NPOI Integration To Read/Write Excel Files Using C#

AzraJabeen Mohamed Ali

Independent researcher, California, USA
Azra.jbn@gmail.com

Abstract:
This paper discusses NPOI integration to read and write the excel files using C#. This paper explores using the NPOI tool to read and write Excel files in C#. The project's main research question is how to read and write Excel files so that mass data can be imported and exported using NPOI which supports Windows and Linux OS. Methodologically speaking, this study is based on careful analysis of the usage of third-party tools like LinqToExcel and NPOI. The course deconstructs the fundamentals of utilizing C# to read and write Excel files with an interface focused NPOI. The study's key findings provide insight into the intricate link between Csharp and NPOI in accessing and storing data in Excel sheets. The paper covers the majority of NPOI Excel's features (cell style, data format, formula, etc.) and explains how these elements contribute to big data processing.

Keywords: Microsoft Visual Studio, Nuget package, NPOI, Excel, interop, Microsoft Office.

1. Introduction

NPOI

NPOI library is capable of reading and writing binary Word and excel documents. It also supports the most recent Excel file format (.xlsx) and the older Excel BIFF format (.xls), which allows the application to read and write older files. It supports the majority of Excel functionality, at least in part. Installation in Microsoft Visual Studio is simple since it's available as a NuGet package. It is an open source, free and a stand-alone implementation and it does not require interop. Excel and word files can be read or written on .Net without the need to install any third part libraries by referencing the Microsoft.Interop assemblies.. It is more efficient than calling Microsoft Excel ActiveX in the background and creates an Excel report without the Microsoft Office suite being installed on the server.
The drawback is that this can be slow because an instance of Word or excel is launched in the background to process the request and it only works if the user actually has the necessary version of Word or excel installed. This didn't work on several projects because either Excel wasn't installed, it was an outdated version, or the user had Excel open, and it somehow prevented them from communicating.
NPOI eliminates these dependencies. It reads and writes binary Excel and Word files using the library. Additionally it supports reading the latest Excel format (.xlsx) and older format (.xls).

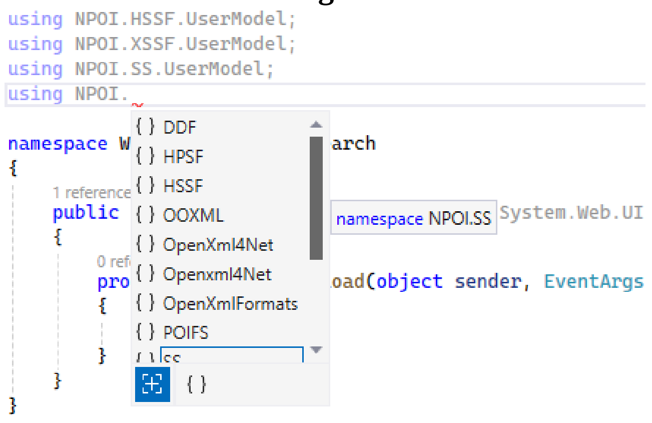
Installation of NPOI:

Right-click on the project in Visual Studio's Solution Explorer, then choose "Manage NuGet Packages." Search for "NPOI" in the NuGet Package Manager by selecting the "Browse" option. Click "Install" to install the "NPOI" package after selecting it.

NPOI Integration:

After the installation is finished, add the following using line to your C# code to utilize NPOI[Fig-1]:

Fig-1



```
using NPOI.HSSF.UserModel;
using NPOI.XSSF.UserModel;
using NPOI.SS.UserModel;
using NPOI.

namespace W                arch
{
    1 reference
    public                           System.Web.UI
    {
        0 refe
        pro                oad(object sender, EventArgs
        {

        }
    }
}
```

NPOI is arranged in C# into distinct namespaces that manage multiple Microsoft Office applications, including Word, PowerPoint, and Excel. Because Excel is the most frequently used application, you will most likely interface with namespaces connected to Excel (both in.xls and.xlsx formats).

NPOI.SS.UserModel (Core Excel Interfaces):

Interfaces and classes shared by all Excel workbook and sheet types are contained in this namespace. This namespace will be used for common Excel tasks including making sheets, rows, and columns. The full Excel worksheet is represented by IWorkbook. A sheet in the workbook is represented by the ISheet. A row in the sheet is denoted by the symbol IRow.

A cell in a row is represented by the symbol ICell.

Cell styles are represented by ICellStyle.

Fonts used in cells are represented by IFont.

IFormulaEvaluator is an interface for assessing cell formulas.

NPOI.HSSF.UserModel (Excel .xls files)

Classes designed to handle HSSF (Horrible Spreadsheet Format), the outdated Excel file format used for.xls files (Excel 97-2003), are contained in this namespace. A workbook in the.xls format is represented by the HSSFWorkbook object.

A sheet in a workbook in.xls format is represented by the HSSFSheet. A row in a.xls sheet is represented by HSSFRow.

A cell in a.xls sheet is represented by the HSSFCell.

NPOI.XSSF.UserModel (Excel .xlsx files)

Classes designed to handle XSSF (XML Spreadsheet Format), the more recent Excel file format used for.xlsx files (Excel 2007 and later), are contained in this namespace. A workbook in the.xlsx format is represented by the file type XSSFWorkbook. A sheet in an XLSX spreadsheet is represented by the symbol XSSFSheet. A row in a.xlsx sheet is represented by the symbol XSSFRow. A cell in a.xlsx sheet is represented by the symbol XSSFCell.

NPOI.SS.Util (Utility Classes):

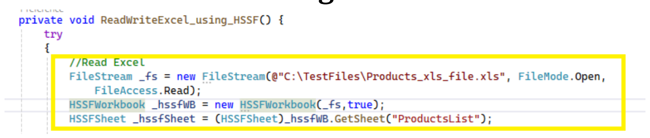
For activities like cell references, cell range addresses, and formulas, this namespace offers utility classes.

Excel cell references (such as "A1" and "B2") can be worked with using the CellReference utility class. AreaReference: Indicates a range of cells (A1:B10, for example).
NPOI.POIFS.FileSystem (For Working with Legacy .xls Formats):
The POIFS (Poor Obfuscation Implementation File System) namespace is used to handle file structures and is utilized by legacy Excel formats, specifically.xls. For managing POIFS file systems (ancient Excel file formats), use POIFSFileSystem.

Accessing Excel to read using NPOI:
XSSF and HSSF are the two Excel versions that NPOI supports in which XSSF supports .xlsx excel format and HSSF works with .xls files. They are set up such that you will always interact with NPOI in the same manner as long as it is aware of the file type it is currently handling.

Read and Write .xls file using NPOI HSSF UserModel:
We will use NPOI HSSFWorkbook to read below shown .xls file. Here we are going to read “ProductsList” worksheet and store it in new worksheet called “OutputList”. The ProductsList worksheet consists of columns “ProductName”,”ProductSerial”,”Cost” with 999 rows records.
Here is the code to read the .xls file using HSSF.Usermodel.
The code you provided is using **NPOI** to read an older Excel .xls file, specifically using HSSFWorkbook for handling .xls files (Excel 97-2003 format) [Fig-2]. Excel file is opened using FileStream in read mode. The HSSFWorkbook class is used to read .xls files (Excel 97-2003 format). Enabled the option true for the HSSFWorkbook constructor, which is intended to support reading the file with the correct character encoding. The HSSFSheet class is the correct type to work with sheets from an .xls file.

Fig-2

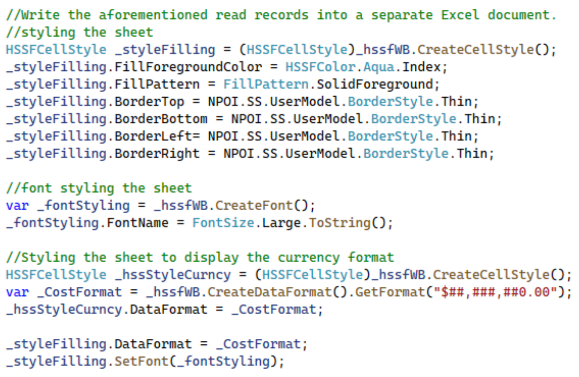


```
private void ReadWriteExcel_using_HSSF() {
    try
    {
        //Read Excel
        FileStream _fs = new FileStream(@"C:\TestFiles\Products_xls_file.xls", FileMode.Open,
            FileAccess.Read);
        HSSFWorkbook _hssfWB = new HSSFWorkbook(_fs,true);
        HSSFSheet _hssfSheet = (HSSFSheet)_hssfWB.GetSheet("ProductsList");
```

Initially, when writing, style the worksheet [Fig-3]. Code snippet is working on applying cell styles (such as background color, borders, and font styling) and a currency format to cells in an **Excel** sheet using

NPOI.
Font Styling: The line FontSize.Large.ToString() is used for setting a font size.
Currency Format: "$#,##0.00" is used to apply the currency format, which formats numbers as currency with two decimal places and a comma for thousands. It is possible to change the format to suit as per needs, for example, by adding a currency symbol or changing the amount of decimals. The currency format style (_hssStyleCurncy) is applied to the specific cell that holds a numeric value.
Cell Styles: For the first styled cell (anotherCell), the _styleFilling object is utilized, to which the background color and border styles are applied. A currency value in the cell (cell) is formatted using the _hssStyleCurncy style.

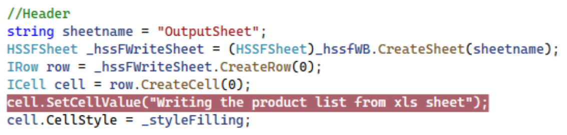
Fig-3



```
//Write the aforementioned read records into a separate Excel document.
//styling the sheet
HSSFCellStyle _styleFilling = (HSSFCellStyle)_hssfWB.CreateCellStyle();
_styleFilling.FillForegroundColor = HSSFColor.Aqua.Index;
_styleFilling.FillPattern = FillPattern.SolidForeground;
_styleFilling.BorderTop = NPOI.SS.UserModel.BorderStyle.Thin;
_styleFilling.BorderBottom = NPOI.SS.UserModel.BorderStyle.Thin;
_styleFilling.BorderLeft= NPOI.SS.UserModel.BorderStyle.Thin;
_styleFilling.BorderRight = NPOI.SS.UserModel.BorderStyle.Thin;

//font styling the sheet
var _fontStyling = _hssfWB.CreateFont();
_fontStyling.FontName = FontSize.Large.ToString();

//Styling the sheet to display the currency format
HSSFCellStyle _hssStyleCurncy = (HSSFCellStyle)_hssfWB.CreateCellStyle();
var _CostFormat = _hssfWB.CreateDataFormat().GetFormat("$##,###,##0.00");
_hssStyleCurncy.DataFormat = _CostFormat;

_styleFilling.DataFormat = _CostFormat;
_styleFilling.SetFont(_fontStyling);
```

Creation and styling header in HSSFWorkbook [Fig-4]. Code is working to create a new sheet named "OutputSheet" in an Excel workbook and write a header row in the first cell with the text "Writing the product list from xls sheet". Applying a custom style (_styleFilling) to this header cell.

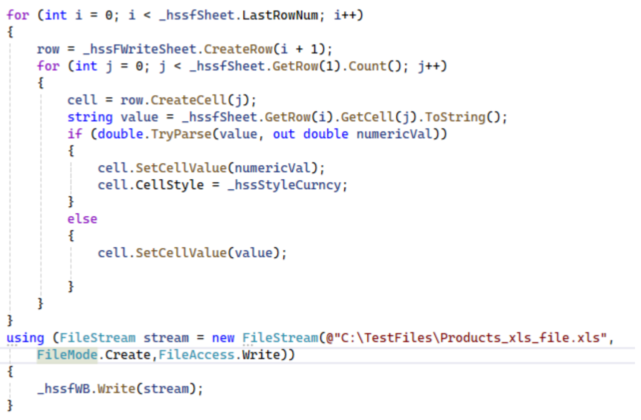
Fig-4



```
//Header
string sheetname = "OutputSheet";
HSSFSheet _hssFWriteSheet = (HSSFSheet)_hssfWB.CreateSheet(sheetname);
IRow row = _hssFWriteSheet.CreateRow(0);
ICell cell = row.CreateCell(0);
cell.SetCellValue("Writing the product list from xls sheet");
cell.CellStyle = _styleFilling;
```

This code is used to write the entries that were extracted from the "ProductsList" worksheet into the "OutputList" worksheet[Fig-5]. Code is iterating through an existing sheet (_hssfSheet), copying its data into a new sheet (_hssFWriteSheet), and applying a currency style (_hssStyleCurncy) to cells that contain numeric values. In the inner loop GetRow() is used to determine the count of cells. The last row is returned by LastRowNum, which is **zero-based**. This means LastRowNum gives the last valid row index, so you should adjust the iteration to make sure it loops through all rows, including the last one. The FileStream is used to save the workbook to a specified path.

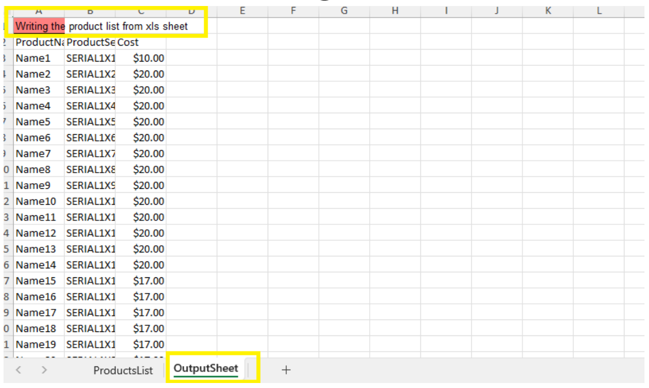
Fig-5



```
for (int i = 0; i < _hssfSheet.LastRowNum; i++)
{
    row = _hssFWriteSheet.CreateRow(i + 1);
    for (int j = 0; j < _hssfSheet.GetRow(1).Count(); j++)
    {
        cell = row.CreateCell(j);
        string value = _hssfSheet.GetRow(i).GetCell(j).ToString();
        if (double.TryParse(value, out double numericVal))
        {
            cell.SetCellValue(numericVal);
            cell.CellStyle = _hssStyleCurncy;
        }
        else
        {
            cell.SetCellValue(value);

        }
    }
}
using (FileStream stream = new FileStream(@"C:\TestFiles\Products_xls_file.xls",
    FileMode.Create,FileAccess.Write))
{
    _hssfWB.Write(stream);
}
```

Output file will be like [Fig-6].

Fig-6



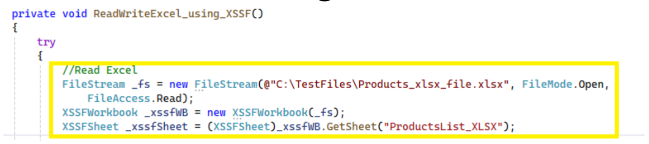
A	B	C
Writing the product list from xls sheet		
ProductNa	ProductSe	Cost
Name1	SERIAL1X1	$10.00
Name2	SERIAL1X2	$20.00
Name3	SERIAL1X3	$20.00
Name4	SERIAL1X4	$20.00
Name5	SERIAL1X5	$20.00
Name6	SERIAL1X6	$20.00
Name7	SERIAL1X7	$20.00
Name8	SERIAL1X8	$20.00
Name9	SERIAL1X9	$20.00
Name10	SERIAL1X1	$20.00
Name11	SERIAL1X1	$20.00
Name12	SERIAL1X1	$20.00
Name13	SERIAL1X1	$20.00
Name14	SERIAL1X1	$20.00
Name15	SERIAL1X1	$17.00
Name16	SERIAL1X1	$17.00
Name17	SERIAL1X1	$17.00
Name18	SERIAL1X1	$17.00
Name19	SERIAL1X1	$17.00

Read and Write .xlsx file using NPOI XSSF UserModel:

We will use NPOI XSSFWorkbook to read below shown .xls file. Here we are going to read "ProductsList" worksheet and store it in new worksheet called "OutputList". The ProductsList worksheet consists of columns "ProductName","ProductSerial","Cost" with 999 rows records.

Here is the code to read the .xlsx file using XSSF.Usermodel [Fig-7]. Code snippet is using the NPOI library to read an Excel file in the .xlsx format (Excel 2007 or later). Opening a file from a specified path and reading the sheet "ProductsList_XLSX". FileStream is used to open an Excel file in read mode (FileAccess.Read). This allows NPOI to read the contents of the file. The Excel workbook is loaded using the XSSFWorkbook class from NPOI. Unlike HSSFWorkbook, which is used for.xls files (Excel 97-2003), this class is especially used for.xlsx files (Excel 2007 and later). Sheets in NPOI are accessed by their name, or by index. Here getting the sheet "ProductsList_XLSX" from the workbook

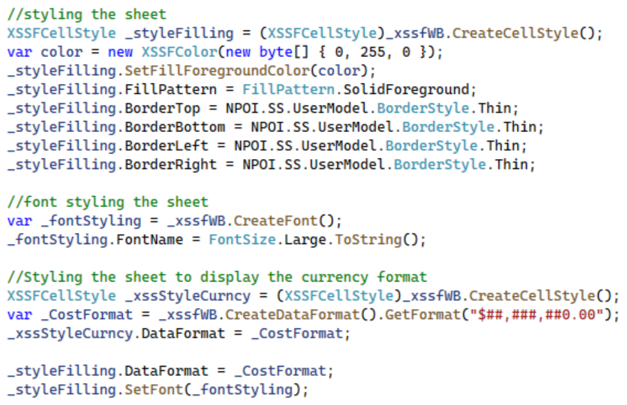
Fig-7



```
private void ReadWriteExcel_using_XSSF()
{
    try
    {
        //Read Excel
        FileStream _fs = new FileStream(@"C:\TestFiles\Products_xlsx_file.xlsx", FileMode.Open,
            FileAccess.Read);
        XSSFWorkbook _xssfWB = new XSSFWorkbook(_fs);
        XSSFSheet _xssfSheet = (XSSFSheet)_xssfWB.GetSheet("ProductsList_XLSX");
```

Initially, when writing, style the worksheet [Fig-8]. Font Styleing, currency Format and cell styling are done using XSSFCellStyle and XSSFont.

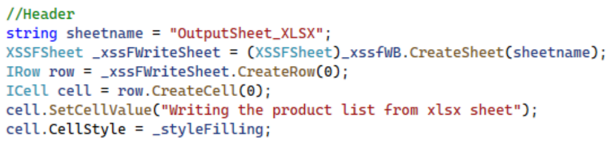
Fig-8



```
//styling the sheet
XSSFCellStyle _styleFilling = (XSSFCellStyle)_xssfWB.CreateCellStyle();
var color = new XSSFColor(new byte[] { 0, 255, 0 });
_styleFilling.SetFillForegroundColor(color);
_styleFilling.FillPattern = FillPattern.SolidForeground;
_styleFilling.BorderTop = NPOI.SS.UserModel.BorderStyle.Thin;
_styleFilling.BorderBottom = NPOI.SS.UserModel.BorderStyle.Thin;
_styleFilling.BorderLeft = NPOI.SS.UserModel.BorderStyle.Thin;
_styleFilling.BorderRight = NPOI.SS.UserModel.BorderStyle.Thin;

//font styling the sheet
var _fontStyling = _xssfWB.CreateFont();
_fontStyling.FontName = FontSize.Large.ToString();

//Styling the sheet to display the currency format
XSSFCellStyle _xssStyleCurncy = (XSSFCellStyle)_xssfWB.CreateCellStyle();
var _CostFormat = _xssfWB.CreateDataFormat().GetFormat("$##,###,##0.00");
_xssStyleCurncy.DataFormat = _CostFormat;

_styleFilling.DataFormat = _CostFormat;
_styleFilling.SetFont(_fontStyling);
```

Creation and styling header in XSSFWorkbook [Fig-9]. The header text ("Writing the product list from xlsx sheet") is applied to the first cell of the first row. applying the _styleFilling to the header cell to give it the fill color and border.

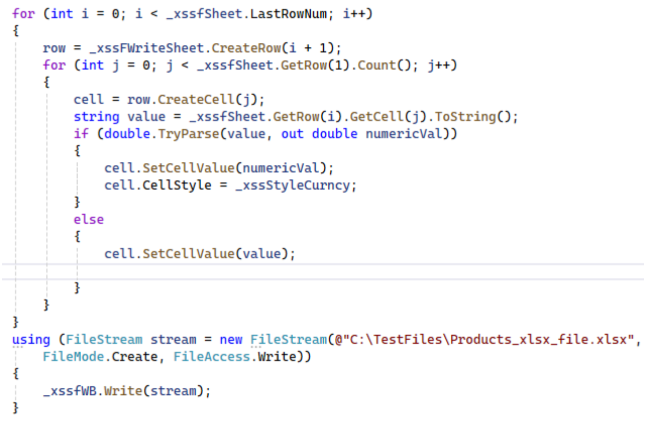
Fig-9



```
//Header
string sheetname = "OutputSheet_XLSX";
XSSFSheet _xssFWriteSheet = (XSSFSheet)_xssfWB.CreateSheet(sheetname);
IRow row = _xssFWriteSheet.CreateRow(0);
ICell cell = row.CreateCell(0);
cell.SetCellValue("Writing the product list from xlsx sheet");
cell.CellStyle = _styleFilling;
```

This code is used to write the entries that were extracted from the "ProductsList_XLSX" worksheet into the "OutputList_XLSX" worksheet [Fig-10]. Code is iterating to transfer data from one sheet to another (_xssfSheet to _xssFWriteSheet). Code is iterating through an existing sheet (_xssfSheet), copying its data into a new sheet (_xssFWriteSheet), and applying a currency style (_xssStyleCurncy) to cells that contain numeric values. In the inner loop GetRow() is used to determine the count of cells. The last row is returned by LastRowNum, which is **zero-based**. This means LastRowNum gives the last valid row index, so you should adjust the iteration to make sure it loops through all rows, including the last one. The FileStream is used to save the workbook to a specified path.

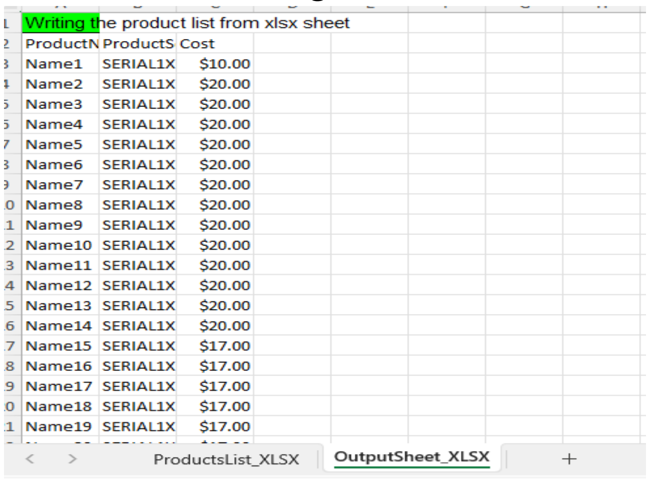
Fig-10



```
for (int i = 0; i < _xssfSheet.LastRowNum; i++)
{
    row = _xssFWriteSheet.CreateRow(i + 1);
    for (int j = 0; j < _xssfSheet.GetRow(1).Count(); j++)
    {
        cell = row.CreateCell(j);
        string value = _xssfSheet.GetRow(i).GetCell(j).ToString();
        if (double.TryParse(value, out double numericVal))
        {
            cell.SetCellValue(numericVal);
            cell.CellStyle = _xssStyleCurncy;
        }
        else
        {
            cell.SetCellValue(value);

        }
    }
}
using (FileStream stream = new FileStream(@"C:\TestFiles\Products_xlsx_file.xlsx",
    FileMode.Create, FileAccess.Write))
{
    _xssfWB.Write(stream);
}
```

Output file will be like [Fig-11].

Fig-11



1	Writing the product list from xlsx sheet		
2	ProductN	ProductS	Cost
3	Name1	SERIAL1X	$10.00
4	Name2	SERIAL1X	$20.00
5	Name3	SERIAL1X	$20.00
6	Name4	SERIAL1X	$20.00
7	Name5	SERIAL1X	$20.00
8	Name6	SERIAL1X	$20.00
9	Name7	SERIAL1X	$20.00
10	Name8	SERIAL1X	$20.00
11	Name9	SERIAL1X	$20.00
12	Name10	SERIAL1X	$20.00
13	Name11	SERIAL1X	$20.00
14	Name12	SERIAL1X	$20.00
15	Name13	SERIAL1X	$20.00
16	Name14	SERIAL1X	$20.00
17	Name15	SERIAL1X	$17.00
18	Name16	SERIAL1X	$17.00
19	Name17	SERIAL1X	$17.00
20	Name18	SERIAL1X	$17.00
21	Name19	SERIAL1X	$17.00

Comparison of HSSF and XSSF Usermodels:

HSSF	XSSF
Handles xls files	Handles xlsx files
All versions of Microsoft Excel can read it.	Excel 2007 and subsequent versions can read this
It can handle larger files and supports up to 65,536 rows and 256 columns.	It can handle smaller files due to XML based open file format and data compression and supports up to 1,048,576 rows and 16,384 columns.

Conclusion

There are many different libraries available in the market to read and write excel files, but NPOI stands out since its free, open source, stand-alone without interoperability. Bold, italics, underlining, font family, font color, and background color stylings are among the styles it offers. Columns can be auto sized. It supports Currency and Number formatting of the excel data. It is very helpful to migrate data from sql table, dataTable, List, Collections to Excel files.

References

1. Independent Software, "READING/WRITING EXCEL FILES WITH C#" https://www.independent-software.com/introduction-to-npoi.html (Nov 12, 2019)
2. M.T, "Generate Excel With (NPOI) in C#" https://dev.to/mtmb/generate-excel-with-npoi-in-c-904 (Jan 29, 2020)
3. Christian Leutloff , "Different Ways to Access Excel 2003 Workbooks using C#" https://www.codeproject.com/Articles/322469/Different-Ways-to-Access-Excel-2003-Workbooks-usin (Feb 21, 2012)
4. Kunal Chowdhury "Here's how to read Excel 2007 document (XLSX) using NPOI libraries" https://www.kunal-chowdhury.com/2017/07/npoi-excel-2007.html#google_vignette (Jul 13, 2017)