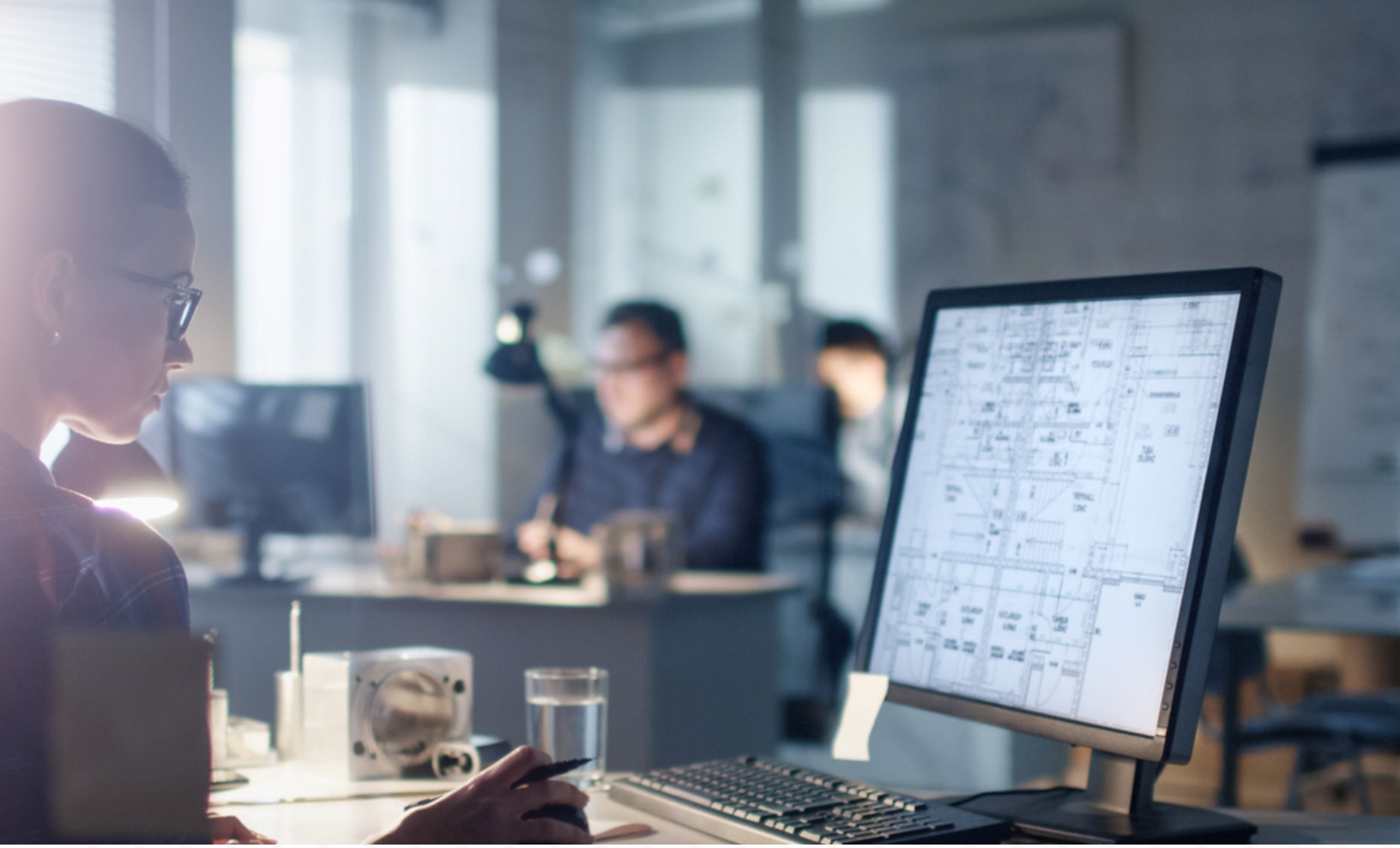
accusoft®

Microsoft Office Automation API Tips and Tricks

Disclaimer: This blog post assumes that you have some familiarity with C# and .NET.

Aram Nshanyan
Software Engineer, PrizmDoc Viewer



Microsoft's (MS) Office Automation .NET API provides powerful and easy-to-use capabilities to integrate MS Office documents in business applications. Below is a set of links that could become a useful starting point to get acquainted with the API:

- How to: Access Office Interop Objects by Using Visual C# Features (C# Programming Guide)
- Microsoft.Office.Interop.Word Namespace
- Microsoft.Office.Interop.Excel Namespace
- Microsoft.Office.Interop.PowerPoint Namespace

In this blog post, I will demonstrate how someone could overcome some well-known (or maybe not yet well-known) problems with Office Automation.

Before we begin, let me introduce this article from Microsoft: **Considerations for server-side Automation of Office**. There are many documented problems when using MS Office Automation for server-side applications, and all of them are confirmed. Although this statement might discourage you from implementing Office Automation:

> Microsoft does not currently recommend, and does not support, Automation of Microsoft Office applications from any unattended, non-interactive client application or component (including ASP, ASP.NET, DCOM, and NT Services), because Office may exhibit unstable behavior and/or deadlock when Office is run in this environment.

Regardless, the above article should not be a blocker for developing business applications based on MS Office. There are some important aspects that should be considered when designing your application, and I'd like to share them here. The facts prove that it is possible to use server-side Automation of Office. Such a considerable fact is our flagship product - PrizmDoc Viewer with its MSO based rendering engine. Please refer to the following page of our online documentation for more information: Natively Render Microsoft Office Documents.

General Tips

Tip 1: Use a real user account with administrative rights for installing the MS Office and running your application.

MS Office Automation does not work under a LocalSystem user or any other Microsoft Windows integrated service account. Therefore, if you are going to run your application as a Windows service, it should run under a real user account that should be a member of the local system's administrators.

Tip 2: Use an activated copy of MS Office.

The installed copy of Microsoft Office must be activated in order for Microsoft Office Automation to work properly: not licensed, not activated, expired, or trial versions of Microsoft Office will not work.

Tip 3: Adjust non-interactive desktop's heap size for Windows service-based application.

If you are going to run your application as a Windows service, then please consider the following page of PrizmDoc Viewer's documentation: Registry Changes. The more MSO instances should run simultaneously, the more you will need to increase the heap size.

Tip 4: Have a default printer set on the system.

When MS Excel is used for printing or exporting to PDF, it invokes the system's default printer driver to render the pages. If the default printer is not set, then the Excel Automation API will fail with COMException. Also, if exporting to PDF is being used, then I would recommend using a **Microsoft XPS Document Writer** printer, because it includes a good set of paper sizes.

Tip 5: Be careful with the automation API's properties; they are not regular C# getters.

Practically all getters of the Automation API are COM objects and are being initialized through Marshalling. So, when getting some property, do not forget to release its value after use. Consider following simple code:

```
using Excel = Microsoft.Office.Interop.Excel;

Excel.Shapes shapes = sheet.Shapes;
try
{
    foreach (Excel.Shape shape in shapes)
    {
        try
        {
            // Use shape
        }
        finally
        {
            Marshal.FinalReleaseComObject(shape);
        }
    }
}
finally
{
    Marshal.FinalReleaseComObject(shapes);
}
```

Any property or object, retrieved via a COM API that is not a C# embedded type, is a COM object that must be released to avoid resource leaks.

Tip 6: Use robust and adaptive error handling.

Consistently use try/catch/finally blocks when dealing with the Automation API. The COMException may raise quite frequently. Be ready for it. Depending on the context, an exception does not necessarily indicate an error case. For instance, upon opening the document, you might get the error:

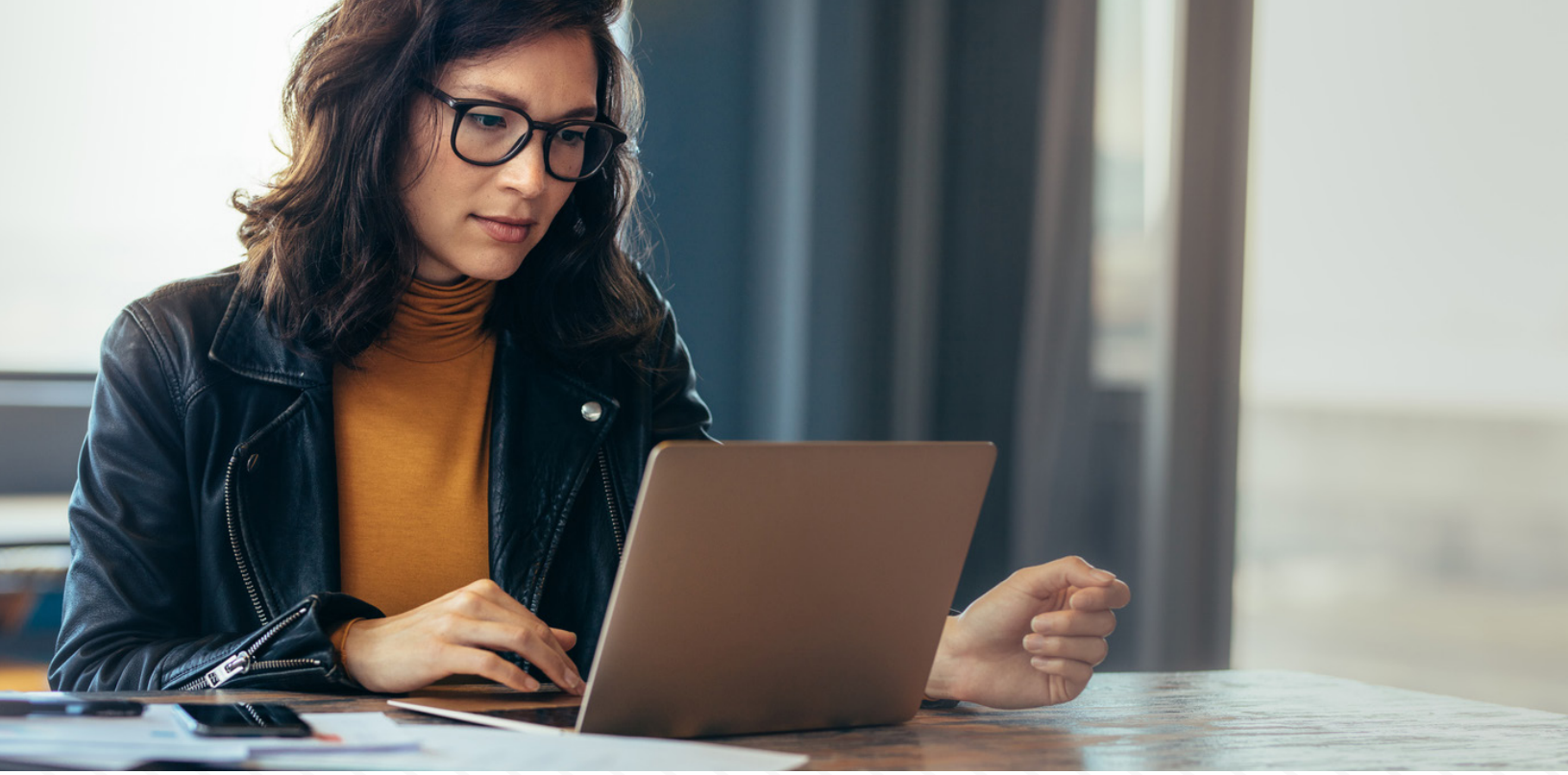
Exception from HRESULT: 0x8001010A (RPC_E_SERVERCALL_RETRYLATER)

One of the possible reasons - the application is busy with some task(s) and cannot execute your command at this time. In such a case, you might consider using retry logic. If after some reasonable number of retries the error is still occurring, then the failing MSO instance should be restarted. Terminate the corresponding MSO application and initialize it again. Below is the example of how to initialize and terminate the Excel application:

```
using Excel = Microsoft.Office.Interop.Excel;

// Initialization
Excel.Application app = new Excel.Application();
app.AutomationSecurity = MsoAutomationSecurity.msoAutomationSecurityForceDisable; //
disable macros
app.DisplayAlerts = false;
app.Visible = false;
app.DisplayDocumentInformationPanel = false;
app.DisplayFullScreen = false;
app.DisplayInfoWindow = false;
app.DisplayRecentFiles = false;
app.EnableAnimations = false;
app.EnableLargeOperationAlert = false;
app.Interactive = false;
app.ShowStartupDialog = false;

// Termination
try
{
   app.Quit();
}
finally
{
   Marshal.FinalReleaseComObject(app);
   app = null;
}
```



Application Specific Tips

Word Tip: Lock interactive fields to avoid application hang.

As already mentioned, MS Office is not designed to be used in non-interactive mode. Even when invoked via the Automation API, the MSO apps may display popup dialogs for user input. This will lead to application deadlock because nobody is supposed to press “OK”/”Cancel” on such dialogs. For Word, the dialogs are usually shown for interactive fields, such as “Fill-in” or “Ask” fields. Corresponding API types are **wdFieldFillIn** and **wdFieldAsk** of [WdFieldType enumeration.](#) To avoid popups, it is necessary to lock such fields. Here is a code example:

```
using Word = Microsoft.Office.Interop.Word;

Word.StoryRanges ranges = this.document.StoryRanges;
try
{
    foreach (Word.Range range in ranges)
    {
        Word.Fields fields = range.Fields;
        try
        {
            foreach (Word.Field field in fields)
            {
                try
                {
```

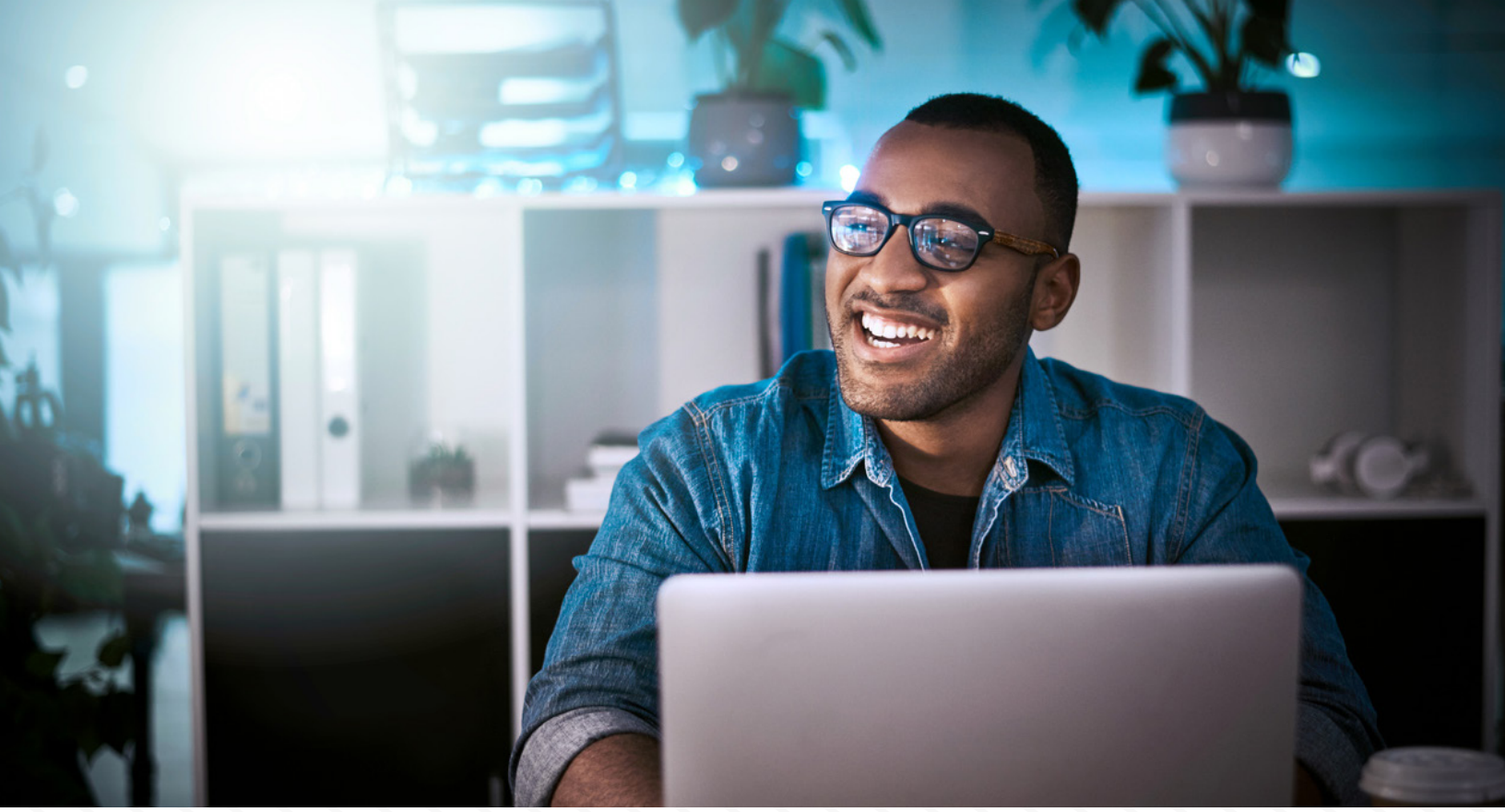
```
                if (this.NeedToLock(field.Type))
                {
                    field.Locked = true;
                }
            }
            finally
            {
                Marshal.FinalReleaseComObject(field);
            }
        }
    }
    finally
    {
        Marshal.FinalReleaseComObject(fields);
        Marshal.FinalReleaseComObject(range);
    }
  }
}
finally
{
   Marshal.FinalReleaseComObject(ranges);
}
```

Excel Tip: Avoid seeking over all cells in the worksheet.

The COM Automation API itself is slow and does not provide good performance. This is more observable in the Excel format when using individual cells of the sheet. To avoid this, remember that each cell is a COM object initialized via Marshalling. Unless it is strictly necessary, avoid seeking over all cells in the loop. Use cell range objects instead. For instance, use **_Worksheet.UsedRange Property** when applicable, instead of **_Worksheet.Cells Property**. Below is an example of how to retrieve e.g. formula cells from the worksheet:

```
using Excel = Microsoft.Office.Interop.Excel;

Excel.Range usedRange = sheet.UsedRange;
Excel.Range formulas = usedRange.SpecialCells(Excel.XlCellType.xlCellTypeFormulas);
...
Marshal.FinalReleaseComObject(formulas);
Marshal.FinalReleaseComObject(usedRange);
```



PowerPoint Tip: Remember that only one PowerPoint application runs on the system.

As opposed to Word and Excel, only a single PowerPoint application runs on the system, despite how many times you are calling: new Microsoft.Office.Interop.PowerPoint.Application(). It is a multithreaded application that handles multiple documents simultaneously. You can load multiple presentations in your application, but take this into account when terminating the PowerPoint instance in your application due to some unrecoverable error. This will affect other open documents. So, for PowerPoint, special logic should be implemented in order to manage multiple documents in parallel.

Afterword

Using Microsoft Automation API, surely, has many obstacles. I believe I mentioned only a subset of them that I have faced in the past. Most probably you will face more. Please don't give up; there are no unresolvable problems. Remember that when you overcome all of them, you will have perfect (from the original manufacturer) rendering fidelity! Although there are some free/paid alternatives to MS Office, personally I did not see any decent competitor from a rendering point of view. I hope my article will be helpful to anyone using the MS Office Automation APIs. Good luck!