

# TENTIVE



Sligro Food Group

## Implementation document

M.G. den Hollander  
Student number: 3803554  
Fontys Hogescholen  
ICT & Software Engineering  
Version: 1.2

## Version

Version	Date	Author	Changes	State
1	17-02-2023	M.G. den Hollander	Created the document and added styling.	Concept
1.1	12-04-2023	M.G. den Hollander	Started working on documenting the implementation	Concept
1.2	12-06-2023	M.G. den Hollander	Finished the implementation document	Complete

## Table of contents

Table of contents .....	3
1 Introduction .....	4
2 Implementation.....	5
Connection setup.....	5
UC-01 Checking unprocessed financial transactions.....	6
UC-02 Printing documents .....	8
UC-03 Data collection and processing.....	10
UC-04 Converting and sending documents .....	15
3 Testing .....	16



## 1 Introduction

Through the research conducted during the internship, it was determined that the best option for the company is to implement a different technique in order to view if it is possible to phase out the robot.

Once it was clear that an alternative to the robot had to be set up, requirements were made so that a prototype could be created. This implementation document contains all of the implementation steps that the alternative to the robot performs. This implementation will be linked to the requirements and use cases that the robot already performs, and the alternative will need to perform as well. All of these requirements and use cases can be found in the [analysis document](#).

It is hoped that this document will help the reader to better understand the implementation and will hopefully make it easier to make adjustments in the future when deemed necessary.



## 2 Implementation

### Connection setup

To establish a connection with the AS400 system, the ADODB (ActiveX Data Objects for .NET) technology was utilized. ADODB allows for the transmission and retrieval of SQL queries to and from the AS400. To incorporate ADODB into the project, a Microsoft-provided package was installed, as shown in Figure 1.



Figure 1 ADODB package

In addition, an App.config file was created to store the necessary connection string for the AS400 (Figure 2). By storing the connection string in a separate configuration file, sensitive information like usernames and passwords could be protected. For security reasons, the screenshot of the App.config file hides the actual username and password.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ConnectionString" value="Provider = IBMDA400; Data Source = SLIGR060; User Id = XXX; Password = XXX;" />
  </appSettings>
</configuration>
```

Figure 2 App.config

To retrieve the connection string from the App.config file, three Microsoft packages were installed (Figure 3). These packages provided the necessary extensions for accessing and utilizing the connection string within the code. The code snippet displayed in Figure 4 demonstrates how the connection string was obtained and prepared for establishing the connection.

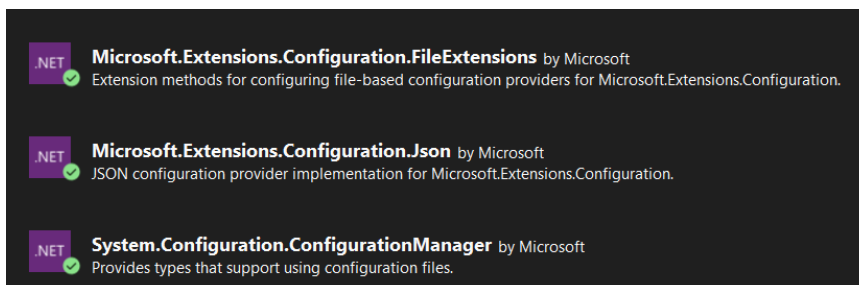


Figure 3 Extension packages

```
string connectionString = System.Configuration.ConfigurationManager.AppSettings["ConnectionString"];
```

Figure 4 Getting the connection string from the App.config file

By implementing the ADODB technology, utilizing the App.config file for storing the connection string, and installing the required Microsoft packages, a robust and secure connection to the AS400 system was established. These steps ensured the successful integration of the AS400 into the project, enabling efficient data transmission and retrieval.

## UC-01 Checking unprocessed financial transactions

This use case involves checking if certain rules are empty on the AS400. Previously, the robot performed this check visually. However, an alternative approach simplifies the process by using a straightforward SQL statement that should return 0. The code for this approach is as follows:

- Since there are multiple use cases that utilize SQL statements, it was decided to create an array of these queries. By consolidating the SQL statements into an array, it provides a convenient and organized way to manage and execute the queries. This approach enhances code reusability and maintainability, allowing for efficient implementation and management of multiple use cases that involve SQL queries.

Figure 5 shows an example of this array.

```
string[] queries = {  
    "SELECT SUM(FKBDIF) FROM SLGFILELIB.inkfktpf EXCEPTION JOIN SLGFILELIB.  
    "SELECT SUM(FKBDIF) FROM SLGFILELIB.inkfktpf WHERE STKDIF in (15, 90)"  
};
```

Figure 5 Array of strings

- To execute the first query, the following code displayed in Figure 6 is executed by opening a connection and retrieving number 1 from the array of queries. The result of this operation is stored in a recordset. This object allows for retrieving and manipulating data from a database.

```
Connection connection = new Connection();  
connection.ConnectionString = connectionString;  
connection.Open();  
Recordset recordset1 = new Recordset();  
recordset1.Open(queries[1], connection);
```

Figure 6 Connection opened and recordset created

- A timer is started (Figure 7). This is because the program needs to check every 15 minutes (Figure 8) for a duration of 1 hour whether the financial transactions have been processed in the meantime. The count goes up for every attempt.

```
int count = 0;  
Timer timer = new Timer((state) =>
```

Figure 7 Timer started

```
    TimeSpan.Zero, TimeSpan.FromMinutes(15)); // Retries every 15 minutes
```

Figure 8 Timer set to retry every 15 minutes

- If there are no more financial transactions, the application can proceed with the remaining steps of the process right away, as described in the following use cases (Figure 9).

```
if (recordset1 == null) // Good flow, no financial mutations  
{  
    PrintingDocuments(); // This section is for the 2nd use case  
    DataRetrieval(queries, results, connection); // This is for the 3rd use case  
    DigitalizingLists(); // This is for the 4th use case  
  
    connection.Close(); // Close the connection and exit the application  
    Environment.Exit(1);  
}
```

Figure 9 No more financial mutations

- As seen in Figure 10, if there are still financial transactions after an hour, the application needs to initiate a script named “mutations.vbs” to notify the relevant department. Afterward, the process will be halted.

```
else // Bad flow, still active mutations
{
    Console.WriteLine("Result is not 0. There are still active financial mutations.");
    if (count == 4) // 15 minutes x 4 = 1 hour
    {
        ProcessStartInfo startInfo = new ProcessStartInfo();
        startInfo.FileName = @"C:\Users\MarcdenHollanderTent\RPA\Scripts\mutations.vbs";
        startInfo.UseShellExecute = true;
        Process.Start(startInfo);
        Console.WriteLine("Starting active mutation script");
        Environment.Exit(1);
    }
}
```

Figure 10 Active mutations after 1 hour

- The script (Figure 11) that is initiated is fairly simple. The variables are defined at the beginning, followed by retrieving the current date, which is needed in the email header. The content of the text is defined based on the email body, which is a warning that the lines are not empty. Then, a new email is generated, all the necessary information is filled in, and finally, the email is sent.

```
Dim objOutlook, currentDate, dateText, emailBody

'get the current date
currentDate = Date()

'convert the date to text
dateText = CStr(currentDate)

emailBody = "Beste collega," & vbNewLine & vbNewLine & "Er zijn een of meerdere lijnen niet leeg."

Set objOutlook = CreateObject("Outlook.Application")
Set objMailItem = objOutlook.CreateItem(0)
objMailItem.Display 'can be removed to work in background
objMailItem.Recipients.Add ("marc.den.hollander@tentive.nl")
objMailItem.Subject = "Dagaansluiting " & dateText & " fout: Lijnen niet leeg"
objMailItem.Body = emailBody
objMailItem.Send
'objMailItem.Quit
Set objMailItem = nothing
Set objOutlook = nothing
```

Figure 11 VBS script

- The result of this code can be seen in Figure 12 on Outlook, marking the completion of this use case.

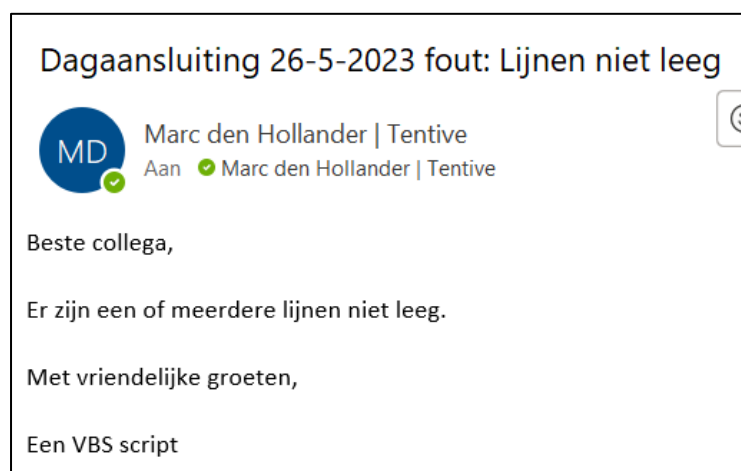


Figure 12 Email

## UC-02 Printing documents

This use case involves printing multiple documents on the AS400 system. These documents are printed through programs running on the AS400, as shown in Figure 13 below. A total of 4 documents need to be printed.



Figure 13 Printing documents

To achieve this use case, several approaches were attempted. Initially, an attempt was made to print these documents by executing remote commands through the existing ADODB connection. As seen in Figure 14, a string was created containing the command to execute stored procedures that call CL or RPG programs on the AS400. Unfortunately, this approach did not work, and the underlying reason could not be determined. Extensive problem solving and documentation reviews were conducted, but much of the documentation was outdated, and debugging was challenging due to generic error messages, as shown in Figure 15.

```
private static void PrintingDocuments(Connection connection)
{
    string nietgekoppeldeInkoopfacturen = "CALL PGM(QGPLSLIG/ACCC05-1) PARM('5000')";

    // Execute the command
    object recordsAffected = Type.Missing;
    connection.Execute(nietgekoppeldeInkoopfacturen, out recordsAffected, (int)CommandTypeEnum.adCmdText);

    Console.WriteLine("Command executed successfully.");
}
```

Figure 14 Execute remote command

**Exception: Unspecified error (0x80004005 (E\_FAIL))**

Figure 15 Generic Efail error

Following this unsuccessful attempt, a decision was made to explore a more visual option. This could be achieved using a useful software tool called AutoIt, which functions as a software capture recording tool. It is freeware, so it does not include any additional costs for the company.

As shown in Figure 16, this recorder tracks all user actions on the computer and translates them into code that can be used in the script. AutoIt utilizes its own scripting language, designed for automating tasks in the Windows environment. It offers a straightforward syntax and a wide range of built-in functions for automating various actions and interactions with GUIs and applications on the Windows operating system. AutoIt scripts can be compiled into executable files, which can then be initiated from the C# console application.



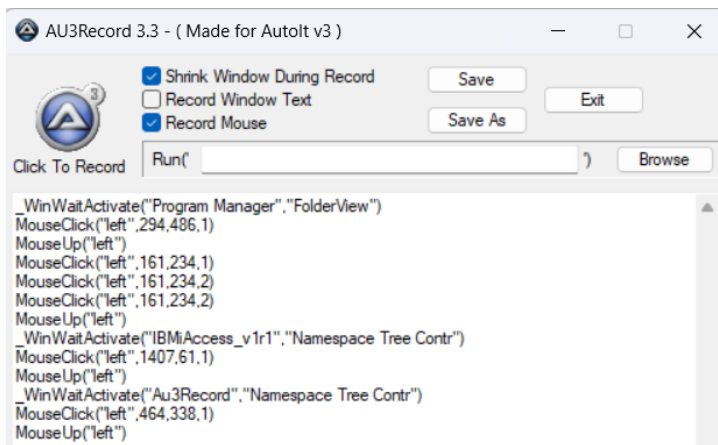


Figure 16 Autolt software recorder

The employees at Sligro utilize IBM i Access Client Solutions (Figure 17) to access the AS400 system through Windows. Since the company's servers also run on Windows, this provided the opportunity to use the Autolt recorder in combination with the emulator. This allows for the recording of all steps, including logging in, navigating to the appropriate menu options, and finally creating the documents.

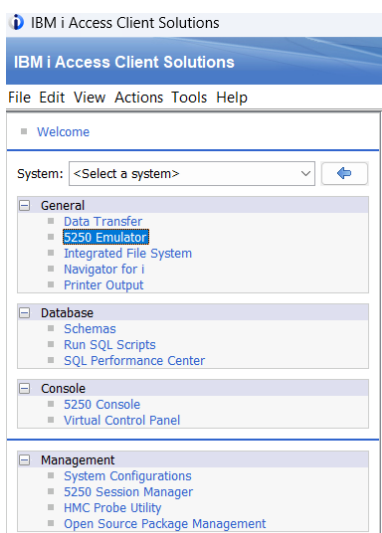


Figure 17 IBM i Access Client Solutions

The recorded script is then converted into an executable file using a tool provided by Autolt, as shown in Figure 18.

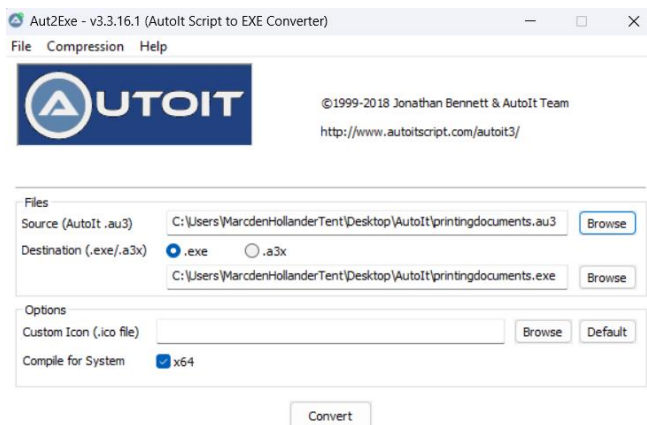


Figure 18 Autolt EXE converter

Lastly, the C# Console application needs to invoke this script. This is done in the same manner as invoking the mutation script from the previous use case, with the only difference being the adjustment of the filename to the appropriate file, as shown in Figure 19.

```
private static void PrintingDocuments()
{
    ProcessStartInfo startInfo = new ProcessStartInfo();
    startInfo.FileName = @"C:\Users\MarcdenHollanderTent\RPA\Scripts\printingdocuments.exe";
    startInfo.UseShellExecute = true;
    Process.Start(startInfo);
    Console.WriteLine("Printing documents");
}
```

Figure 19 Invoking the printingdocuments script

### UC-03 Data collection and processing

To execute this use case, multiple scripts have been developed and executed by the C# console application. The application retrieves the required values from the AS400 using SQL statements, facilitated by the connection that was set up. The code for this use case is as follows:

- In a "for-loop" method, a recordset is created for each query in the array that has been created for UC-01. The data is then collected and stored as an array of results. If the recordsets are empty, this means that either the SQL statements are incorrect, or the connection to the AS400 is not working properly.

```
private static void DataRetrieval(string[] queries, decimal[] results, Connection connection)
{
    for (int i = queries.Length - 4; i < queries.Length; i++) // For every query 1 recordset
    {
        Recordset recordset = new Recordset();
        recordset.Open(queries[i], connection);

        if (!recordset.EOF && !recordset.BOF)
        {
            results[i] = Convert.ToDecimal(recordset.Fields[0].Value);
        }
        else
        {
            results[i] = 0;
            Console.WriteLine("Warning: no results found. Please check the SQL statements and connection");
        }

        recordset.Close();
    }
}
```

Figure 20 Data retrieval method

- Lastly, as seen in Figure 21, a script is invoked, which is described later in this chapter. This script sends the retrieved results for further processing in an Excel document.

```
ProcessStartInfo startInfo = new ProcessStartInfo();
startInfo.FileName = @"C:\Users\MarcdenHollanderTent\RPA\Scripts\automation.vbs";
startInfo.Arguments = string.Join(" ", results);
startInfo.UseShellExecute = true;
Process.Start(startInfo);
Console.WriteLine("Starting script");
```

Figure 21 Invoking the automation script

The scripts required to perform the subsequent steps of this use case are written in Visual Basic (VBS) and Visual Basic for Applications (VBA). The first script, named "automation.vbs," written in VBS, performs the following tasks:

- As seen in Figure 22, all necessary variables are initialized at the beginning, and then the script checks for the values that are passed from the C# console application to this script.

```
Dim objWMIService, processes, arg, value, value1, value2, value3, value4
Set objWMIService = GetObject("winmgmts:\\.\root\cimv2")
Set processes = objWMIService.ExecQuery("SELECT * FROM Win32_Process WHERE Name='outlook.exe'")
Set args = WScript.Arguments

'Check if arguments are provided
If args.Count > 0 Then
    'Loop through the arguments
    For i = 0 To args.Count - 1
        'Get the value of each argument
        value = args(i)

        ' Assign the value to the corresponding variable
        Select Case i
            Case 0
                value1 = value
            Case 1
                value2 = value
            Case 2
                value3 = value
            Case 3
                value4 = value
            ' Add more cases for additional values if needed
        End Select
    Next
Else
    ' No arguments provided
    WScript.Echo "No arguments provided."
End If
```

Figure 22 Automation VBS

- Figure 23 shows that an instance of Excel is launched, and the pre-existing Excel file named "dagaansluiting" is located. This file contains a macro written in VBA that will place the passed variables in the right positions. Further details about this macro will

be described later in this chapter.

```
'Create Excel App Instance & Open Xlsm File
Set objExcelApp = CreateObject("Excel.Application")
objExcelApp.Visible = True
objExcelApp.DisplayAlerts = False

'Define Macro File & Path
sFilePathXlsm = "C:\Users\MarcdenHollanderTent\RPA\dagaansluiting.xlsm"
Set iWb = objExcelApp.Workbooks.Open(sFilePathXlsm)

'Run the macro with arguments
objExcelApp.Run "dailyquery", CDb1(value1), CDb1(value2), CDb1(value3), CDb1(value4)

'Save & Close file
iWb.Save
iWb.Close
objExcelApp.DisplayAlerts = True
objExcelApp.Quit
```

Figure 23 Starting up Excel

- Finally, In Figure 24, the script performs a check to ensure that Outlook is running. If not, it is initiated, and then the email script is executed, which sends the processed Excel document to the appropriate recipients.

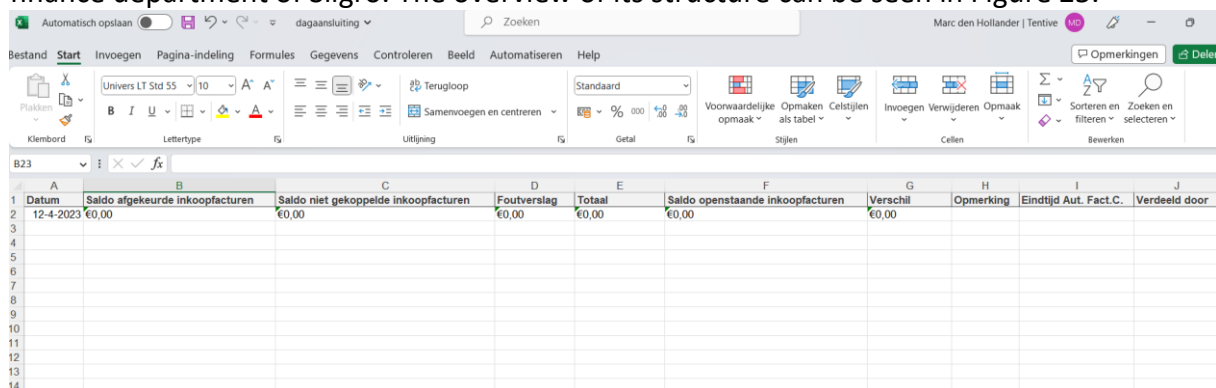
```
'Create a shell object
Set objShell = CreateObject("WScript.Shell")

'Check if outlook is running
If processes.Count = 0 Then
objShell.run "outlook.exe"
End If

'Run the mail script
objShell.Run "C:\Users\MarcdenHollanderTent\RPA\Scripts\mail.vbs"
```

Figure 24 Checking for an Outlook instance

As mentioned earlier, an Excel document has been created to serve as a report for the finance department of Sligro. The overview of its structure can be seen in Figure 25.



	A	B	C	D	E	F	G	H	I	J
	Datum	Saldo afgekeurde inkoopfacturen	Saldo niet gekoppelde inkoopfacturen	Foutverslag	Totaal	Saldo openstaande inkoopfacturen	Verschil	Opmerking	Eindtijd Aut. Fact.C.	Verdeeld door
1	12-4-2023	€0,00	€0,00	€0,00	€0,00	€0,00	€0,00			
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										

Figure 25 Excel report

For the sake of confidentiality, the amounts in the screenshot have been set to 0 euros. Only the empty document will be shared at the end of the internship. To populate this report with the retrieved results, a macro has been implemented using VBS. This macro, named "DailyQuery," performs the following tasks:

- It initializes the necessary variables and retrieves the current date. To ensure clarity, the corresponding values for each column are clearly indicated (Figure 26).

```
Sub DailyQuery(value1 As Double, value2 As Double, value3 As Double, value4 As Double)
    'Saldo afgekeurde inkoopfacturen value1
    'Saldo niet gekoppelde inkoopfacturen value2
    'Foutverslag value3
    'Saldo openstaande inkoopfacturen value4

    Dim dateValue As String
    Dim total As Double
    Dim verschil As Double
    Dim lastRow As Long
    Dim fso As Object

    ' get current date in required format
    dateValue = Format(Date, "mm/dd/yyyy")
End Sub
```

Figure 26 Defining variables and current date

- The total amount is calculated by summing up the values, excluding the "Saldo openstaande inkoopfacturen" (Figure 27).

```
' calculate totaal
totaal = (value1 + value2 + value3)
```

Figure 27 Calculating total amount

- The difference is calculated, which is an essential step for subsequent handling via email. Further details on this will be addressed in a later part of the script (Figure 28).

```
' calculate difference (verschil)
If totaal - value4 > 99999999 Then
    verschil = totaal - value4 - 100000000
Else
    verschil = totaal - value4
End If
```

Figure 28 Calculated difference amount

- This step involves placing the values in the right positions within the document. When executed the next day, new values will be inserted on the following line, along with the corresponding date (Figure 29).

```
' insert results into worksheet
lastRow = Sheets("dagelijkse aansluiting").Cells(Rows.Count, 1).End(xlUp).Row
Sheets("dagelijkse aansluiting").Cells(lastRow + 1, 1) = dateValue
Sheets("dagelijkse aansluiting").Cells(lastRow + 1, 2).Value = Format(value1, "€#,##0.00")
Sheets("dagelijkse aansluiting").Cells(lastRow + 1, 3).Value = Format(value2, "€#,##0.00")
Sheets("dagelijkse aansluiting").Cells(lastRow + 1, 4).Value = Format(value3, "€#,##0.00")
Sheets("dagelijkse aansluiting").Cells(lastRow + 1, 5).Value = Format(totaal, "€#,##0.00")
Sheets("dagelijkse aansluiting").Cells(lastRow + 1, 6).Value = Format(verschil, "€#,##0.00")

Set fso = CreateObject("Scripting.FileSystemObject")
```

Figure 29 Placing the values in Excel

- In the final step of this macro, the difference value is stored in a .txt file. The reason behind this is that the value cannot be directly returned from the Excel macro for later handling via email. Hence, a simple approach was adopted by writing the value

to a .txt file. This value is overwritten daily (Figure 30).

```
' Set the file path and name
Dim filePath As String
filePath = "C:\Users\MarcdenHollanderTent\RPA\Temp\value.txt"

' Write value verschil to temp file
Dim file As Object
Set file = fso.CreateTextFile(filePath, True)
file.WriteLine verschil
file.Close
```

Figure 30 Write difference to .txt file

The final script, named "mail.vbs," performs the action previously mentioned, which involves sending an email containing the complete “dagaansluiting” report. The script carries out the following tasks:

- All necessary variables are created at the beginning. Next, the difference calculated in the Excel macro is retrieved from the "value.txt" file. If there is an actual difference, it will affect the content of the email in a later step of this script. Additionally, the current date is retrieved, which needs to be mentioned by default (Figure 31).

```
Dim objOutlook, currentDate, dateText, emailBody, verschil, fso, file, filePath

'get verschil from the temp folder
Set fso = CreateObject("Scripting.FileSystemObject")
filePath = "C:\Users\MarcdenHollanderTent\RPA\Temp\value.txt"
Set file = fso.OpenTextFile(filePath)
verschil = file.ReadLine
file.Close

'get the current date
currentDate = Date()

'convert the date to text
dateText = CStr(currentDate)
```

Figure 31 Defining variables and current date

- As mentioned in the previous step, the email content is adjusted if there is a difference calculated. This notifies the recipients who then need to perform their own manual actions based on this difference (Figure 32).

```
'Fill the email body based on verschil
If verschil < 0 Or verschil > 0 Then
emailBody = "Beste collega," & vbNewLine & vbNewLine & "De dagaansluiting is helaas met een verschil voltooid. Zie de bijlage." & vbNewLine & vbNewLine & "Met vriendelijke groeten"
Else
emailBody = "Beste collega," & vbNewLine & vbNewLine & "De dagaansluiting is succesvol voltooid. Zie de bijlage." & vbNewLine & vbNewLine & "Met vriendelijke groeten"
End If
```

Figure 32 Email content based on calculated difference

- The final step of this script fills in the recipient, sets the email subject to the retrieved date, and attaches the Excel document to the email (Figure 33). After this, the email is sent, marking the completion of the use case.

```
Set objOutlook = CreateObject("Outlook.Application")
Set objMailItem = objOutlook.CreateItem(0)
objMailItem.Display 'can be removed to work in background
objMailItem.Recipients.Add ("marc.den.hollander@tentive.nl")
objMailItem.Subject = "Dagaansluiting " & dateText
objMailItem.Body = emailBody
objMailItem.Attachments.Add "C:\Users\MarcdenHollanderTent\RPA\dagaansluiting.xlsm"
objMailItem.Send
'objMailItem.Quit
Set objMailItem = nothing
Set objOutlook = nothing
```

Figure 33 Sending the email

- The result of this script can be found in Figure 34.

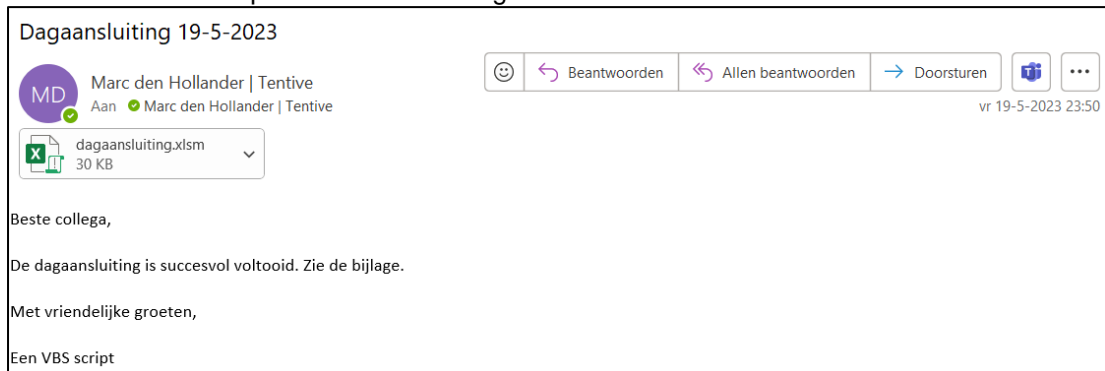


Figure 34 Email with the report as an attachment

## UC-04 Converting and sending documents

This final use case involves invoking the SNDNETSPLF procedure on the AS400 system as seen in Figure 35. This sends the required documents to the right network disks, and converts them to pdf's.



Figure 35 SNDNETSPLF procedure

The implementation for this use case is similar to that of UC-02. Initially, an attempt was made to invoke the procedure using a remote command, as shown in Figure 35. This procedure needs to be called once to place the files in the appropriate output queue, and once to convert the files to .PDF format on the AS400.

```
private static void DigitalizingLists(Connection connection)
{
    string BST071 = "CALL QSYS2.QCMDEXC(SNDNETSPLF FILE(BST071) TOUSRID(ADMINP5 SLIGR060)";
    string BST028 = "CALL QSYS2.QCMDEXC(SNDNETSPLF FILE(BST028) TOUSRID(ADMINP5 SLIGR060)";

    // Execute the command
    object recordsAffected = Type.Missing;
    connection.Execute(BST071, out recordsAffected, (int)CommandTypeEnum.adCmdText);
    connection.Execute(BST028, out recordsAffected, (int)CommandTypeEnum.adCmdText);

    //conversion to PDF
    string BST071PDF = "CALL QSYS2.QCMDEXC(SNDNETSPLF FILE(BST071) TOUSRID(ADMINP2PDF SLIGR060)";
    string BST028PDF = "CALL QSYS2.QCMDEXC(SNDNETSPLF FILE(BST028) TOUSRID(ADMINP2PDF SLIGR060)";
    connection.Execute(BST071PDF, out recordsAffected, (int)CommandTypeEnum.adCmdText);
    connection.Execute(BST028PDF, out recordsAffected, (int)CommandTypeEnum.adCmdText);
}
```

Figure 36 Remote commands

For this use case, automation was also chosen using an AutoIt script. The software recorder was once again utilized to record the necessary steps, and the script was then converted into an executable file. As shown in Figure 37, the application "digitalizinglist.exe" is invoked, and the tasks are executed.

```
private static void DigitalizingLists()
{
    ProcessStartInfo startInfo = new ProcessStartInfo();
    startInfo.FileName = @"C:\Users\MarcdenHollanderTent\RPA\Scripts\digitalizinglists.exe";
    startInfo.UseShellExecute = true;
    Process.Start(startInfo);
    Console.WriteLine("Digitalizing lists");
}
```

Figure 37 Invoking the digitalizinglists script

### 3 Testing

While testing written code is usually important, it turns out it is not highly applicable to this project. Since it involves data retrieval and script invocation, there isn't much logic to test apart from the final result. The demonstration will be used to show that all use cases work as intended.

In this project, the focus is not on extensively testing the internal logic of the code. Instead, the main concern is ensuring that the overall process functions correctly and produces the desired outcomes. The key objective is to demonstrate that the system can successfully retrieve data and execute scripts next to the current RPA automation, resulting in the expected results.

During the demo, all use cases will be showed to ensure they operate as expected. This includes validating accurate data retrieval, correct script invocation, and successful actions.





The demo serves as a comprehensive showcase of the project's functionality, confirming that it performs as intended and meets the project's requirements.

In conclusion, while testing written code is typically important, it is less relevant in this project due to its focus on data retrieval and script invocation. The project's success relies on demonstrating the effectiveness of the combined process through a comprehensive demo, where all use cases are showcased and validated.

