**Guidance Document: Commercial**

Automated Commercial Invoices





**Contents**

[1.0 Overview 3](#_Toc193887881)

[2.0 Maintenance 3](#_Toc193887882)

[3.0 Data Governance 4](#_Toc193887883)

[4.0 Detailed Design 4](#_Toc193887884)

[4.1 Design of reports 4](#_Toc193887885)

[4.2 Design of flows 7](#_Toc193887886)

[4.3 Design of scripts 9](#_Toc193887887)

[5.0 Appendix 14](#_Toc193887888)

[5.1 Business Objects material 14](#_Toc193887889)

[5.2 Power Automate flows 18](#_Toc193887890)

[5.3 Office Scripts 20](#_Toc193887891)

1. **Overview**

#### 1.1 The Commercial Team requires invoices to be produced for each customer on the 15th of each month. All orders received between the 15th of the current and previous months must be billed to generate revenue for the department.

#### 1.2 This business process improvement project aims to address the delays associated with the generation, transformation, and delivery of commercial invoices. The benefits of which are the savings of 30 hours of manual processing time per month, reducing the time spent issuing commercial invoices by 96.8%.

#### 1.3 The requirements for an invoice were as follows:

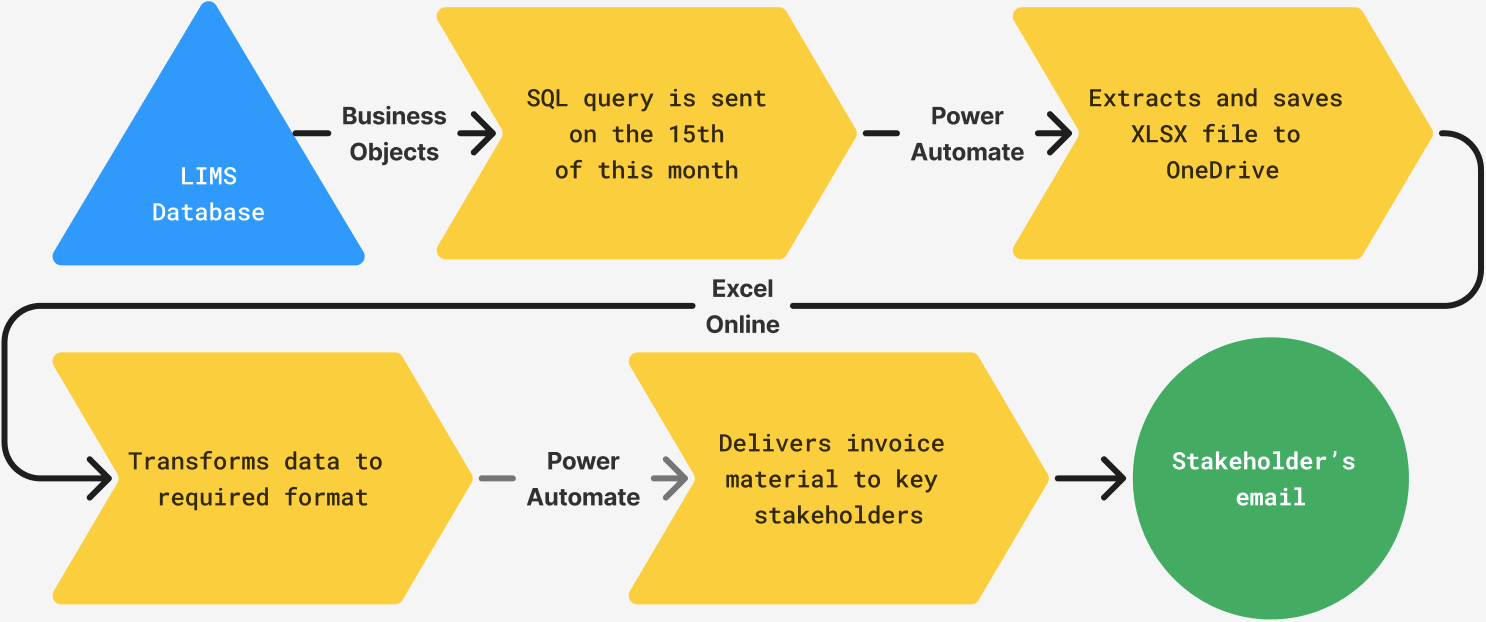
* A raw data spreadsheet of all received orders within the last billing cycle
* An invoice sheet where expenses were billed either by the unit or suite costs

#### 1.4 This was achieved using 2 flows capturing 5 precise SQL reports. The results of which were delivered to Enquiries.Commercial inbox where the material can be reviewed before issuing to customers.

#### 1.5 Data from LIMS relating to commercial customers was extracted using Business Objects and received in David.Golacis’ inbox on the 15th of each month.

Keywords from the email’s title activated a Power Automate flow, allowing the attached XLSX file to be saved to OneDrive. The files were processed and transformed in the cloud using 2 TypeScript programmes per flow through Excel Online API calls.

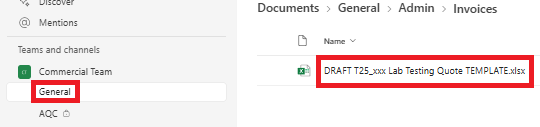
#### 1.6 Process map for the order of operations:



# 2.0 Maintenance

#### 2.1 A yearly template of the relevant customer’s quote sheet must be updated upon contract renewal. The location of this spreadsheet:

**Teams: Commercial Team/ Documents/ General/ Admin/ Invoices**



#### 2.2 As of present, this quote is only used to calculate the sum of analytical costs for 2 customers. These customers were Eden Springs and ELGA 2030.

# 3.0 Data Governance

#### 3.1 All Business Objects files (SQL reports) are stored online at [Affinity's BO Portal](https://boe.grpdom.vwuk.corp/BOE/BI) within this location:

**Public Folders/ Sample Manager/ Commercial Invoices**

A screenshot of a computer

AI-generated content may be incorrect.

#### 3.2 Queries, flows, and Office scripts are provided in the [appendix](#_5.0_Appendix).

# 4.0 Detailed Design

## 4.1 Design of reports

#### 4.1.1 Business Objects generated and delivered scheduled queries from an Oracle database.

#### 4.1.2 From the database the following 3 tables were used: sample, test, and result.

These were inner joined together to minimise the chance of anomalous records from being included. Records which did not comply with all constraints were eliminated without requiring additional lines of SQL.

The following database features were used throughout this project:

|  |  |  |  |
| --- | --- | --- | --- |
| Table | Property | Description | Type |
| Sample | id\_numeric | Foreign key to join with test.sample; one-to-one relationship | Number |
|  | recd\_date | Date of when the order was received | Date-time |
|  | customer\_id | Which customer does this work belong to | String |
|  | status | The progression of job status | String |
| Result | test\_number | Foreign key to join with test.test\_number; one-or-many-to-one | Number |
|  | name | Name of the parameter | String |
|  | rep\_control | Boolean to show the parameter on the final certificate | Date-time |
| Test | test\_number | Foreign key to join with result.test\_number; one-to-one-or-many relationship | Number |
|  | sample | Foreign key to join with sample.id\_numeric; one-to-one relationship | Number |
|  | analysis | Name of the test | String |

Logical entity-relationship diagram:

A screenshot of a computer

AI-generated content may be incorrect.

#### 4.1.3 These queries shared safety features that restricted the data pulled from the cloud, reducing the server's memory usage and improving processing speed.

Techniques used to hone searches were:

* Limiting the date range used:

WHERE

( sample.recd\_date BETWEEN

TRUNC ( ADD\_MONTHS ( SYSDATE, -1 ), 'MM' ) + 14

AND TRUNC ( SYSDATE, 'MM' ) + 14 )

* Fetching samples with an associated customer ID:

WHERE

TRIM ( sample.customer\_id ) IS NOT NULL

* Filtering out results which are not reported:

WHERE

( result.rep\_control <> 'N' OR test.analysis = 'MATRIX' )

* Specifying which sample status was required:

WHERE

sample.status IN ( 'X' )

#### 4.1.4 An example selection is shown below of a count of suites table:

|  |  |  |
| --- | --- | --- |
| **Customer ID** | **Template ID** | **Count of Samples** |
| EDEN\_SPR | RE\_EDEN1 | 3 |
| EDEN\_SPR | RE\_EDEN10 | 5 |
| EDEN\_SPR | RE\_EDEN12 | 15 |

# 4.2 Design of flows

#### 4.2.1 Once an email containing a report has been received, a Power Automate flow attempts to match the email's title to keywords. If a match is found, a series of steps take place to save the attached XLSX file for processing.

To ensure reliability, conditional filters were used to eliminate potential problems that could occur during an action. Considerations included were:

* Confirmation of email requirements:

A screenshot of a computer

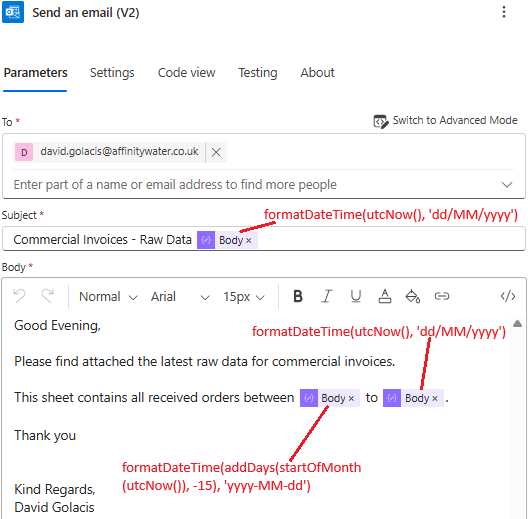
AI-generated content may be incorrect.

* Check for attachment(s):

A screenshot of a computer

Description automatically generated

The transformed invoice material was then sent to Enquiries.Commercial inbox stating the limits within the email:

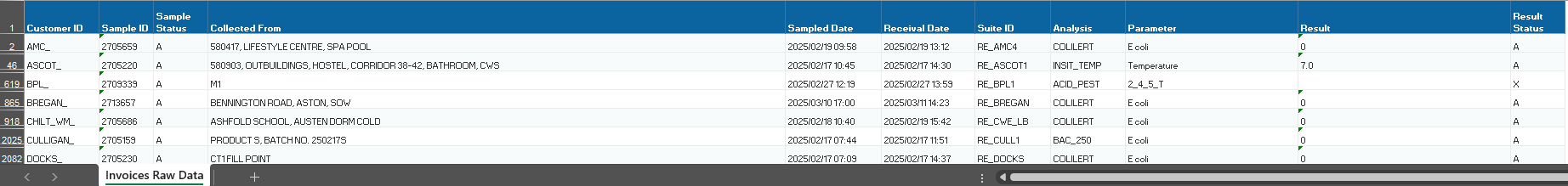


# 4.3 Design of scripts

#### 4.3.1 Power Automate provides access to Excel Online for the use of Office Scripts, enabling functions to be written and executed on queries.

#### 4.3.2 Monthly raw data

First, the initial query required the customers to be separated into their own, unique worksheets.



The processing of the monthly raw data query began by looping through all the present customers in the table and separating each customer’s data into an array.

for (let customer of splitCustomers) {

let tempArray: string[][] = [];

This was achieved by filtering the data by matching the Customer ID property of a record to the search string in the loop, resulting in positive matches being inserted into an empty array.

if (nestedObjects[a]['Customer ID'] === customer) {

let values = Object.values(nestedObjects[a]);

tempArray.push(values);

}

If there was data in the array, then a new worksheet was generated and the array contents was pasted in place.

if (tempArray.length > 0) {

let lastRow = tempArray.length;

let lastColumn = columnToLetter(tempArray[0].length);

let createSheet = workbook.addWorksheet(customer);

let selectSheet = workbook.getWorksheet(customer);

selectSheet.getRange(`A1:${lastColumn}${lastRow}`).setValues(tempArray);

}

During this step, a bug was also fixed by converting the date format from a number to a long date.

selectSheet.getRange('E:F').setNumberFormatLocal('[$-en-GB]dd mmm yyyy  hh:mm');

Upon completion of all customers, the original sheet was deleted, leaving only individual customer sheets. This produced the following sheet:

A screenshot of a computer

AI-generated content may be incorrect.

Next, the sheets were looped through to pivot the results into a wide format allowing for simple counting of each analysis.

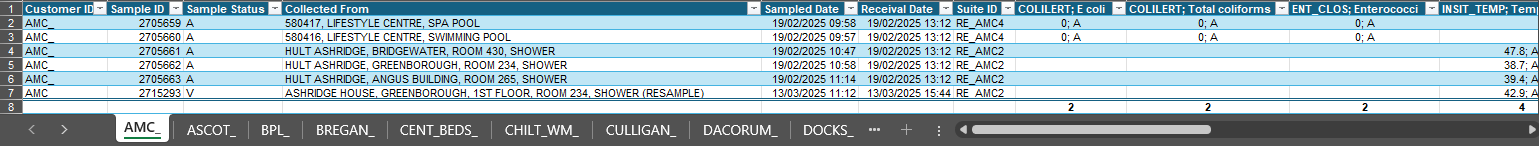
For the pivot function, the parameter, result value, and result status columns were extracted and used for pivoting, producing this result:

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer ID** | **Sample ID** | **COLILERT; E coli** | **COLILERT; Total coliforms** |
| AMC\_ | 2705659 | 0; A | 0; A |
| AMC\_ | 2705660 | 0; A | 0; A |

By pairing the value with the status of the result, one can easily see if the result was valid, i.e. wasn’t QC rejected, nulling the value of the test for the customer. To help with the visibility of this problem, a conditional filter was added to highlight these issues for the end users, where results in status ‘R’, ‘X’, or ‘U’ appeared prominently in red colouring.

|  |  |  |
| --- | --- | --- |
| **Customer ID** | **Sample ID** | **LEGIONELLA; L pneumophila gp 2-14** |
| AMC\_ | 2705663 | 0; A |
| AMC\_ | 2715293 | ; U |

A formula to count the range between the header and last row was placed below the table for immediate insight of each parameter’s quantities, achieving this result:



A tally was placed below the last row to be validated by a responsible person before transferring outputs to an invoice sheet for billing.

#### 4.3.4 Suite-based billing

A template quote document containing prices was stored on Teams for extraction of prices for this task. A dirty approach was used to achieve this, which could break this function if the main columns were misaligned.

The quote doc:

A screenshot of a computer

AI-generated content may be incorrect.

The range was looped until 3 columns returned blank results, indicating the end of the list.

if (position > 8 && rangeText[position][0] === ''

&& rangeText[position][1] === '' && rangeText[position][3] === ''

&& rangeText[position][6] === '') {

break;

}

Elements at each index were validated to confirm if data was present. These were added to a new list for easier manipulation.

// Loop to capture all analytical rates

for (let position = 5; position < length2; position++) {

//console.log(rangeText[b]);

// If parameter and cost columns are not blank, add it to analysis list

if (rangeText[position][1] !== '' && rangeText[position][6] !== '') {

analyses.push(rangeText[position]);

continue;

}

// If parameter and limits columns are not blank, add it to the analysis list

if (rangeText[position][1] !== '' && rangeText[position][3] !== '') {

analyses.push(rangeText[position]);

continue;

}

// If suite and limits columns are not blank, add it to the analysis list

if (rangeText[position][0] !== '' && rangeText[position][3] !== '') {

analyses.push(rangeText[position]);

continue;

}

}

Items which didn’t have a cost property were appended the result which did previously, assuming this grouping to be a suite of tests under a fixed cost, enabling this pricing method for a series of parameters.

// Loop through costs data...

while (analysesLength--) {

// If cost is blank, append to the following parameter

if (analyses[analysesLength][2] === '') {

analyses[analysesLength - 1][1] = analyses[analysesLength - 1][1]

.concat(`, ${analyses[analysesLength][1]}`);

// Remove final line

analyses.splice(analysesLength, 1);

}

}

The extracted result was a string of nested JSON objects containing information about the suite, parameter, and cost:

|  |  |  |
| --- | --- | --- |
| Suite | Determinand | Rate |
| BROMS | Bromate as BrO3, Bromide as Br, Chlorate as ClO3, Chlorite as ClO2 | 55.09 |
| COLOUR | Colour | 5.33 |
| COT | Quantitative Odour | 26.28 |

A similar approach was used to draw the counts and parameters tables from the initialising query.

Counts table:

|  |  |  |
| --- | --- | --- |
| **Customer ID** | **Template ID** | **Count of Samples** |
| EDEN\_SPR | RE\_EDEN1 | 3 |
| EDEN\_SPR | RE\_EDEN10 | 5 |
| EDEN\_SPR | RE\_EDEN12 | 15 |
| EDEN\_SPR | RE\_EDEN13 | 30 |

Parameters table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer ID** | **Template ID** | **Analysis** | **Parameter** |
| EDEN\_SPR | RE\_EDEN1 | BAC\_250 | E coli |
| EDEN\_SPR | RE\_EDEN1 | BAC\_250 | Total coliforms |
| EDEN\_SPR | RE\_EDEN1 | ENTCLO\_250 | Enterococci |
| EDEN\_SPR | RE\_EDEN1 | PS\_AER\_250 | Pseudomonas aeruginosa |

The parameter’s names were modified to match the quote document for precise selection. The reason the parameters were chosen over the quote document was because the SQL query will always produce consistent outputs, while the costs documents could gain transcription errors with each iteration, making string amendment errors challenging to spot.

if (rangeText[a][3].includes('Total Coliforms') === true) {

rangeText[a][3] = 'Total coliforms';

}

Two loops were used to identify which parameters were in each template. The parameter was then checked against a loop of the cost’s objects. Positive matches between the parameter’s name and cost’s name were used to identify the parameter’s rate, adding the value to a running total for this template’s expenses.

Suites of tests were identified by the comma in the string, which was used to loop through the items.

// Apply suite cost if multiple parameters

if (splitSuite.length > 1) {

for (let d = 0; d < splitSuite.length; d++) {

if (splitAnalysis[0] === costsData[c]['Suite']

&& splitAnalysis[1] === splitSuite[d]) {

suiteCheck = true;

multipleAnalyses.push(splitAnalysis[0]);

//console.log(`Adding ${costsData[c]['Rate']} of

${splitAnalysis[1]}, suite and parameter matched (${splitSuite[d]})`);

suiteCost += Number(costsData[c]['Rate']);

break;

}

}

}

The name of the suite was appended to a list outside this costs loop to compare to the next item in the parameter’s list; if this suite was identified in the next iteration, the parameter will be rejected and not accounted for.

// Check if current parameter is part of a processed suite

for (let e = 0; e < multipleAnalyses.length; e++) {

if (splitAnalysis[0] === multipleAnalyses[e]) {

alreadyProcessedSuite = true;

}

}

The template’s costs were totalled and reported per unit price along with the total price for all templates of this kind. Any missed parameters from the bill were entered adjacent.

The resulting table for the customer:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Customer ID** | **Template ID** | **Count of Samples** | **Unit Price (£)** | **Total Cost (£)** | **Exceptions** |
| EDEN\_SPR | RE\_EDEN1 | 3 | 52.70 | 158.10 |  |
| EDEN\_SPR | RE\_EDEN10 | 5 | 20.92 | 104.60 |  |
| EDEN\_SPR | RE\_EDEN12 | 15 | 23.52 | 352.80 |  |
| EDEN\_SPR | RE\_EDEN13 | 30 | 39.21 | 1176.30 |  |
| EDEN\_SPR | RE\_EDEN2 | 114 | 114.55 | 13058.70 |  |
| EDEN\_SPR | RE\_EDEN3 | 1 | 1164.33 | 1164.33 | KONE; Nitrite Nitrate Formula,SUB\_BPA; Bisphenol A |

# 5.0 Appendix

### 5.1 Business Objects material

#### 5.1.1 Monthly raw data

SQL query:

**SELECT**

TRIM ( sample.customer\_id ),

TRIM ( sample.id\_numeric ),

TRIM ( sample.status ),

TRIM ( sample.collected\_from ),

sample.sampled\_date,

sample.recd\_date,

TRIM ( sample.template\_id ),

TRIM ( test.analysis ),

TRIM ( result.name ),

TRIM ( result.text ),

TRIM ( result.status )

**FROM**

sample

INNER JOIN test

ON test.sample = sample.id\_numeric

INNER JOIN result

ON result.test\_number = test.test\_number

**WHERE**

( sample.recd\_date BETWEEN

TRUNC ( ADD\_MONTHS ( SYSDATE, -1 ), 'MM' ) + 14

AND TRUNC ( SYSDATE, 'MM' ) + 14 )

AND TRIM ( sample.customer\_id ) IS NOT NULL

AND ( result.rep\_control <> 'N' OR test.analysis = 'MATRIX' )

AND sample.status NOT IN ( 'X' )

Report:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Customer ID** | **Sample ID** | **Sample Status** | **Collected From** | **Sampled Date** | **Receival Date** | **Suite ID** | **Analysis** | **Parameter** | **Result** | **Result Status** |
| AMC\_ | 2705659 | A | 580417, LIFESTYLE CENTRE, SPA POOL | 2025/02/19 09:58 | 2025/02/19 13:12 | RE\_AMC4 | COLILERT | E coli | 0 | A |
| AMC\_ | 2705659 | A | 580417, LIFESTYLE CENTRE, SPA POOL | 2025/02/19 09:58 | 2025/02/19 13:12 | RE\_AMC4 | COLILERT | Total coliforms | 0 | A |
| AMC\_ | 2705659 | A | 580417, LIFESTYLE CENTRE, SPA POOL | 2025/02/19 09:58 | 2025/02/19 13:12 | RE\_AMC4 | ENT\_CLOS | Enterococci | 0 | A |
| AMC\_ | 2705659 | A | 580417, LIFESTYLE CENTRE, SPA POOL | 2025/02/19 09:58 | 2025/02/19 13:12 | RE\_AMC4 | MATRIX | Sample details supplied by | Affinity Water | A |
| AMC\_ | 2705659 | A | 580417, LIFESTYLE CENTRE, SPA POOL | 2025/02/19 09:58 | 2025/02/19 13:12 | RE\_AMC4 | MATRIX | Water Matrix | Recreational Water | A |

#### 5.1.2 Suites-based billing

Customer 1 suites SQL query:

**SELECT**

sample.customer\_id,

sample.template\_id,

COUNT ( sample.id\_numeric )

**FROM** sample

**WHERE**

( sample.recd\_date BETWEEN

TRUNC ( ADD\_MONTHS ( SYSDATE, -1 ), 'MM' ) + 14

AND TRUNC ( SYSDATE, 'MM' ) + 14 )

AND sample.customer\_id = 'EDEN\_SPR'

AND sample.status NOT IN ( 'X', 'R' )

**GROUP** **BY** sample.customer\_id, sample.template\_id

Report:

|  |  |  |
| --- | --- | --- |
| **Customer ID** | **Template ID** | **Count of Samples** |
| EDEN\_SPR | RE\_EDEN1 | 3 |
| EDEN\_SPR | RE\_EDEN10 | 5 |
| EDEN\_SPR | RE\_EDEN12 | 15 |
| EDEN\_SPR | RE\_EDEN13 | 30 |
| EDEN\_SPR | RE\_EDEN2 | 114 |
| EDEN\_SPR | RE\_EDEN3 | 1 |

Customer 1 suite breakdown SQL query:

**SELECT**

sample.customer\_id,

sample.template\_id,

test.analysis,

CASE TRIM ( test.analysis )

WHEN 'ACID\_PEST'

THEN 'ACID Suite'

WHEN 'OCPP'

THEN 'OCP Suite'

WHEN 'ONP\_PEST'

THEN 'ONS Suite'

WHEN 'PAH'

THEN 'PAH Suite'

WHEN 'TRZ\_URON'

THEN 'TRZ\_URON Suite'

WHEN 'VOC'

THEN 'VOC Suite'

ELSE result.name

END AS name

**FROM** sample

INNER JOIN test

ON test.sample = sample.id\_numeric

INNER JOIN result

ON result.test\_number = test.test\_number

**WHERE**

( sample.recd\_date BETWEEN

TRUNC ( ADD\_MONTHS ( SYSDATE, -1 ), 'MM' ) + 14

AND TRUNC ( SYSDATE, 'MM' ) + 14 )

AND sample.customer\_id = 'EDEN\_SPR'

AND sample.status NOT IN ('X', 'R')

AND result.rep\_control <> 'N'

AND test.analysis NOT IN ( 'NON\_CONF\_S' )

Report:

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer ID** | **Template ID** | **Analysis** | **Parameter** |
| EDEN\_SPR | RE\_EDEN2 | BAC\_250 | E coli |
| EDEN\_SPR | RE\_EDEN2 | BAC\_250 | Total coliforms |
| EDEN\_SPR | RE\_EDEN2 | BROMS | Bromate as BrO3 |
| EDEN\_SPR | RE\_EDEN2 | ENTCLO\_250 | Enterococci |
| EDEN\_SPR | RE\_EDEN2 | PS\_AER\_250 | Pseudomonas aeruginosa |
| EDEN\_SPR | RE\_EDEN2 | TVC | 1 day plate count 37C |

Customer 2 suites SQL query:

**SELECT**

sample.customer\_id,

sample.template\_id,

COUNT ( sample.id\_numeric )

**FROM** sample

**WHERE**

( sample.recd\_date BETWEEN

TRUNC ( ADD\_MONTHS ( SYSDATE, -1 ), 'MM' ) + 14

AND TRUNC ( SYSDATE, 'MM' ) + 14 )

AND sample.customer\_id = 'ELGA\_2030'

AND sample.status NOT IN ( 'X', 'R' )

**GROUP** **BY** sample.customer\_id, sample.template\_id

Report:

|  |  |  |
| --- | --- | --- |
| **Customer ID** | **Template ID** | **Count of Samples** |
| ELGA\_2030 | RE\_DENTAL | 6 |
| ELGA\_2030 | RE\_ENDO | 58 |
| ELGA\_2030 | RE\_ENT | 2 |
| ELGA\_2030 | RE\_RENAL | 4 |
| ELGA\_2030 | RE\_SSD | 10 |
| ELGA\_2030 | RE\_SSDMYCO | 11 |

Customer 2 suite breakdown SQL query:

**SELECT**

sample.customer\_id,

sample.template\_id,

test.analysis,

CASE TRIM ( test.analysis )

WHEN 'ACID\_PEST'

THEN 'ACID Suite'

WHEN 'OCPP'

THEN 'OCP Suite'

WHEN 'ONP\_PEST'

THEN 'ONS Suite'

WHEN 'PAH'

THEN 'PAH Suite'

WHEN 'TRZ\_URON'

THEN 'TRZ\_URON Suite'

WHEN 'VOC'

THEN 'VOC Suite'

ELSE result.name

END AS name

**FROM** sample

INNER JOIN test

ON test.sample = sample.id\_numeric

INNER JOIN result

ON result.test\_number = test.test\_number

**WHERE**

( sample.recd\_date BETWEEN

TRUNC ( ADD\_MONTHS ( SYSDATE, -1 ), 'MM' ) + 14

AND TRUNC ( SYSDATE, 'MM' ) + 14 )

AND sample.customer\_id = 'ELGA\_2030'

AND sample.status NOT IN ('X', 'R')

AND result.rep\_control <> 'N'

AND test.analysis NOT IN ( 'NON\_CONF\_S' )

Report:

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer ID** | **Template ID** | **Analysis** | **Parameter** |
| ELGA\_2030 | RE\_DENTAL | HEAVY\_MET | Heavy Metals |
| ELGA\_2030 | RE\_DENTAL | ICP | Cadmium as Cd |
| ELGA\_2030 | RE\_DENTAL | ICP | Cobalt as Co |
| ELGA\_2030 | RE\_DENTAL | ICP | Iron as Fe |
| ELGA\_2030 | RE\_DENTAL | ICP | Sodium as Na |
| ELGA\_2030 | RE\_DENTAL | KONE | Chloride as Cl |
| ELGA\_2030 | RE\_DENTAL | KONE | Silica as SiO2 |

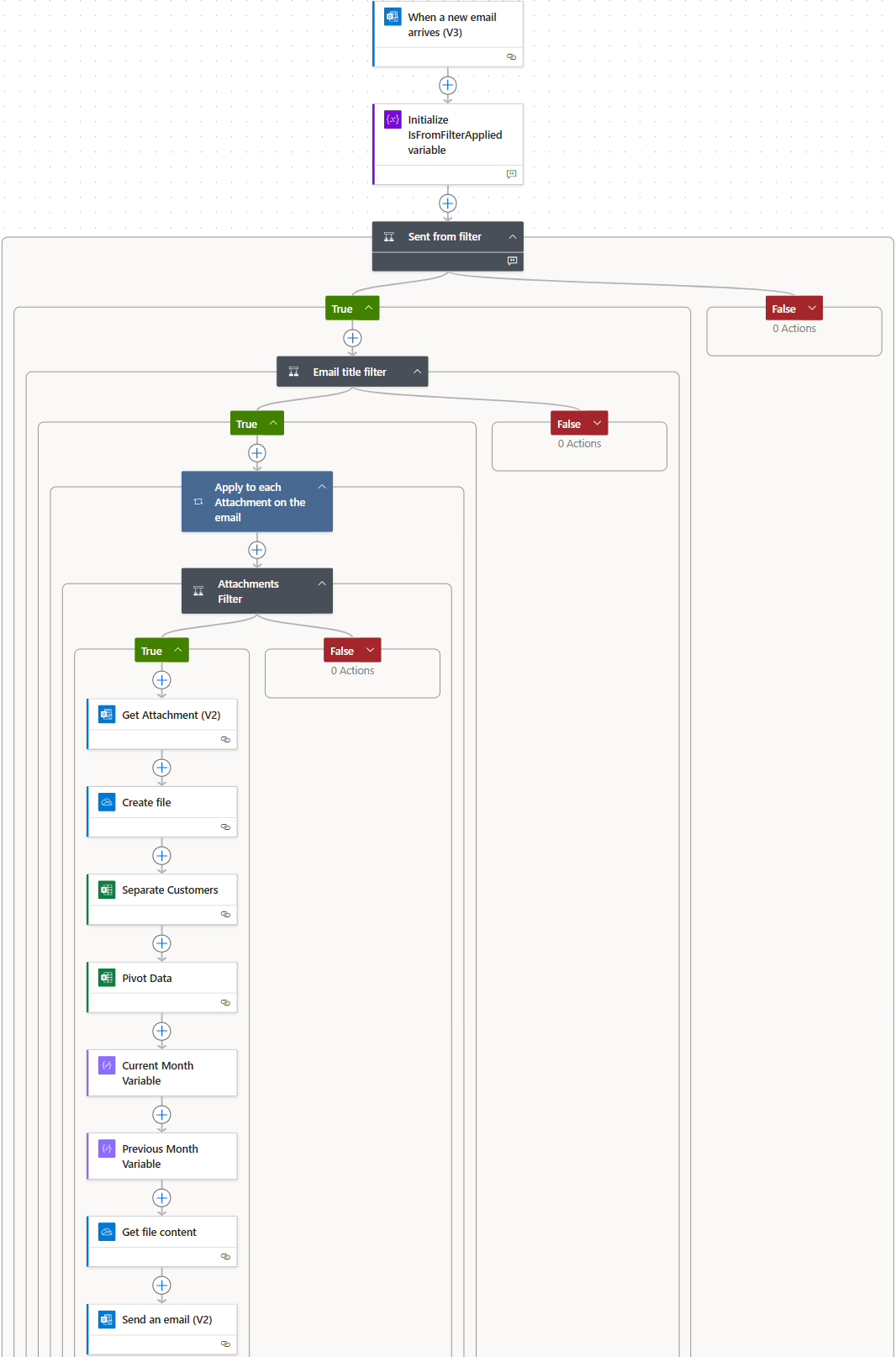
### 5.2 Power Automate flows

#### 5.2.1 Monthly raws flow:

A screenshot of a computer

AI-generated content may be incorrect.

#### 5.2.2 Suites flow:



### 5.3 Office Scripts

#### 5.3.1 Raw data splitting programme

// Function to extract and separate customers by worksheet

function main(workbook: ExcelScript.Workbook): void {

    // Extract range from 1st sheet

    let inputData = tableToString(workbook);

    //console.log(inputData);

    // Convert range to nested objects

    const nestedObjects = stringToObjects(inputData);

    //console.log(JSON.stringify(outputData));

    // Find unique customers as a string, and split them into an array

    let customers = uniqueCustomers(nestedObjects);

    let splitCustomers: string[] = customers.split(', ');

    //console.log(splitCustomers);

    // Keys for nested arrays

    let keys = Object.keys(nestedObjects[0]);

    //console.log(keys);

    // Loop by unique customers...

    for (let customer of splitCustomers) {

        // Create temporary nested array for new sheets

        let tempArray: string[][] = [];

        tempArray.push(keys);

        // Populate nested array

        for (let a = 0; a < nestedObjects.length; a++) {

            //console.log(outputData[a]['Customer ID']);

            if (nestedObjects[a]['Customer ID'] === customer) {

                let values = Object.values(nestedObjects[a]);

                tempArray.push(values);

            }

        }

        //console.log(tempArray);

        // If there's data, create new sheet

        if (tempArray.length > 0) {

            let lastRow = tempArray.length;

            let lastColumn = columnToLetter(tempArray[0].length);

            //console.log(`A1:${lastColumn}${lastRow}`);

            let createSheet = workbook.addWorksheet(customer);

            let selectSheet = workbook.getWorksheet(customer);

            selectSheet.getRange(`A1:${lastColumn}${lastRow}`).setValues(tempArray);

            // Set format for range E:F on selectedSheet

            selectSheet.getRange('E:F').setNumberFormatLocal('[$-en-GB]dd mmm yyyy  hh:mm');

        } else {

            continue;

        }

    }

    workbook.getWorksheets()[0].delete();

}

// Function to convert any number to equivalent alphabetical letter

function columnToLetter(column: number) {

    var temp: number, letter = '';

    while (column > 0) {

        temp = (column - 1) % 26;

        letter = String.fromCharCode(temp + 65) + letter;

        column = (column - temp - 1) / 26;

    }

    return letter;

}

// Function to separate customers from nested objects

function filterCustomers(inputObjects: string[]): string {

    // Finding unique dates's from inputObjects

    const uniqueObjs: string[] = inputObjects.reduce((newArr, element) => {

        // Before element: x-1, and empty array

        //console.log(newArr);

        // Current element: x

        //console.log(element);

        // If the the current object's customer ID is not in newArr, add it

        if (newArr.some(item => item['Customer ID'] === element['Customer ID'])) {

            // Add missing object to array

            newArr.push(element);

            //console.log(element);

        }

        // Return nested objects

        return newArr;

    }, []);

    //console.log(uniqueObjs);

}

// Function to extract unique customers from nested objects

function uniqueCustomers(inputObjects: string[]): string {

    // Finding unique dates's from inputObjects

    const uniqueObjs: string[] = inputObjects.reduce((newArr, element) => {

        // Before element: x-1, and empty array

        //console.log(newArr);

        // Current element: x

        //console.log(element);

        // If the the current object's customer ID is not in newArr, add it

        if (!newArr.some(item => item['Customer ID'] === element['Customer ID'])) {

            // Add missing object to array

            newArr.push(element);

            //console.log(element);

        }

        // Return nested objects

        return newArr;

    }, []);

    //console.log(uniqueObjs);

    // Extracting the parameter values into array

    let uniqueIDs = '';

    for (let index = 0; index < uniqueObjs.length; index++) {

        //console.log(uniqueObjs[index]);

        // If uniqueIDs is blank, no comma

        if (uniqueIDs === '') {

            uniqueIDs = String(uniqueObjs[index]['Customer ID']);

        }

        // Otherwise, add comma

        else {

            uniqueIDs = uniqueIDs.concat(`, ${String(uniqueObjs[index]['Customer ID'])}`);

        }

    }

    //console.log(uniqueIDs);

    return uniqueIDs;

}

// Function to extract table as JSON string

function tableToString(workbook: ExcelScript.Workbook): string[][] {

    // Select 1st sheet in workbook containing all data

    const selectedSheet = workbook.getWorksheets()[0];

    // Get the working range as string

    const rangeText = selectedSheet.getUsedRange().getTexts();

    //console.log(rangeText);

    // Cleaning string

    let length = rangeText.length;

    while (length--) {

        // Remove blank rows

        if (rangeText[length][0] === '') {

            rangeText.splice(length, 1);

            continue;

        }

        // Remove blank columns

        if (rangeText[length][-1] === '') {

            rangeText[length].splice(-1, 1);

            continue;

        }

    }

    //console.log(rangeText);

    return rangeText;

}

// Function to convert a 2D array string to nested objects

function stringToObjects(tableString: string[][]): string[][] {

    // Key: Value pairs

    var objectKeys: string[] = [];

    // Result

    var outputArray: string[][] = [];

    // for each element in array...

    for (var index = 0; index < tableString.length; index++) {

        // Use the 1st element of array as keys

        if (index === 0) {

            objectKeys = tableString[index];

            continue;

        }

        // Empty object to store key and values

        var tempObject: Object = {};

        // For the length of an array within nest...

        for (var element = 0; element < tableString[index].length; element++) {

            //console.log(objectKeys[element]);

            // Set the value of newObject with objectKeys at position (element);

            // Using values of tableString at position (index) at key (element)

            tempObject[objectKeys[element]] = tableString[index][element];

        }

        // Push object into output array

        outputArray.push(tempObject);

        continue;

    }

    return outputArray;

}

#### 5.3.2 Raw data pivoting programme

// Function to extract and clean test results to invoicing standards

function main(workbook: ExcelScript.Workbook): void {

    // Number of worksheets

    let sheets = workbook.getWorksheets().length;

    //console.log(sheets);

    // Loop through sheets...

    for (let sheetID = 0; sheetID < sheets; sheetID++) {

        // Select sheet at position

        let selectedSheet = workbook.getWorksheets()[sheetID];

        // Extract used range from sheet as string

        let rangeText = tableToString(selectedSheet, workbook);

        //console.log(rangeText);

        // Loop through range...

        if (rangeText.length > 1) {

            // Splice of sample ID, parameter, and result value

            const analyses = parameterFilter(rangeText);

            //console.log(analyses);

            // Pivoted parameters

            let transposed = getPivotArray(analyses, 0, 1, 2);

            //console.log(transposed);

            // Keeping a line per unique ID only

            const uniqueIDs = separateIDs(rangeText);

            //console.log(uniqueIDs);

            // Appending pivoted results to uniqueIDs

            for (let a = 0; a < uniqueIDs.length; a++) {

                //console.log(uniqueIDs[a]);

                // Remove old analysis and results column

                uniqueIDs[a].splice(7, 2);

                //console.log(uniqueIDs[a][1]);

                //console.log(transposed[a][0]);

                // If record IDs match in both lists...

                if (uniqueIDs[a][1] === transposed[a][0]) {

                    // Remove Sample ID before appending

                    const results = transposed[a].splice(1, transposed[a].length);

                    //console.log(results);

                    //console.log(uniqueIDs[a]);

                    uniqueIDs[a] = uniqueIDs[a].concat(results);

                }

            }

            //console.log(uniqueIDs);

            // Convert nested array to nested objects

            const newObjects = stringToObjects(uniqueIDs);

            //console.log(newObjects);

            // Output array

            let nestedArray: string[][] = [];

            // Convert nested objects to array

            for (let a = 0; a < newObjects.length; a++) {

                //console.log(newObjects[a]);

                if (a === 0) {

                    //

                    const keys = Object.keys(newObjects[a]);

                    //console.log(keys);

                    nestedArray.push(keys);

                    //

                    const values = Object.values(newObjects[a]);

                    //console.log(values);

                    nestedArray.push(values);

                } else {

                    const values = Object.values(newObjects[a]);

                    //console.log(values);

                    nestedArray.push(values);

                }

            }

            //console.log(nestedArray[0]);

            // Loop across the table...

            if (nestedArray[0].length > 0) {

                // Converting entry length to column letter

                const columnLetters = columnToLetter(nestedArray[0].length);

                // Extract range from sheet

                let usedRange = selectedSheet.getUsedRange();

                usedRange.clear();

                // Select new range to enter data

                const range = selectedSheet.getRange(`A1:${columnLetters}${nestedArray.length}`);

                //console.log(range.getAddress());

                // Set values from nested array

                range.setValues(nestedArray);

                // Convert range to a table

                const table = selectedSheet.addTable(range, true);

                // Format area

                selectedSheet.getRange(`A:${columnLetters}`).getFormat().autofitColumns();

                // Draw table to loop through

                let tableText = selectedSheet.getTables()[0].getRange().getTexts();

                //console.log(tableText);

                // Loop by rows...

                for (let b = 0; b < tableText.length; b++) {

                    //console.log(tableText[b]);

                    // Variables to select an entire row

                    let endColumn = columnToLetter(tableText[b].length);

                    let lineIndex = b + 1;

                    // Skip headers

                    if (b === 0) {

                        selectedSheet.getRange(`H2:${endColumn}${tableText.length}`).getFormat().setHorizontalAlignment(ExcelScript.HorizontalAlignment.center);

                        continue;

                    }

                    // If sample status isn't authorised or complete, colour the record line

                    if (tableText[b][2] !== 'V' && tableText[b][2] !== 'A') {

                        selectedSheet.getRange(`A${lineIndex}:${endColumn}${lineIndex}`).getFormat().getFill().setColor('#FF9393');

                    }

                    // If parameters were built in lieu of a suite, colour the record line

                    if (tableText[b][6] === 'RE') {

                        selectedSheet.getRange(`A${lineIndex}:${endColumn}${lineIndex}`).getFormat().getFill().setColor('#FFC000');

                    }

                    // Tally up the results

                    for (let c = 7; c < tableText[b].length; c++) {

                        // Skip null values

                        if (tableText[b][c] === '') {

                            continue;

                        }

                        // Parameters to select current cell

                        let rowNumber = tableText.length;

                        let column = columnToLetter(c + 1);

                        // Split cell values to identify result status

                        let splitResults = tableText[b][c].split('; ');

                        //console.log(splitResults);

                        // If result status isn't reportable, apply conditional colouring

                        if (splitResults[1] === 'X') {

                            let activeCell = selectedSheet.getCell(b, c);

                            activeCell.getFormat().getFill().setColor('#ff0000');

                            continue;

                        } else if (splitResults[1] === 'U') {

                            let activeCell = selectedSheet.getCell(b, c);

                            activeCell.getFormat().getFill().setColor('#ff0000');

                            continue;

                        } else if (splitResults[1] === 'E') {

                            let activeCell = selectedSheet.getCell(b, c);

                            activeCell.getFormat().getFill().setColor('#ff0000');

                            continue;

                        } else if (splitResults[1] === 'R') {

                            let activeCell = selectedSheet.getCell(b, c);

                            activeCell.getFormat().getFill().setColor('#ff0000');

                            continue;

                        }

                        // Enter counta formula

                        let formulaCell = selectedSheet.getCell(rowNumber, c);

                        let formulaString = `=COUNTA(${column}2:${column}${tableText.length})`;

                        formulaCell.setValue(formulaString);

                    }

                }

            }

        }

    }

}

// Function to convert a number to an alphabetical letter

function columnToLetter(column: number) {

    // Variables

    var temp: number, letter = '';

    // Loop through all letters in alphabetical order

    while (column > 0) {

        temp = (column - 1) % 26;

        letter = String.fromCharCode(temp + 65) + letter;

        column = (column - temp - 1) / 26;

    }

    return letter;

}

// Function to add unique record dates to array

function separateIDs(inputObjects: string[][]): string[][] {

    // Finding unique dates's from inputObjects

    const uniqueObjs = inputObjects.reduce((newArr, element) => {

        // Before element: x-1, and empty array

        //console.log(newArr);

        // Current element: x

        //console.log(element);

        // If the the current object's year is not in newArr, add it

        if (!newArr.some(item => item[1] === element[1])) {

            // Add missing object to array

            newArr.push(element);

            //console.log(element);

        };

        // Return nested objects

        return newArr;

    }, []);

    //console.log(uniqueObjs);

    return uniqueObjs;

}

// Function to pivot nested array

function getPivotArray(dataArray: string[][], rowIndex: number, colIndex: number, dataIndex: number): string[][] {

    // Loop variables

    var result: Object = {};

    var ret: string[][] = [];

    var newCols: string[] = [];

    // Loop through nested array...

    for (var index = 0; index < dataArray.length; index++) {

        if (!result[dataArray[index][rowIndex]]) {

            result[dataArray[index][rowIndex]] = {};

        }

        result[dataArray[index][rowIndex]][dataArray[index][colIndex]] = dataArray[index][dataIndex];

        //To get column names

        if (newCols.indexOf(dataArray[index][colIndex]) == -1) {

            newCols.push(dataArray[index][colIndex]);

        }

    }

    newCols.sort();

    var item: string[] = [];

    // Add Header Row

    item.push('Sample ID');

    item.push.apply(item, newCols);

    ret.push(item);

    // Add content

    for (var key in result) {

        item = [];

        item.push(key);

        for (var i = 0; i < newCols.length; i++) {

            item.push(result[key][newCols[i]] || "");

        }

        ret.push(item);

        //console.log(ret);

    }

    return ret;

}

// Function to trim excess columns, allowing the analyses to be transposed

function parameterFilter(nestedArray: string[][]): string[][] {

    // Make a copy of nested array

    let transposing = [...nestedArray];

    //console.log(transposing);

    // Transposing analyses for wide format

    for (let a = 0; a < transposing.length; a++) {

        //console.log(transposing[a]);

        // Array of analysis and parameter pairs

        let analysisName = transposing[a].splice(7, 2);

        //console.log(analysisName);

        //console.log(transposing[a]);

        // String of analysis and parameter pairs

        let analysisString = analysisName.join('; ');

        //console.log(analysisString);

        // Array of result and status pairs

        let resultName = transposing[a].splice(7, 2);

        //console.log(resultName);

        //console.log(transposing[a]);

        // String of result and status pairs

        let resultString = resultName.join('; ');

        //console.log(resultString);

        // Addinding analysisString and resultString back to array

        transposing[a].splice(7, 0, analysisString);

        transposing[a].splice(8, 0, resultString);

        //console.log(transposing[a]);

    }

    //console.log(transposing);

    // Remove headers

    transposing.shift();

    // Keep only relevant columns; IDs, tests, and values

    var x = transposing.map((array) => {

        return array.slice(1, 2).concat(array.slice(7, 9));

    });

    //console.log(x);

    return x;

}

// Function to extract table as JSON string

function tableToString(selectedSheet: ExcelScript.Worksheet): string[][] {

    const rangeText = selectedSheet.getUsedRange().getTexts();

    //console.log(rangeText);

    // Cleaning string

    let length = rangeText.length;

    while (length--) {

        // Remove blank rows

        if (rangeText[length][0] === '') {

            rangeText.splice(length, 1);

            continue;

        }

        // Remove blank columns

        if (rangeText[length][-1] === '') {

            rangeText[length].splice(-1, 1);

            continue;

        }

        // Remove null result/ cancelled values

        if (rangeText[length][9] === '' && rangeText[length][10] === 'X') {

            rangeText.splice(length, 1);

            continue;

        }

    }

    //console.log(rangeText);

    return rangeText;

}

// Function to convert a 2D array string to nested objects

function stringToObjects(tableString: string[][]): string[][] {

    // Key: Value pairs

    var objectKeys: string[] = [];

    // Result

    var outputArray: string[][] = [];

    // for each element in array...

    for (var index = 0; index < tableString.length; index++) {

        // Use the 1st element of array as keys

        if (index === 0) {

            objectKeys = tableString[index];

            continue;

        }

        // Empty object to store key and values

        var tempObject: Object = {};

        // For the length of an array within nest...

        for (var element = 0; element < tableString[index].length; element++) {

            //console.log(objectKeys[element]);

            // Set the value of newObject with objectKeys at position (element);

            // Using values of tableString at position (index) at key (element)

            tempObject[objectKeys[element]] = tableString[index][element];

        }

        // Push object into output array

        outputArray.push(tempObject);

        continue;

    }

    return outputArray;

}

#### 5.3.3 Extracting quote document programme

// Function to extract costs and output as nested objects

function main(workbook: ExcelScript.Workbook): string {

    // Select 1st sheet in workbook

    const selectedSheet = workbook.getWorksheets()[0];

    // Get the working range as string

    const usedRange = selectedSheet.getUsedRange();

    let rangeText = usedRange.getTexts();

    //console.log(rangeText);

    // Loop conditions

    let length2 = rangeText.length;

    let analyses: string[][] = [];

    // Loop to capture all analytical rates

    for (let position = 5; position < length2; position++) {

        //console.log(rangeText[b]);

        // Stops at the first blank line past headers

        if (position > 8 && rangeText[position][0] === '' && rangeText[position][1] === '' && rangeText[position][3] === '' && rangeText[position][6] === '') {

            break;

        }

        // If parameter and cost columns are not blank, add it to analysis list

        if (rangeText[position][1] !== '' && rangeText[position][6] !== '') {

            analyses.push(rangeText[position]);

            continue;

        }

        // If parameter and limits columns are not blank, add it to the analysis list

        if (rangeText[position][1] !== '' && rangeText[position][3] !== '') {

            analyses.push(rangeText[position]);

            continue;

        }

        // If suite and limits columns are not blank, add it to the analysis list

        if (rangeText[position][0] !== '' && rangeText[position][3] !== '') {

            analyses.push(rangeText[position]);

            continue;

        }

    }

    //console.log(analyses);

    // Remove unnecessary properties

    for (let a = 0; a < analyses.length; a++) {

        //analyses[a].splice(0, 1);

        analyses[a].splice(-1, 1);

        analyses[a].splice(2, 4);

        //console.log(analyses[a]);

        // If suite is blank, copy previous suite name

        if (analyses[a][0] === '') {

            analyses[a][0] = analyses[a - 1][0];

        }

        // To distinguish: KONE (Nitrogen Suite)

        if (a > 1 && a + 2 < analyses.length && analyses[a - 1][0] === analyses[a + 1][0]) {

            // Append (Nitrogen Suite) to previous value

            analyses[a + 1][0] = analyses[a + 1][0].concat(` ${analyses[a + 2][0]}`);

            // Delete (Nitrogen Suite) so 'KONE (Nitrogen Suite)' is copied down

            analyses[a + 2][0] = '';

        }

        // Remove currency sign from rates for simple addition

        if (analyses[a][2].includes('£') === true) {

            analyses[a][2] = analyses[a][2].replace('£', '');

        }

        // If no parameter is given, use suite name

        if (analyses[a][1] === '') {

            analyses[a][1] = analyses[a][0];

        }

    }

    // Grouping analyses into suites for bulk pricing

    let analysesLength = analyses.length;

    // Loop through costs data...

    while (analysesLength--) {

        // If cost is blank, append to the following parameter

        if (analyses[analysesLength][2] === '') {

            analyses[analysesLength - 1][1] = analyses[analysesLength - 1][1].concat(`, ${analyses[analysesLength][1]}`);

            // Remove final line

            analyses.splice(analysesLength, 1);

        }

    }

    //console.log(analyses);

    // Convert range to nested objects

    let outputData = stringToObjects(analyses);

    console.log(JSON.stringify(outputData));

    return JSON.stringify(outputData);

}

// Function to convert a 2D array string to nested objects

function stringToObjects(tableString: string[][]): string[][] {

    // Key: Value pairs

    var objectKeys: string[] = [];

    // Result

    var outputArray: string[][] = [];

    // for each element in array...

    for (var index = 0; index < tableString.length; index++) {

        // Use the 1st element of array as keys

        if (index === 0) {

            objectKeys = tableString[index];

            continue;

        }

        // Empty object to store key and values

        var tempObject: Object = {};

        // For the length of an array within nest...

        for (var element = 0; element < tableString[index].length; element++) {

            //console.log(objectKeys[element]);

            // Set the value of newObject with objectKeys at position (element);

            // Using values of tableString at position (index) at key (element)

            tempObject[objectKeys[element]] = tableString[index][element];

        }

        // Push object into output array

        outputArray.push(tempObject);

        continue;

    }

    return outputArray;

}

#### 5.3.4 Transforming suites sheet with costs and missed parameters

// Function to extract data and output nested objects

function main(workbook: ExcelScript.Workbook, costs: string): void {

    // Costs table

    let costsData: string[][] = JSON.parse(costs);

    //console.log(costsData);

    // Number of worksheets

    let sheets = workbook.getWorksheets().length;

    //console.log(sheets);

    // Loop through sheets...

    for (let sheetID = 0; sheetID < sheets; sheetID++) {

        // Final array to enter on page

        let outputArray: string[][] = [];

        // Suite cost headers

        let priceTitles: string[] = [];

        priceTitles.push('Unit Price (£)');

        priceTitles.push('Total Cost (£)');

        priceTitles.push('Exceptions');

        outputArray.push(priceTitles);

        // Select 1st sheet in workbook

        const selectedSheet = workbook.getWorksheets()[sheetID];

        // Parameters in suites table

        let parametersText = parametersToString(selectedSheet);

        //console.log(parametersText);

        // Count of suites table

        let countsText = countsToString(selectedSheet);

        //console.log(countsText);

        // Loop through through count of suites...

        for (let a = 1; a < countsText.length; a++) {

            //console.log(countsText[a][1]);

            // Not accounted for parameters catch

            let notAdded: string[] = [];

            // Which suites have been billed once already

            let suiteParameters: string[] = [];

            // Loop through suite parameters...

            for (let b = 1; b < parametersText.length; b++) {

                // If current suite from counts and parameters table matched...

                if (countsText[a][1] === parametersText[b][1]) {

                    // Append suite name and parameter, insert into array

                    let tempString = parametersText[b][2].concat('; ' + parametersText[b][3]);

                    suiteParameters.push(tempString);

                }

            }

            //console.log(suiteParameters);

            // Temp cost of analysis

            let suiteCost = 0;

            // Catch to account for suites once

            let multipleAnalyses: string[] = [];

            // Loop through parameters array...

            for (let analysis of suiteParameters) {

                //console.log(analysis);

                //console.log(multipleAnalyses);

                let splitAnalysis = analysis.split('; ');

                //console.log(splitAnalysis);

                // Boolean to skip completed suites

                let alreadyProcessedSuite = false;

                // Check if current parameter is part of a processed suite

                for (let e = 0; e < multipleAnalyses.length; e++) {

                    if (splitAnalysis[0] === multipleAnalyses[e]) {

                        alreadyProcessedSuite = true;

                    }

                }

                // Skip if suite has been accounted for already

                if (alreadyProcessedSuite === true) {

                    continue;

                }

                // Check to charge by suite or individual rates

                let suiteCheck = false;

                let unitCheck = false;

                // Loop through costs data

                for (let c = 0; c < costsData.length; c++) {

                    // If suite, i.e. more than 1 item in list

                    let splitSuite: string[] = [];

                    if (costsData[c]['Determinand'].includes(', ') === true) {

                        splitSuite = costsData[c]['Determinand'].split(', ');

                        //console.log(splitSuite);

                    }

                    //console.log(splitSuite.length);

                    // Apply suite cost if multiple parameters

                    if (splitSuite.length > 1) {

                        for (let d = 0; d < splitSuite.length; d++) {

                            if (splitAnalysis[0] === costsData[c]['Suite'] && splitAnalysis[1] === splitSuite[d]) {

                                suiteCheck = true;

                                multipleAnalyses.push(splitAnalysis[0]);

                                //console.log(`Adding ${costsData[c]['Rate']} of ${splitAnalysis[1]}, suite and parameter matched (${splitSuite[d]})`);

                                suiteCost += Number(costsData[c]['Rate']);

                                break;

                            }

                        }

                    }

                    // Otherwise, charge individual rate

                    if (splitAnalysis[0] === costsData[c]['Suite'] && splitAnalysis[1] === costsData[c]['Determinand']) {

                        unitCheck = true;

                        //console.log(`Adding ${costsData[c]['Rate']} of ${splitAnalysis[1]}, suite and parameter matched`);

                        suiteCost += Number(costsData[c]['Rate']);

                        break;

                    }

                    //console.log(costsData.length - c);

                }

                // Append non-chargeables

                if (suiteCheck === false && unitCheck === false) {

                    notAdded.push(analysis);

                }

            }

            //console.log(suiteCost);

            //console.log(notAdded);

            // Array to enter on sheet

            let totalSuiteCost: string[] = [];

            // Rounding unit cost

            suiteCost = Math.round(suiteCost \* 100) / 100;

            // Multiplying unit cost by quantity

            let tempSuiteCost = suiteCost \* Number(countsText[a][2]);

            // Adding suite prices to each row

            totalSuiteCost.push(String(suiteCost));

            totalSuiteCost.push(String(tempSuiteCost));

            totalSuiteCost.push(String(notAdded));

            //console.log(totalSuiteCost);

            outputArray.push(totalSuiteCost);

        }

        // Add 3 columns next to count of suites

        //selectedSheet.getRange('D:F').insert(ExcelScript.InsertShiftDirection.right);

        // Insert processed data

        //selectedSheet.getRange(`D1:F${countsText.length}`).setValues(outputArray);

        //selectedSheet.getRange('D:F').getFormat().autofitColumns();

    }

}

// Function to extract table as JSON string

function countsToString(selectedSheet: ExcelScript.Worksheet): string[][] {

    const rangeText = selectedSheet.getUsedRange().getTexts();

    //console.log(rangeText);

    // Cleaning string

    let length = rangeText.length;

    while (length--) {

        // Remove blank rows

        if (rangeText[length][0] === '') {

            rangeText.splice(length, 1);

            continue;

        }

        // Remove blank columns

        if (rangeText[length][3] === '') {

            rangeText[length].splice(3, rangeText[length].length);

            continue;

        }

    }

    //console.log(rangeText);

    return rangeText;

}

// Function to extract table as JSON string

function parametersToString(selectedSheet: ExcelScript.Worksheet): string[][] {

    const rangeText = selectedSheet.getUsedRange().getTexts();

    //console.log(rangeText);

    // Cleaning string

    let length = rangeText.length;

    while (length--) {

        // Remove blank rows

        if (rangeText[length][4] === '') {

            rangeText.splice(length, 1);

            continue;

        }

        // Remove blank columns

        if (rangeText[length][-1] === '') {

            rangeText[length].splice(-1, 1);

            continue;

        }

        // Remove spaces between tables

        if (rangeText[length][3] === '') {

            rangeText[length].splice(0, 4);

            continue;

        }

    }

    //console.log(rangeText);

    // Amending parameter names to match quotes

    for (let a = 1; a < rangeText.length; a++) {

        //console.log(rangeText[a]);

        if (rangeText[a][3].includes('Total Coliforms') === true) {

            rangeText[a][3] = 'Total coliforms';

        }

        if (rangeText[a][3].includes('E coli') === true) {

            rangeText[a][2] = 'COLILERT';

        }

        if (rangeText[a][3].includes('Total coliforms') === true) {

            rangeText[a][2] = 'COLILERT';

        }

        if (rangeText[a][2].includes('BAC\_CAPS') === true && rangeText[a][3].includes('3 day plate count 22C') === true) {

            rangeText[a][2] = 'TVC';

            rangeText[a][3] = '3 day plate count @22c';

        }

        if (rangeText[a][2].includes('EDEN\_EXP') === true && rangeText[a][3].includes('3 day plate count plate') === true) {

            rangeText[a][2] = 'TVC';

            rangeText[a][3] = '3 day plate count @22c';

        }

        if (rangeText[a][2].includes('COLIF\_RINS') === true && rangeText[a][3].includes('3 day plate count 22C') === true) {

            rangeText[a][2] = 'TVC';

            rangeText[a][3] = '3 day plate count @22c';

        }

        if (rangeText[a][2].includes('BAC\_250') === true) {

            rangeText[a][2] = 'COLILERT';

        }

        if (rangeText[a][2].includes('STD\_BAC') === true) {

            rangeText[a][2] = 'COLILERT';

        }

        if (rangeText[a][2].includes('ENTCLO\_250') === true) {

            rangeText[a][2] = 'ENT\_CLOS';

        }

        if (rangeText[a][2].includes('CRYPTO\_BUL') === true) {

            rangeText[a][2] = 'CRYPTO';

        }

        if (rangeText[a][3].includes('Oocysts') === true) {

            rangeText[a][3] = 'Crypto Bulk Bag ';

        }

        if (rangeText[a][2].includes('EDEN\_EXP') === true && rangeText[a][3].includes('Pseudomonas aeruginosa') === true) {

            rangeText[a][2] = 'PSEUD';

        }

        if (rangeText[a][3].includes('Pseudomonas aeruginosa') === true) {

            rangeText[a][3] = 'Pseudomonas Aeruginosa';

        }

        if (rangeText[a][2].includes('PS\_AER\_250') === true) {

            rangeText[a][2] = 'PSEUD';

        }

        if (rangeText[a][3].includes('7 day plate count 22C') === true) {

            rangeText[a][3] = '7 day plate count @ 37c';

        }

        if (rangeText[a][2].includes('TVC\_100\_SC') === true) {

            rangeText[a][2] = 'TVC\_100ii';

        }

        if (rangeText[a][2].includes('TVC\_REN\_SC') === true) {

            rangeText[a][2] = 'TVC\_RENALii';

        }

        if (rangeText[a][3].includes('2 day plate count 22C') === true) {

            rangeText[a][3] = '2 day plate count @ 37c';

        }

        if (rangeText[a][2].includes('TVC\_CFPPSC') === true) {

            rangeText[a][2] = 'TVC\_CFPPii';

        }

        if (rangeText[a][2].includes('MYCOBAC') === true) {

            rangeText[a][2] = 'MYCOBACii';

            rangeText[a][3] = 'Mycobacteria';

        }

        if (rangeText[a][2].includes('SC\_ENDOTOX') === true) {

            rangeText[a][2] = 'ENDOTOXii';

        }

        if (rangeText[a][3].includes('Nitrate as NO3') === true) {

            rangeText[a][3] = 'Nitrate as NO3 (by calculation)';

        }

        if (rangeText[a][2].includes('CHROMIUMVI') === true) {

            rangeText[a][2] = 'CHR6';

        }

        if (rangeText[a][2].includes('MISC\_PHYS') === true) {

            rangeText[a][2] = 'MISC PHYS i';

        }

        if (rangeText[a][2].includes('SUB\_ACRYL') === true) {

            rangeText[a][2] = 'ACRYL i';

        }

        if (rangeText[a][2].includes('ACID\_PEST') === true) {

            rangeText[a][2] = 'ACID Suite';

        }

        if (rangeText[a][2].includes('OCPP') === true) {

            rangeText[a][2] = 'OCP Suite';

        }

        if (rangeText[a][2].includes('ONP\_PEST') === true) {

            rangeText[a][2] = 'ONS Suite';

        }

        if (rangeText[a][2].includes('PAH') === true) {

            rangeText[a][2] = 'PAH Suite';

        }

        if (rangeText[a][2].includes('TRZ\_URON') === true) {

            rangeText[a][2] = 'TRZ\_URON Suite';

        }

        if (rangeText[a][2].includes('VOC') === true) {

            rangeText[a][2] = 'VOC Suite';

        }

        if (rangeText[a][3].includes('Beryllium as Be') === true) {

            rangeText[a][3] = 'Beryllium';

        }

        if (rangeText[a][3].includes('Chromium VI as CrVI') === true) {

            rangeText[a][3] = 'Chromium VI';

        }

        if (rangeText[a][3].includes('Total Dissolved Solids') === true) {

            rangeText[a][3] = 'Dissolved Organic Carbon ii';

        }

        if (rangeText[a][2].includes('EPICHLOR') === true) {

            rangeText[a][2] = 'EPICHLOR ii';

            rangeText[a][3] = 'Epichlorohydrin';

        }

        if (rangeText[a][3].includes('L pneumophila gp 2-14') === true) {

            rangeText[a][3] = 'Legionella';

        }

        if (rangeText[a][3].includes('Soluble Reactive Phosphate as P') === true) {

            rangeText[a][3] = 'Soluble Reactive Phosphate';

        }

        if (rangeText[a][3].includes('Thallium as Tl') === true) {

            rangeText[a][3] = 'Thallium';

        }

        if (rangeText[a][2].includes('VINYL\_CHLO') === true) {

            rangeText[a][2] = 'VINYL\_CHLii';

            rangeText[a][3] = 'Vinyl chloride';

        }

        if (rangeText[a][2].includes('TRITIUM') === true) {

            rangeText[a][2] = 'TRITIUM ii';

        }

        if (rangeText[a][2].includes('TDS\_DRYR') === true) {

            rangeText[a][2] = 'TOC';

        }

        if (rangeText[a][2].includes('A\_B\_RADIO') === true) {

            rangeText[a][2] = 'RADIO ii';

        }

        if (rangeText[a][2].includes('KONE') === true) {

            if (rangeText[a][3].includes('Nitrite as NO2') === true) {

                rangeText[a][2] = 'KONE (Nitrogen Suite)';

            }

            if (rangeText[a][3].includes('Total Oxidised Nitrogen as NO3') === true) {

                rangeText[a][2] = 'KONE (Nitrogen Suite)';

            }

            if (rangeText[a][3].includes('Nitrate as NO3 (by calculation)') === true) {

                rangeText[a][2] = 'KONE (Nitrogen Suite)';

            }

        }

        if (rangeText[a][3].includes('37C') === true) {

            rangeText[a][3] = rangeText[a][3].replace('37C', '@ 37c');

        }

        if (rangeText[a][3].includes('22C') === true) {

            rangeText[a][3] = rangeText[a][3].replace('22C', '@22c');

        }

        if (rangeText[a][3].includes('30C') === true) {

            rangeText[a][3] = rangeText[a][3].replace('30C', '@ 30c');

        }

    }

    //console.log(rangeText);

    return rangeText;

}

// Function to convert a 2D array string to nested objects

function stringToObjects(tableString: string[][]): string[][] {

    // Key: Value pairs

    var objectKeys: string[] = [];

    // Result

    var outputArray: string[][] = [];

    // for each element in array...

    for (var index = 0; index < tableString.length; index++) {

        // Use the 1st element of array as keys

        if (index === 0) {

            objectKeys = tableString[index];

            continue;

        }

        // Empty object to store key and values

        var tempObject: Object = {};

        // For the length of an array within nest...

        for (var element = 0; element < tableString[index].length; element++) {

            //console.log(objectKeys[element]);

            // Set the value of newObject with objectKeys at position (element);

            // Using values of tableString at position (index) at key (element)

            tempObject[objectKeys[element]] = tableString[index][element];

        }

        // Push object into output array

        outputArray.push(tempObject);

        continue;

    }

    return outputArray;

}