Run a Form with a temporary datasource from code (job).

In this article I will explain how we can run a from, which have a temporary datasource assigned. Once the form is running, I will explain how to make a selection in the form grid and export a grid selection into a file (csv). At latest it's show how to archive this file in a dedicated 'Archive' file.

Situation

We have a project, and this project there is a job, which we will run. We also see a class and a form object. The class contains all logic to let the form work properly. The form contains one datasource, of an existing temporary table in Ax (tmpCompanyLookup). In this example we use the standard temporary table TmpCompanyLookup to used in the grid of the form. The information to fill this temp table is coming from table `LogisticsAddressCountryRegion.` In order not to make it too complex for now we used the fields CountryId and name for our temp table.
When users running the class a dialog appear. The values of this dialog is used accross the program.

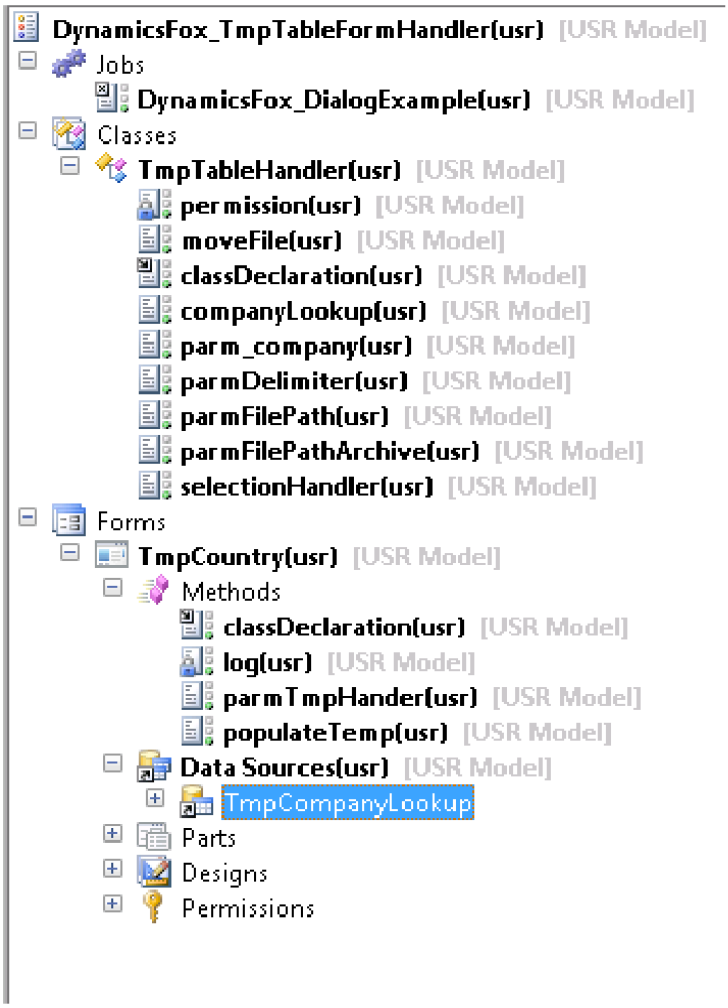
In the code we see that two main processed happened when we run the job:

1. Generate and handling of the Dialog. (2.1.)

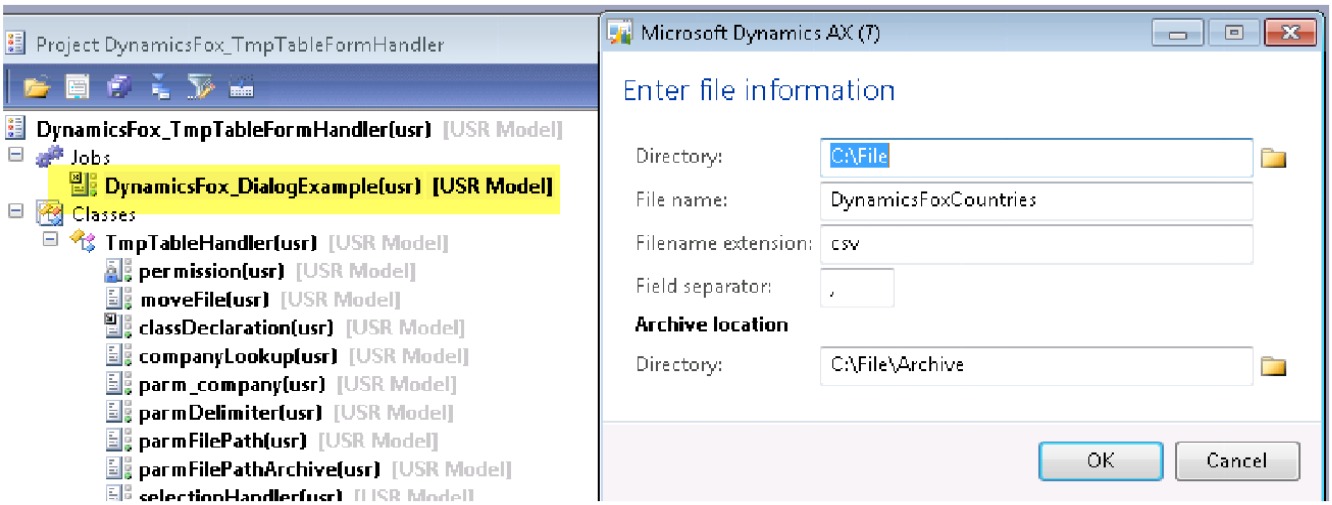
2. Run the form. Once the form is open, also in this form we can do some processes.(2.2.)

```
        selectPath();
        runCountryForm();
    }
```

2.1. SelectPath function, generate and processing Dialog.



If we open the job, we see the following dialog:



Have a close look to the code for this dialog:

We see that the dialog contains 5 dialogfields. In the code we see that the datatype (EDT) of each of the fields is defined in a container with the name 'TypeList'. This makes it easy to modify the dialog , by only adding / removing the dilaogfields datatype in this container.

```
21      container                          countryList,
22                                         dialogValues,
23                                         typeList = [#filePath,
24                                                     #fileName,
25                                                     #fileName,
26                                                     #delimiter,
27                                                     #filePath];
28      Counter                            counter;
```

In this snapshot we see the code how the dialog is actually generated, based on the elements of container 'TypeList':

```
// create dialog
    void selectPath()
    {
        Object          dialogField;
        str             type,
                        objectValue;
        Object          object,
                        dialogPath      = new Dialog("@CON13");

        for(counter = 1; counter <= conlen(typeList); counter++)
        {
            type = conPeek(typeList, counter);

            if (counter == 5)
            {
                dialogPath.addGroup("@CON12");
            }

            if (type)
            {
                dialogField  = dialogPath.addField(type);

                if (counter == 3)
                {
                    dialogField.label("@CON19");
                }

                dialogFields.insert(counter, dialogField);
                getPreviousvalue(counter, dialogField);
            }
        }

        if (dialogPath.run())
        {
            dialogFieldsMap     = new MapEnumerator(dialogFields);

            while (dialogFieldsMap.moveNext())
            {
                object      = dialogFieldsMap.currentValue();
                objectValue = object.value();

                dialogValues = conIns(dialogValues,
                                      dialogFieldsMap.currentKey(),
                                      objectValue);
            }
```

```
        saveValue(dialogValues);
    }

    dialogField         = null;
}
```

Take a look to the following section:

Here we see that dialogfields are created based on the 'Typelist' type.

We see that for dialog fields nr 3 and 5, we not decide to use standard label which belongs to EDT, but make our own preferences with regard to label and group.

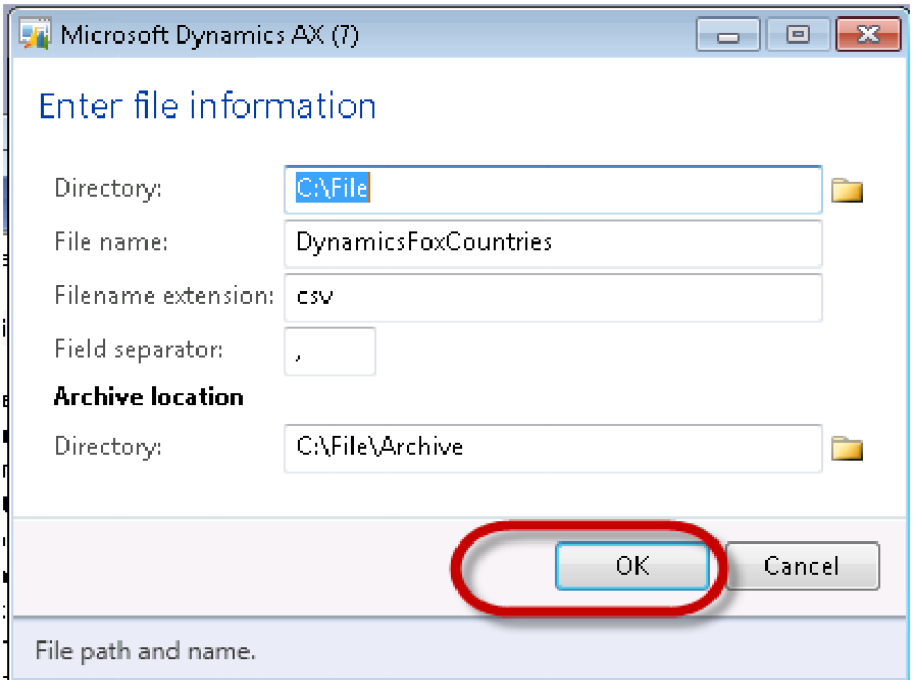
```
105         Object          object,
106                         dialogPath      = new Dialog("@CON13");
107
108         for(counter = 1; counter <= conlen(typeList); counter++)
109         {
110             type = conPeek(typeList, counter);
111
112             if (counter == 5)
113             {
114                 dialogPath.addGroup("@CON12");
115             }
116
117             if (type)
118             {
119                 dialogField  = dialogPath.addField(type);
120
121                 if (counter == 3)
122                 {
123                     dialogField.label("@CON19");
124                 }
125
126                 dialogFields.insert(counter, dialogField);
127                 getPreviousvalue(counter, dialogField);
128             }
129         }
```

There is also a check if there are previous dialog values available. We use the standard class xSysLastValue ('GetValue') to obtain eventually previous values. We also have a code to save ('PutValue') dialog input.

```
118             {
119                 dialogField  = dialogPath.addField(type);
120
121                 if (counter == 3)
122                 {
123                     dialogField.label("@CON19");
124                 }
125
126                 dialogFields.insert(counter, dialogField);
127                 getPreviousvalue(counter, dialogField);
128             }
129         }
130
131         if (dialogPath.run())
132         {
```

```
    //Get the previous dialog values
    void getPreviousValue(counter _id, Object _dialogField)
    {
        dialogValues = conNull();

        dialogValues = xSysLastValue::getValue(curext(), curUserId(),
                UtilElementType::Job, #dialogPath);

        if (dialogValues)
        {
            _dialogField.value(conPeek(dialogValues, _id));
        }
    }

    //Save the dialog values
    void saveValue(container _dialogValues)
    {
        xSysLastValue::putValue(_dialogValues,
                                curext(),
                                curUserId(),
                                UtilElementType::Job,
                                #dialogPath);
    }

```

When we click the 'OK' button on the dialog



The following code is called, and eventually modified dialog values will be assigned to the dialogfields.

```
130         |
131             if (dialogPath.run())
132             {
133                 dialogFieldsMap     = new MapEnumerator(dialogFields);
134
135                 while (dialogFieldsMap.moveNext())
136                 {
137                     object      = dialogFieldsMap.currentValue();
138                     objectValue = object.value();
139
140                     dialogValues = conIns(dialogValues,
141                                           dialogFieldsMap.currentKey(),
142                                           objectValue);
143                 }
144
145                 saveValue(dialogValues);
146             }
147
148             dialogField         = null;
149         }
150
```

Eventually the modified dialog values will be assigned to a variable with the name 'DialogValues', en send to the xSysLastValue class to be saved.

2.2. Run the form

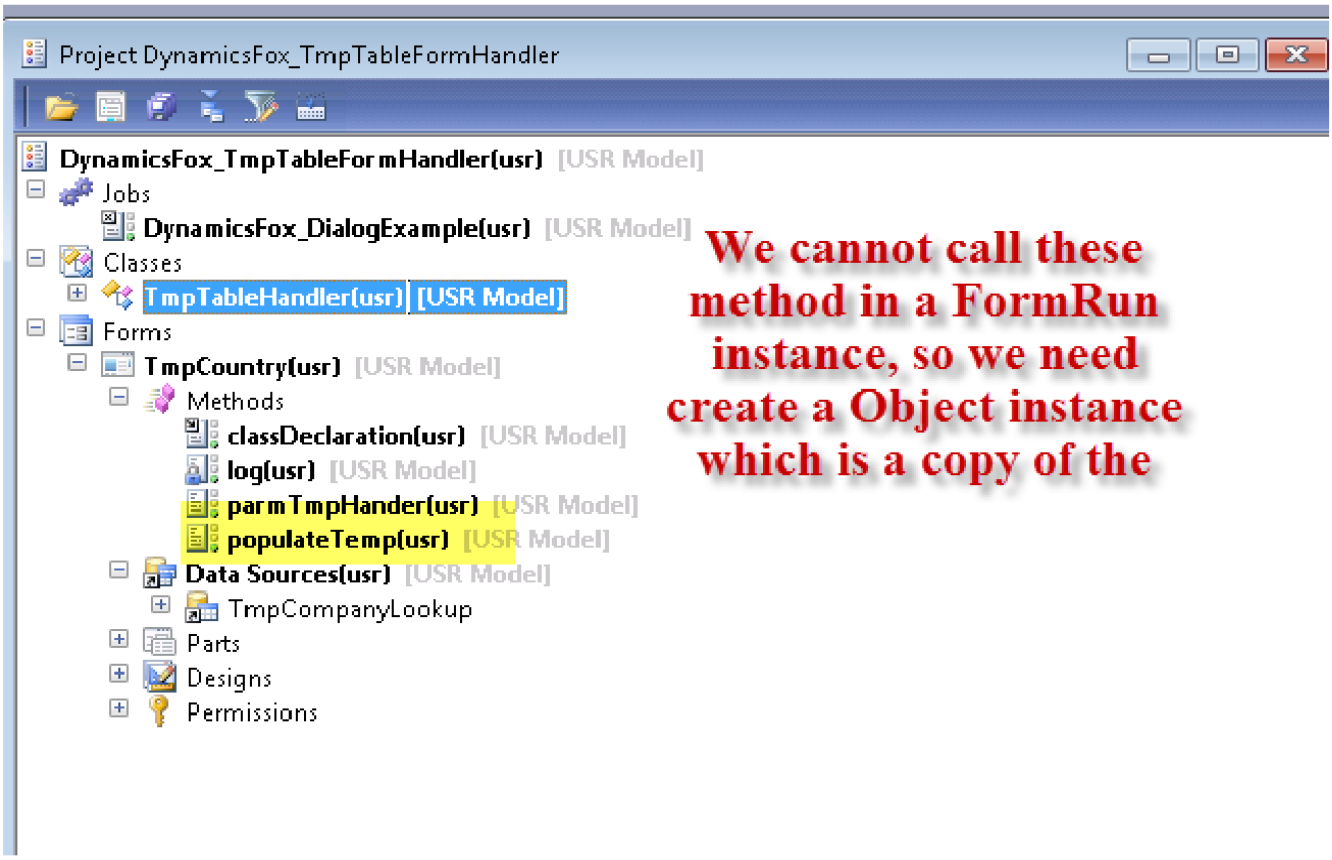
Actually what happened here is that we gather the information from the dialog for processing and operate the form.

Note:

We see that we are using two objects which act as FormRun. FormRun is the runtime class for processing forms, but we can also declare 'Object'. That's done becuase we have added our own methods to the form, which are not recognized by a FormRun instance.

'Object' is used to handle this, but it will not generate an errormessge when we make a spelling mistake with the declaration of the methodname, so it's the developers responsibility to enter correct code.

```
void runCountryForm()
    {
        Formrun                         formRun;
        object                          populateTempTable;
        Args                            args = new Args();
        ;

        filepath = strFmt(@'%1\%2.%3',
                        conPeek(dialogValues,1),
                        conPeek(dialogValues,2),
                        conPeek(dialogValues,3));

        filePathArchive = strFmt(@'%1\%2.%3',
                        conPeek(dialogValues,5),
                        conPeek(dialogValues,2),
                        conPeek(dialogValues,3));

        args.name(formStr(TmpCountry));
        formRun = ClassFactory.formRunClass(args);

        populateTempTable = formRun;

        populateTempTable.parmTmpHander(tmpHandler);

        delimiter = conPeek(dialogValues,4);
        tmpHandler.parmDelimiter(delimiter);
        tmpHandler.parmFilePath(filepath);

        tmpHandler.parmFilePathArchive(filePathArchive);

        formRun.init();
        populateTempTable.populateTemp(countryList);

        formRun.run();
        formRun.wait();
    }
```



And here you can see how we do this (in the job):

```
37      // call the form and fill the temptable
38      void runCountryForm()
39      {
40          Formrun                         formRun;
41          object                          populateTempTable;
42          Args                            args = new Args();
43          ;
44
45          filepath = strFmt(@'%1\%2.%3',
46                            conPeek(dialogValues,1),
47                            conPeek(dialogValues,2),
48                            conPeek(dialogValues,3));
49
50          filePathArchive = strFmt(@'%1\%2.%3',
51                            conPeek(dialogValues,5),
52                            conPeek(dialogValues,2),
53                            conPeek(dialogValues,3));
54
55          args.name(formStr(TmpCountry));
56          formRun = ClassFactory.formRunClass(args);
57
58          populateTempTable = formRun;
59
60          populateTempTable.parmTmpHander(tmpHandler);
61
62          delimiter = conPeek(dialogValues,4);
63          tmpHandler.parmDelimiter(delimiter);
64          tmpHandler.parmFilePath(filepath);
65
66          tmpHandler.parmFilePathArchive(filePathArchive);
67
```

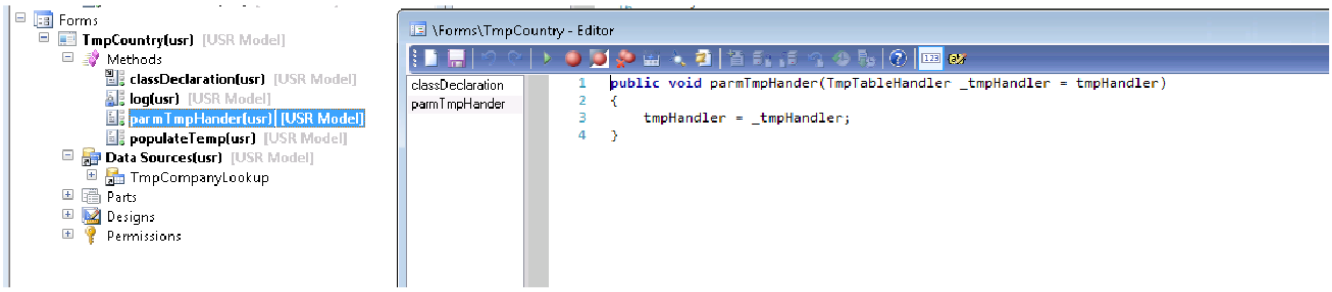
2.2.1 How to insert data in the temporary table?

In this example and I think it's best practise as well, define separate helper class, which is doing as logic for us. Off course it's also possible to build the logic in the form, but there are more benefits related to maintain the code and reusabilty if we use separate classes.

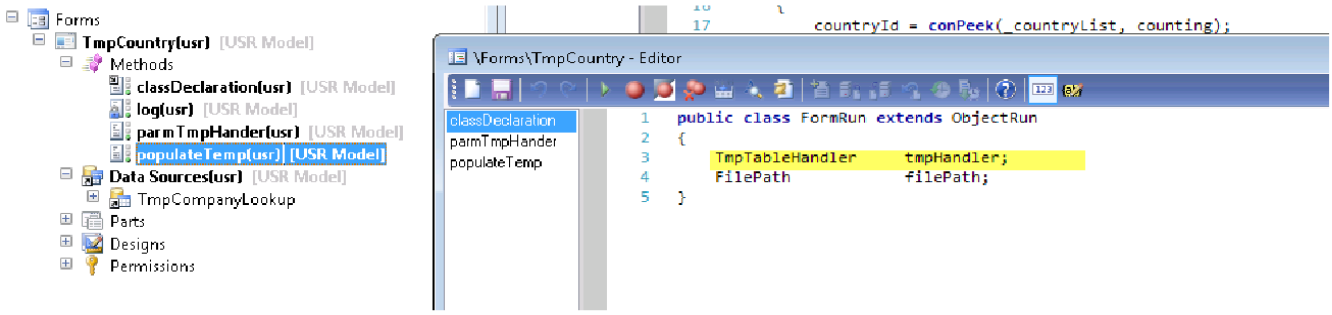
Step 1) Create the class and make a link in the form to this class:

Step 2) Make a parmMethod in Form, not for instiating. I make the choice in this example not to initializing the class again as the class is already instantiated in the job. The reason we are choosing this approach is that we already have assigned the dialogfield values to varables in the class, and thus remain available for further processing. When we instantiate the class here, we will loose this

information.



Step 3) Declare temporary table in the class declaration of the form:



Step 4) Make a query to fill the temporary job in the class:

Watch the queryBuildRange statement. The plan here is that user should makes a selection, once the form is running , and mark records he would like to write in file. This selection will be put in a container and in yellow marked area, we see how we can use the container values to define the query range (this piece of code will be invoked from class once user select records, is not relevant for us now). When we run the form for the first time (from job), the container is empty, that meant that all records will be selected.

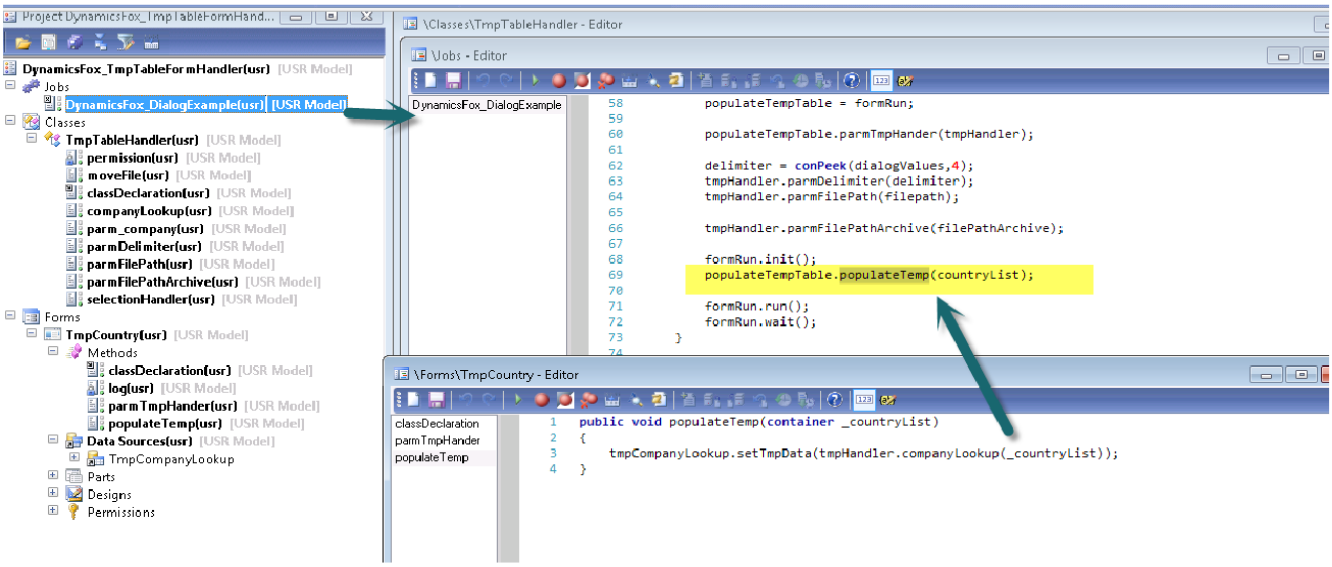
```
TmpCompanyLookup companyLookup(container _countryList = conNull())
{
    Query                          query = new query();
    QueryBuildDataSource           qbds;
    QueryBuildRange                qbr;
    QueryRun                       qr;

    LogisticsAddressCountryRegion  country;
    TmpCompanyLookup               tmpTable;
    Counter                        counting;
    DataAreaId                     countryId;

    qbds = query.addDataSource(tableNum(LogisticsAddressCountryRegion));

    for (counting = 1; counting <= conLen(_countryList); counting++)
    {
        countryId = conPeek(_countryList, counting);
        qbr = qbds.addRange(fieldNum(LogisticsAddressCountryRegion, CountryRegionId));
        qbr.value(SysQuery::value(countryId));
    }

    qr = new QueryRun(query);

    while (qr.next())
    {
        country = qr.get(tableNum(LogisticsAddressCountryRegion));
        tmpTable.DataArea   = country.CountryRegionId;
        tmpTable.Name       = country.displayName();
        tmpTable.insert();
    }

    return tmpTable;
}
```

Step 5) In order to call this query we need to add the following code in the form:

```
public void populateTemp(container _countryList)
{
    tmpCompanyLookup.setTmpData(tmpHandler.companyLookup(_countryList));
}
```

Step 6) Following modification in job:

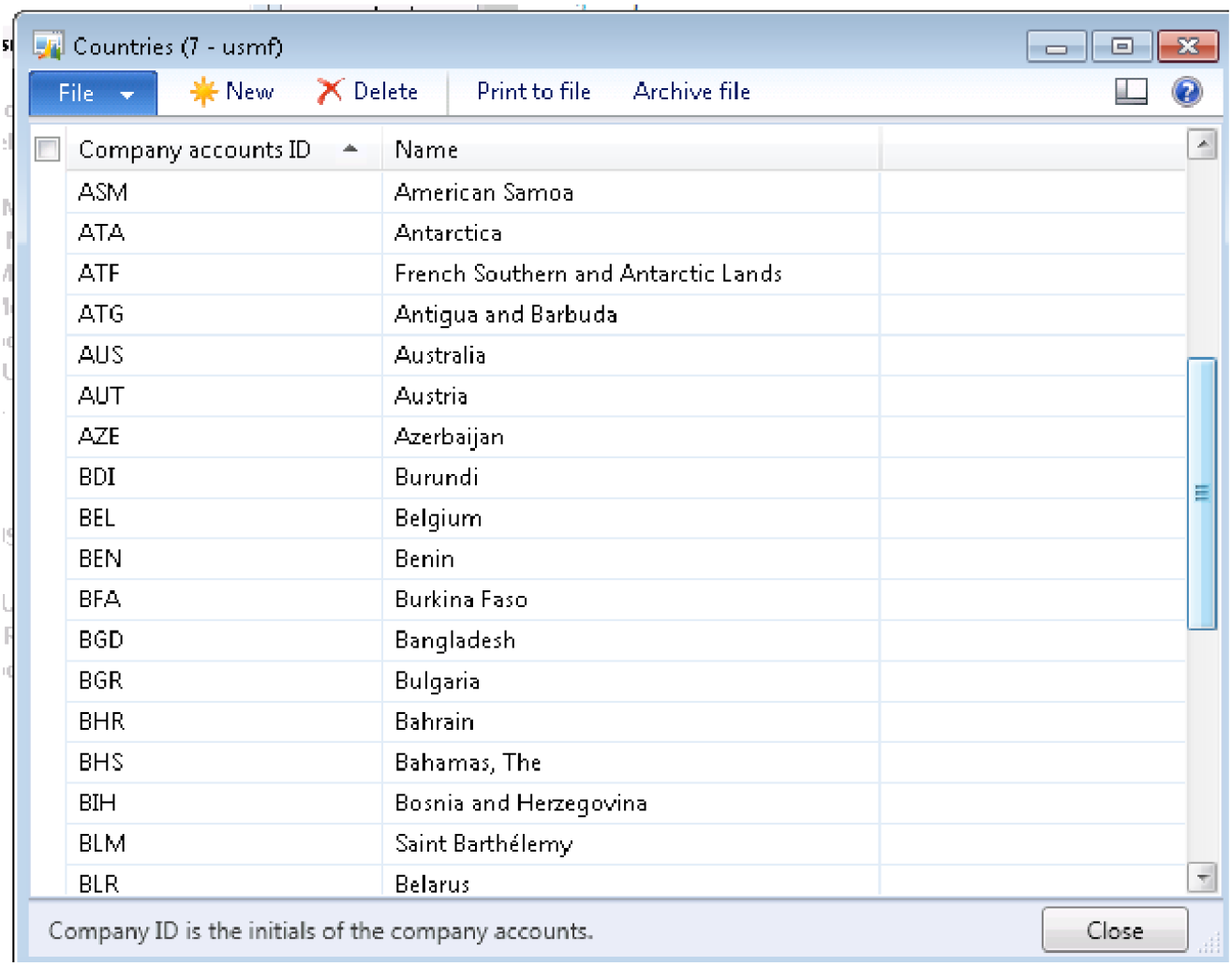
Before we can use a Temp table we need make it ready for use by the 'setTmpData' statement below. As parameter for 'SetTmp' we call a method of same type as the temporary table is. In our example the temporary table is TmpCompanyLookup, so the return type for this method should be the same.



2.2.2. Running the Form

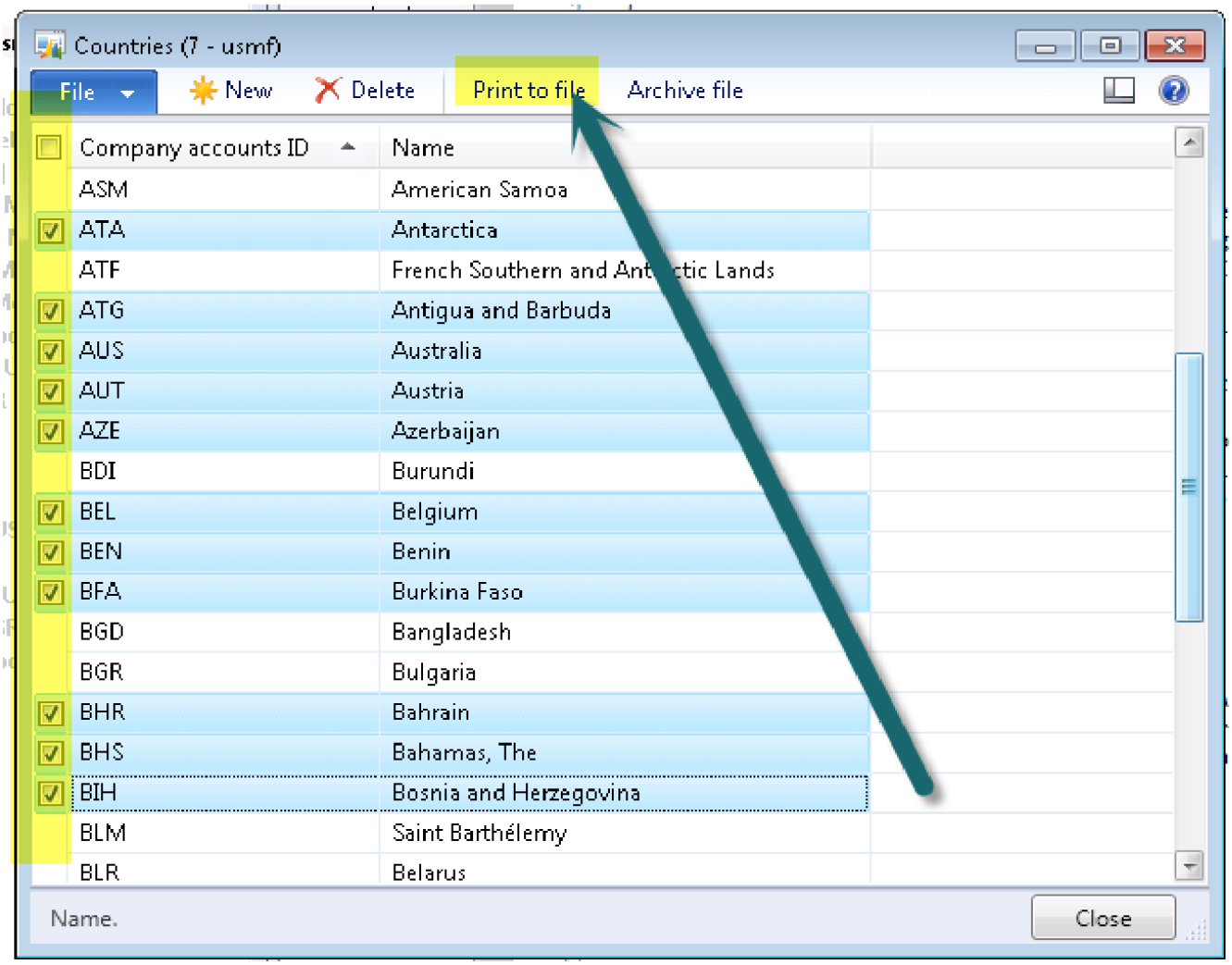
When this steps are done, we can open the form (from the job).

When opened form looks like this:

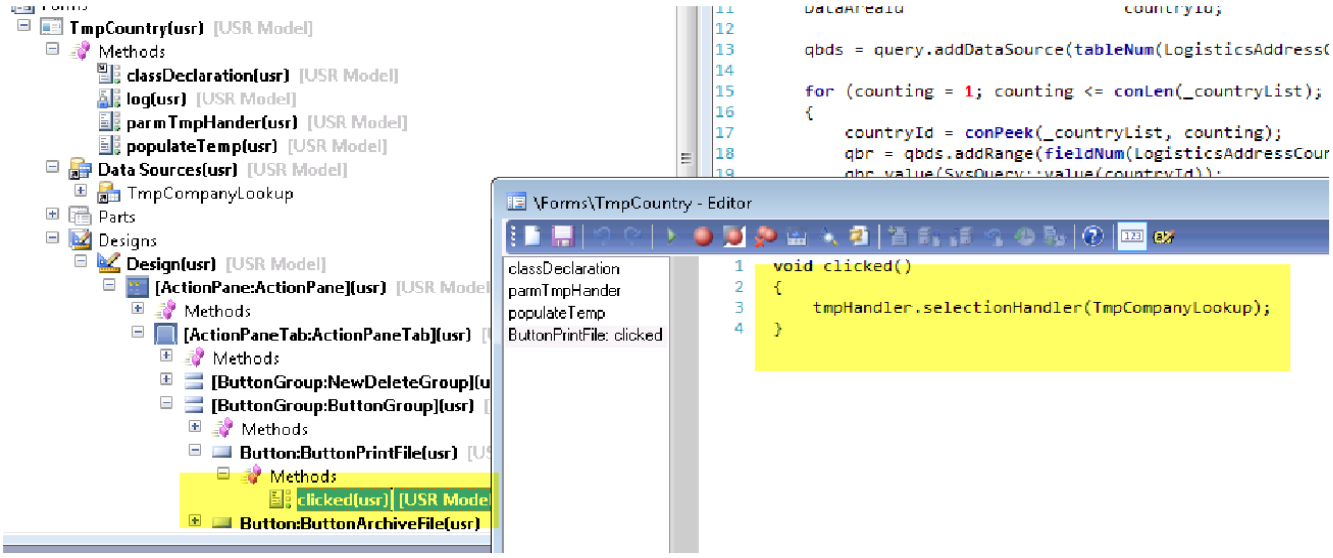


To select records of your choise click mark the checkboxes at the left side.

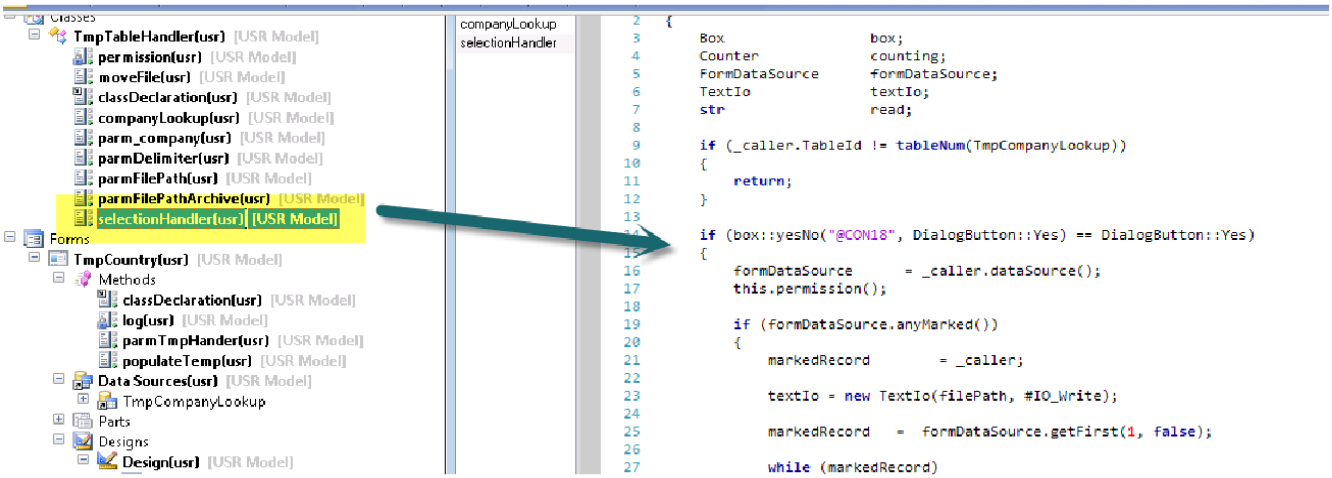
In order to generate a file, based on the configuration we entered in the dialog, we need click the button 'Print to File'.



This is the logic behind that button:



The button is calling the method 'selectionHandler' in the class:

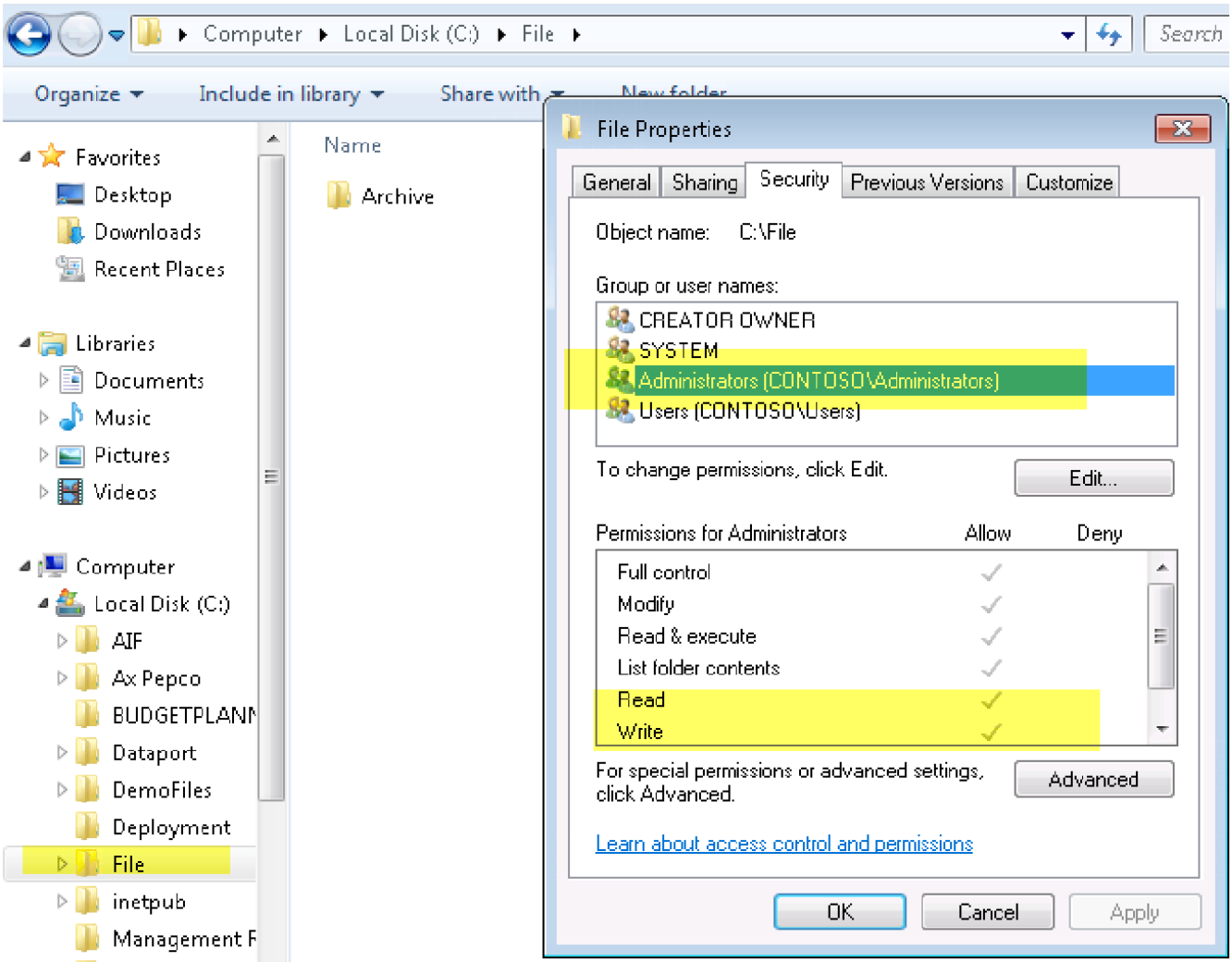


a full print of this class:

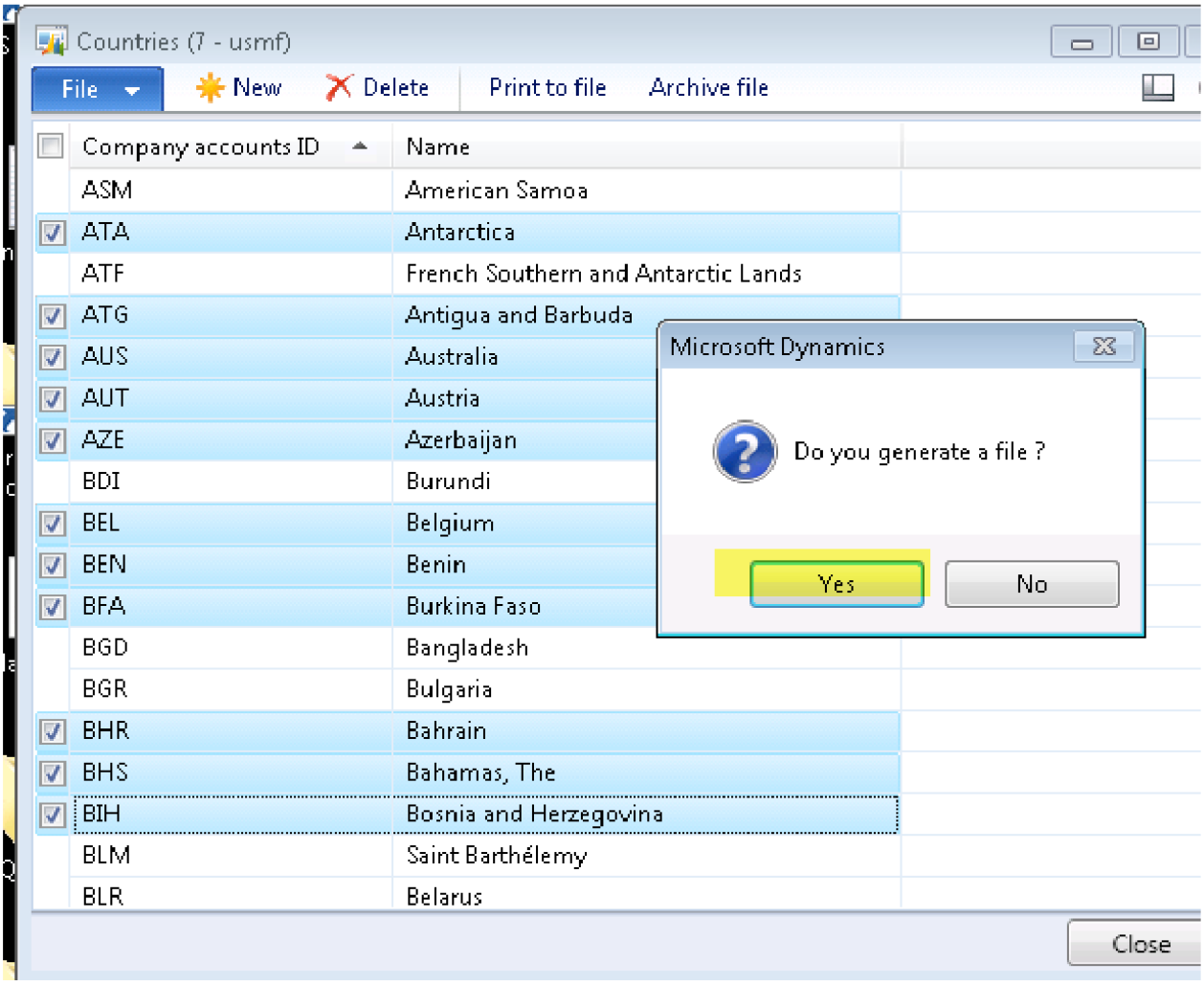
```
public void selectionHandler(Common _caller)
{
    Box                 box;
    Counter             counting;
    FormDataSource      formDataSource;
    TextIo              textIo;
    str                 read;

    if (_caller.TableId != tableNum(TmpCompanyLookup))
    {
        return;
    }

    if (box::yesNo("@CON18", DialogButton::Yes) == DialogButton::Yes)
    {
        formDataSource      = _caller.dataSource();
        this.permission();

        if (formDataSource.anyMarked())
        {
            markedRecord        = _caller;

            textIo = new TextIo(filePath, #IO_Write);

            markedRecord   =  formDataSource.getFirst(1, false);

            while (markedRecord)
            {
                counting++;

                read = markedRecord.(fieldNum(TmpCompanyLookup, DataArea))
                        + delimiter
                        + markedRecord.(fieldNum(TmpCompanyLookup, Name));

                info(read);
                textIo.write(read);


                markedRecord = formDataSource.getNext();
            }

        }

        CodeAccessPermission::revertAssert();
```

```
    }
}
```

Before we can generate a file, we need to be sure that we have proper permissions for the maps we selected in the dialog:

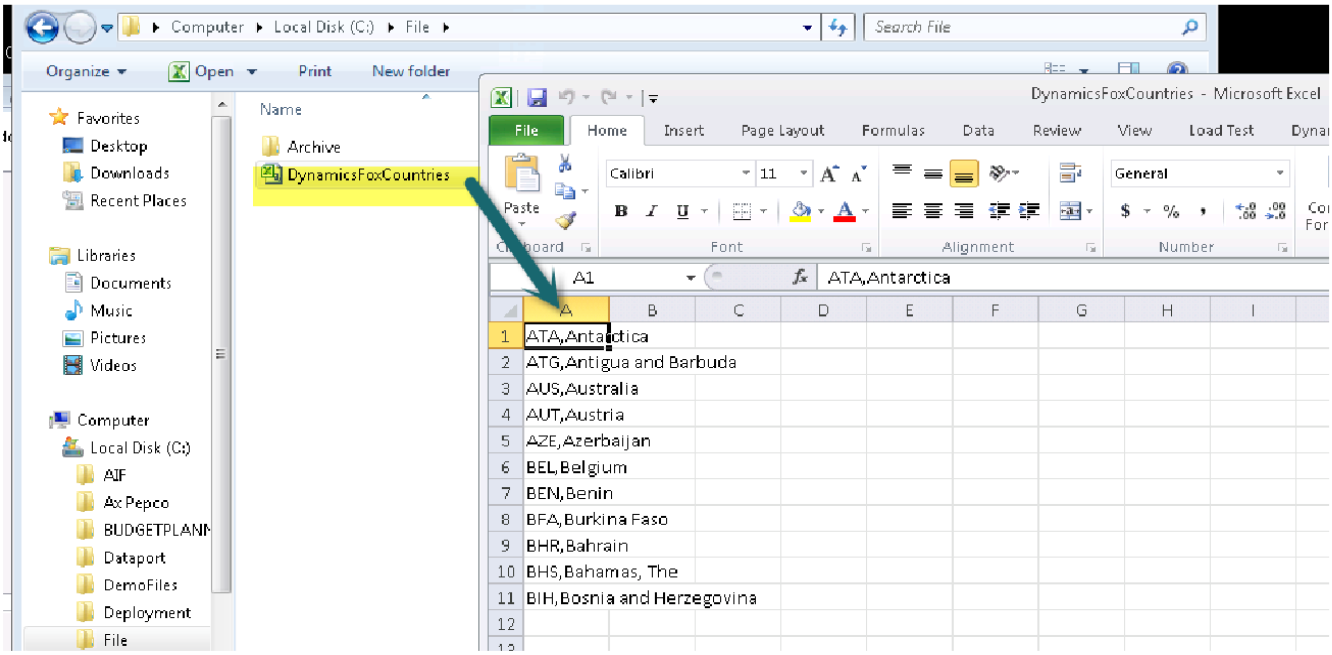


If we make selection and click the button, result looks like this:

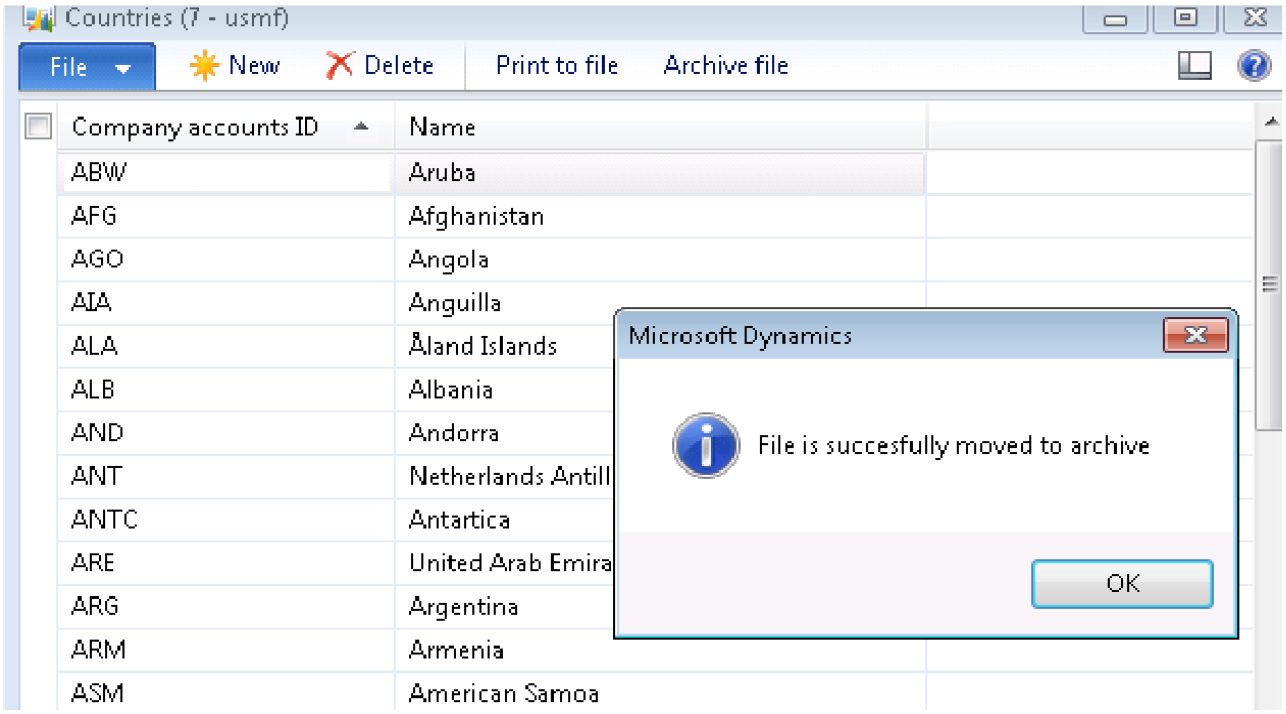


Click 'Yes'

and in the map, we see that the file is generated.



To archive this file click button 'Archive file'. The file will be moved to the what is configured as archive location in the dialog (first close it completely). Also check authorizations on the 'archive' map as defined in the dialog are sufficient (read & write).



Logic behind:

```
classDeclaration
companyLookup
selectionHandler
permission

private void permission()
{

    permissionSet = new Set(Types::Class);
    permissionSet.add(new FileIOPermission(filePath,#io_write));
    permissionSet.add(new InteropPermission(InteropKind::ClrInterop));

    CodeAccessPermission::assertMultiple(permissionSet);
}
```

We use the System.IO.File class.

```
\Classes\TmpTableHandler - Editor
classDeclaration
companyLookup
selectionHandler
permission
moveFile

public void moveFile()
{
    this.permission();

    CodeAccessPermission::assertMultiple(permissionSet);

    System.IO.File::Move(filePath, filePathArchive);
    System.IO.File::Delete(filePath);

    CodeAccessPermission::revertAssert();

    Box::info('@CON22');
}
```

ParmMethod in the class 'TmpTableHandler'

```
\Classes\TmpTableHandler - Editor
classDeclaration
companyLookup
selectionHandler
permission
moveFile
parmFilePathArchive

public void parmFilePathArchive(FilePath _filePath = filePathArchive)
{
    filePathArchive = _filePath;
}
```

Called from job:

DynamicsFox_DialogExample

```
40          Formrun                         formRun;
41          object                          populateTempTable;
42          Args                            args = new Args();
43          ;
44
45          filepath = strFmt(@'%1\%2.%3',
46                          conPeek(dialogValues,1),
47                          conPeek(dialogValues,2),
48                          conPeek(dialogValues,3));
49
50          filePathArchive = strFmt(@'%1\%2.%3',
51                          conPeek(dialogValues,5),
52                          conPeek(dialogValues,2),
53                          conPeek(dialogValues,3));
54
55          args.name(formStr(TmpCountry));
56          formRun = ClassFactory.formRunClass(args);
57
58          populateTempTable = formRun;
59
60          populateTempTable.parmTmpHander(tmpHandler);
61
62          delimiter = conPeek(dialogValues,4);
63          tmpHandler.parmDelimiter(delimiter);
64          tmpHandler.parmFilePath(filepath);
65
66          tmpHandler.parmFilePathArchive(filePathArchive);
67
68          formRun.init();
69          populateTempTable.populateTemp(countryList);
70
71          formRun.run();
72          formRun.wait();
73      }
74
```