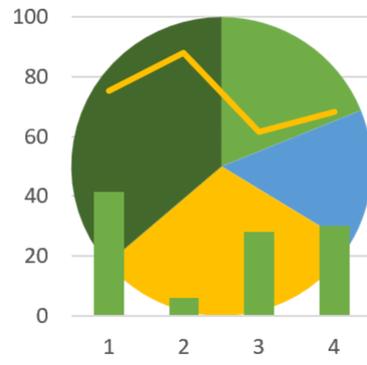
FlexCel



Excel

FlexCel

Combo Column, Line and Pie



Series2 Series1 Series3

Combo



Series

Copyright information

Main Copyright

Unless in the parts specifically mentioned below, all files in this distribution are copyright (c) Adrian Gallero and licensed under the terms detailed in the file license.rtf.

Third party copyrights

This distribution might also contain the following licensed code:

File MD5: MD5 Message-Digest Algorithm.

Parts are Copyright (c) 1990-2, RSA Data Security, Inc.

Those parts are licensed under the following terms:

```
Copyright (C) 1990-2, RSA Data Security, Inc. Created 1990.  
All rights reserved.
```

```
RSA Data Security, Inc. makes no representations concerning either  
the merchantability of this software or the suitability of this  
software for any particular purpose. It is provided "as is"  
without express or implied warranty of any kind.
```

```
These notices must be retained in any copies of any part of this  
documentation and/or software.
```

```
Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991.  
All rights reserved.
```

Parts are Copyright (c) 2002 GL Conseil. All rights reserved.

Those parts are licensed under the following terms:

```
Copyright 2002 GL Conseil. All rights reserved.  
Permission is granted to anyone to use this software for any purpose on  
any computer system, and to alter it and redistribute it, subject  
to the following restrictions:
```

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.

SHA1 Algorithm.

FlexCel uses the SHA1 algorithm which is Copyright (C) The Internet Society (2001). All Rights Reserved.

SHA1 is licensed under the following terms:

RFC 3174 - US Secure Hash Algorithm 1 (SHA1)

Copyright(C) The Internet Society(2001). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

Files UCompress20: Zlib algorithm implementation.

Parts are Copyright (C) 1995-2004 Jean-loup Gailly and Mark Adler

Those parts are licensed under the following terms:

Copyright (C) 1995-2004 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

File EllipticalArc: Implementation of an arc using Bezier curves.

Parts are Copyright (c) 2003-2004, Luc Maisonobe

Those parts are licensed under the following terms:

```
// Copyright (c) 2003-2004, Luc Maisonobe
// All rights reserved.
//
// Redistribution and use in source and binary forms, with
// or without modification, are permitted provided that
// the following conditions are met:
//
//   Redistributions of source code must retain the
//   above copyright notice, this list of conditions and
//   the following disclaimer.
//   Redistributions in binary form must reproduce the
//   above copyright notice, this list of conditions and
//   the following disclaimer in the documentation
//   and/or other materials provided with the
//   distribution.
//   Neither the names of spaceroots.org, spaceroots.com
//   nor the names of their contributors may be used to
//   endorse or promote products derived from this
//   software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
```

```
// CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
// WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
// PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
// THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY
// DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
// PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
// USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
// HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
// IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
// NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
// USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.
```

File BidiReference, BidiPBReference and ArabicShaping: Implementation of arabic shaping algorithm.

Parts are Copyright (c) 1991-2019 Unicode, Inc. All rights reserved. Distributed under the Terms of Use in <http://www.unicode.org/copyright.html>

Those parts are licensed under the following following terms:

UNICODE, INC. LICENSE AGREEMENT - DATA FILES AND SOFTWARE

Unicode Data Files include all data files under the directories <http://www.unicode.org/Public/>, <http://www.unicode.org/reports/>, and <http://www.unicode.org/cldr/data/>. Unicode Data Files do not include PDF online code charts under the directory <http://www.unicode.org/Public/>. Software includes any source code published in the Unicode Standard or under the directories <http://www.unicode.org/Public/>, <http://www.unicode.org/reports/>, and <http://www.unicode.org/cldr/data/>.

NOTICE TO USER: Carefully read the following legal agreement. BY DOWNLOADING, INSTALLING, COPYING OR OTHERWISE USING UNICODE INC.'S DATA FILES ("DATA FILES"), AND/OR SOFTWARE ("SOFTWARE"), YOU UNEQUIVOCALLY ACCEPT, AND AGREE TO BE BOUND BY, ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE, DO NOT DOWNLOAD, INSTALL, COPY, DISTRIBUTE OR USE THE DATA FILES OR SOFTWARE.

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1991-2019 Unicode, Inc. All rights reserved. Distributed under the Terms of Use in <http://www.unicode.org/copyright.html>.

Permission is hereby granted, free of charge, to any person obtaining a copy of the Unicode data files and any associated documentation (the "Data Files") or Unicode software and any associated documentation (the "Software") to deal in the Data Files or Software without restriction, including without limitation the

rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Data Files or Software, and to permit persons to whom the Data Files or Software are furnished to do so, provided that (a) the above copyright notice(s) and this permission notice appear with all copies of the Data Files or Software, (b) both the above copyright notice(s) and this permission notice appear in associated documentation, and (c) there is clear notice in each modified Data File or in the Software as well as in the documentation associated with the Data File(s) or Software that the data or software has been modified.

THE DATA FILES AND SOFTWARE ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE DATA FILES OR SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in these Data Files or Software without prior written authorization of the copyright holder

File KhmerShaper: Shaper for Khmer language.

Parts are Copyright (c) 2008 Nokia Corporation and/or its subsidiary(-ies)

Those parts are licensed under the following following terms:

```
/*
 * Copyright (C) 2008 Nokia Corporation and/or its subsidiary(-ies)
 *
 * This is part of HarfBuzz, an OpenType Layout engine library.
 *
 * Permission is hereby granted, without written agreement and without
 * license or royalty fees, to use, copy, modify, and distribute this
 * software and its documentation for any purpose, provided that the
 * above copyright notice and the following two paragraphs appear in
 * all copies of this software.
 *
 * IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
 * ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN
 * IF THE COPYRIGHT HOLDER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 *
 * THE COPYRIGHT HOLDER SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING,
 * BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
 * FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE COPYRIGHT HOLDER HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
 */
```

Files OctreeQuantizer, OctreeQuantizer, Quantizer: Color Quantizer to convert images to grayscale.

Parts are Copyright (c) Morgan Skinner

Those parts are licensed under the following following terms:

```
THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
PARTICULAR PURPOSE.
```

```
This is sample code and is freely distributable.
```

File: sRGB_IEC61966-2-1_black_scaled.icc: Color profile for embedding in PDF/A files.

Copyright International Color Consortium, 2009

This file contains a color profile which will be embedded when you create PDF/A files, or when you set the PDF generation options to embed the color profile. It is licensed on the following terms:

To anyone who acknowledges that the file "sRGB_IEC61966-2-1_black_scaled.icc" is provided "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTY, permission to use, copy and distribute these file for any purpose is hereby granted without fee, provided that the file is not changed including the ICC copyright notice tag, and that the name of ICC shall not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. ICC makes no representations about the suitability of this software for any purpose.

File: PdfStandardFontInfo.data.gz: Standard PDF Font data.

Copyright (c) Adobe systems

PdfStandardFontInfo.data.gz contains postscript data for fonts, and data comes from the AFM files published by Adobe, which are distributed under the following license:

This file and the 14 PostScript(R) AFM files it accompanies may be used, copied, and distributed for any purpose and without charge, with or without modification, provided that all copyright notices are retained; that the AFM files are not distributed without this file; that all modifications to this file or any of the AFM files are prominently noted in the modified file(s); and that this paragraph is not modified. Adobe Systems has no responsibility or obligation to support the use of the AFM files.

File: Northwind.sqlite: Demo database.

Copyright (c) 2008, Microsoft

This file is used in the FlexCel demos (not used in the FlexCel itself) and it is under the following license:

Copyright (c) 2008, Microsoft
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of Microsoft nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior

written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Files: All files in 20.Reports\SharedData\SQLite folder: Embedded DB Server for the Demos.

Public Domain

The SQLite files are used only in the demos (not in used in FlexCel itself). SQLite is in the Public Domain:

All of the code and documentation in SQLite has been dedicated to the public domain by the authors. All code authors, and representatives of the companies they work for, have signed affidavits dedicating their contributions to the public domain and originals of those signed affidavits are stored in a firesafe at the main offices of Hwaci. Anyone is free to copy, modify, publish, use, compile, sell, or distribute the original SQLite code, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

Icons in demos and tools.

Some of the icons used in demos and tools are created using objects which are Copyright (c) Axialis Software Corporation

Those objects are used under the following license:

This is a legal agreement between you (the "User") and Axialis Software Corporation ("Axialis"). By downloading this object pack (the "Objects") from Axialis.com, the user agrees to the following:

License Grant

Axialis grants the User a non-exclusive, non-transferable, royalty-free license to use these objects as indicated herein. The Objects can be assembled together to create icons (the "Icons").

You MAY:(a) assemble the Objects to create Icons using an Axialis product only; (b) use the Objects "as is" in personal or commercial software projects (menus, toolbars, dialog boxes...) as long as you own a licensed version of the associated Axialis Product; (c) use the created Icons in personal or commercial software projects (menus, toolbars, dialog boxes...) as long as you own a licensed version of the associated Axialis Product.

Restrictions

You MAY NOT:(a) use the Objects without owning a licensed version of the associated Axialis Product; (b) redistribute, loan, rent, sell the Objects "as is" and/or the created Icons as a set of Icons or individually; (c) use the Icons or Objects to illustrate pornographic, immoral, illegal or defamatory material.

Copyright / Ownership

The Objects are proprietary products of AXIALIS and are protected by copyright and other intellectual property laws. The Objects are licensed and not sold. You acquire only the right to use the Objects and do not acquire any rights, express or implied, in the Software other than those specified in this License.

Disclaimer of warranties

The Objects is supplied "as is". AXIALIS disclaims all warranties, expressed or implied, including, without limitation, the warranties of merchantability and of fitness for any purpose. The user must assume the entire risk of using the Objects.

Disclaimer of damages

AXIALIS assumes no liability for damages, direct or consequential, which may result from the use of the Objects, even if AXIALIS has been advised of the possibility of such damages. Any liability of the seller will be limited to refund the purchase price if any.

TMS component single developer license agreement

TMS component license agreement between TMS software and holder of the component license.

The license of the component gives you the right to:

- use the component for development of applications or any type of software module in general by a single developer within the company holding the license.
- sell any commercial compiled application with the control, published by the company holding the license.
- make modifications to the source code of component for own use.
- use the component and source code on all development systems used by the developer assigned by the company holding the license.
- request future versions of the component during the license period of one year after purchase date. After expiry of the registration TMS software can no longer provide any old version of software, documentation or samples. TMS software is not a backup service and expects backups to be made by the licensed user.
- access to priority email, web forum support by the single developer assigned by the company holding the license during the license period.
- sell any number of applications in any quantity without any additional run-time fees required.
- use the software as-is perpetually.

The license agreement prevents you from:

- distributing parts or full source code of any component from TMS software.
- using parts or full source code of components from the TMS software for creating any type of other components that are distributed or sold with or without source code.
- changing the source code of any component from TMS software and sell or distribute this as a modified product.
- creating a descendant compiled product such as OCX, ActiveX, .NET, Web, VCL control and sell or distribute this as a product.
- using the control in applications sold with different publisher name than the company holding the license.
- transferring the license to another developer.
- transferring the license to another company.
- using the components by multiple developers in the company holding the license.
- installing the components on machines not primarily used by the licensed developer.

The license agreement terminates immediately after violation of any of the terms and conditions described

Disclaimer:

THIS SOFTWARE IS PROVIDED TO YOU "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED INCLUDING BUT NOT LIMITED TO THE APPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE. YOU ASSUME THE ENTIRE RISK AS TO THE ACCURACY AND THE USE OF THE SOFTWARE AND ALL OTHER RISK ARISING

OUT OF THE USE OR PERFORMANCE OF THIS SOFTWARE AND DOCUMENTATION.
tmssoftware.com bvba SHALL NOT BE LIABLE FOR ANY DAMAGES WHATSOEVER ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF tmssoftware.com bvba HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL tmssoftware.com bvba BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, DIRECT, INDIRECT, SPECIAL, PUNITIVE, OR OTHER DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO DAMAGES OR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS, EVEN IF tmssoftware.com bvba HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TMS component site license agreement

TMS component license agreement between TMS software and holder of the component license.

The license of the component gives you the right to:

- use the component for development of applications or any type of software module in general by all developers within the company holding the license.
- sell any commercial compiled application with the control, published by the company holding the license.
- make modifications to the source code of component for own use.
- use the component and source code on all development systems used by all developers in the company holding the license.
- request future versions of the component at any time either through the web or by email for one year after purchase. After expiry of the registration TMS software can no longer provide any old version of software, documentation or samples. TMS software is not a backup service and expects backups to be made by the licensed user.
- access to priority email, web forum support by all developers in the company holding the license during the license period.
- sell any number of applications in any quantity without any additional run-time fees required.
- change at any time the number of developers using the TMS software components within the company holding the license.
- notify TMS software at any time to allow new developers within the company to access the priority email, web forum support during the license period of one year.
- allow any number of developers within the company holding the license to access the web based interface for obtaining product updates.
- use the software as-is perpetually.

The license agreement prevents you from:

- distributing parts or full source code of any component from TMS software.
- using parts or full source code of components from the TMS software for creating any type of other components that are distributed or sold with or without source code.
- changing the source code of any component from TMS software and sell or distribute this as a modified product.
- creating a descendant compiled product such as OCX, ActiveX, .NET, Web, VCL control and sell or distribute this as a product.
- using the control in applications sold with different publisher name than the company holding the license.

The license agreement terminates immediately after violation of any of the terms and conditions described

Disclaimer:

THIS SOFTWARE IS PROVIDED TO YOU "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED INCLUDING BUT NOT LIMITED TO THE APPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE. YOU ASSUME THE ENTIRE RISK AS TO THE ACCURACY AND THE USE OF THE SOFTWARE AND ALL OTHER RISK ARISING OUT OF THE USE OR PERFORMANCE OF THIS SOFTWARE AND DOCUMENTATION.

tmssoftware.com bvba SHALL NOT BE LIABLE FOR ANY DAMAGES WHATSOEVER ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF tmssoftware.com bvba HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL tmssoftware.com bvba BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, DIRECT, INDIRECT, SPECIAL, PUNITIVE, OR OTHER DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO DAMAGES OR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS, EVEN IF tmssoftware.com bvba HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
