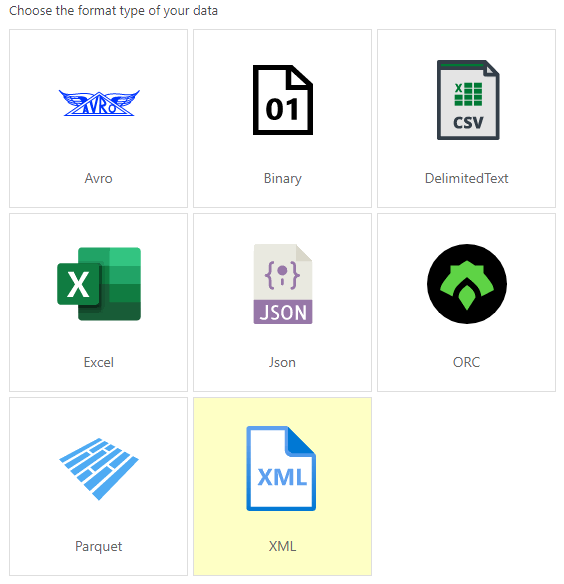
Remember when we diverted a Delimited Text dataset so the Azure Data Factory could read it as XML and dump it into the SQL Server? Fortunately our hack is obsolete by now. The Data Factory now natively supports XML files in Copy Activity and Data Flows. Let’s take a look!

**Simple file, easy process**  
Reading XML files is easy when the file structure is straightforward. Consider the customer file we used : A customer record has a unique number, a mail address and an operation like “insert” or “update”. The XML is a simple list of customers, each nested in a row tag:

<Customers>  
 <row>  
 <CUSTOMERNO>12</CUSTOMERNO>  
 <OPERATION>INSERT</OPERATION>  
 <EMAIL>bill.gates@microsoft.com</EMAIL>  
 </row>  
 <row>  
 <CUSTOMERNO>947</CUSTOMERNO>  
 <OPERATION>INSERT</OPERATION>  
 <EMAIL>steve.jobs@apple.com</EMAIL>  
 </row>  
 ...  
</Customers>

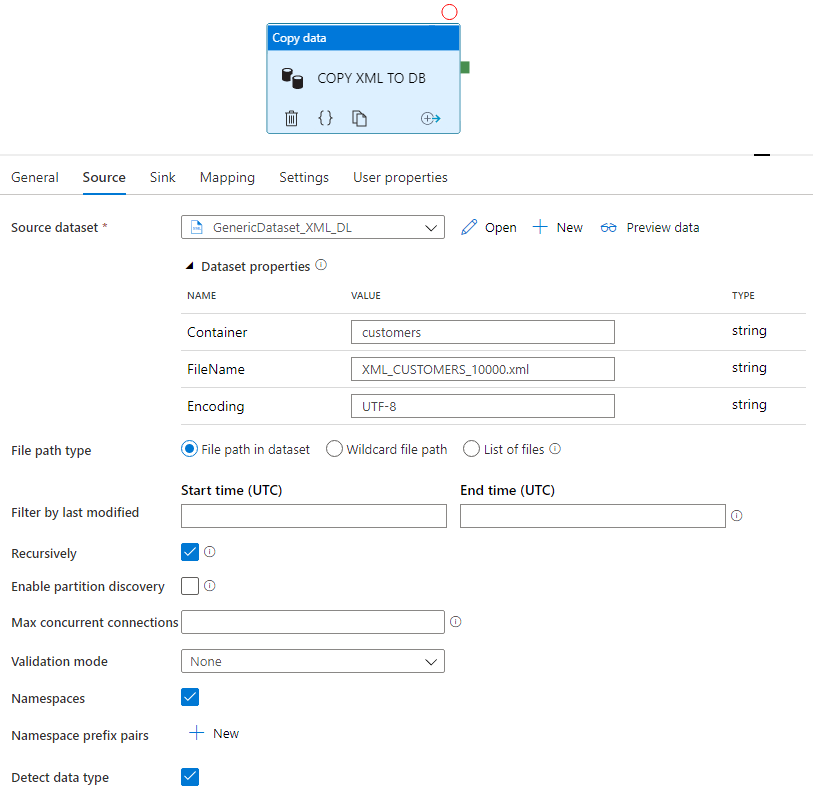
This structure can easily be extracted from an existing SQL Server customer table using the FOR XML PATH clause.

Once uploaded to an Azure Data Lake Storage (v2) the file can be accessed via the Data Factory. First create a new Dataset, choose XML as format type, and point it to the location of the file. Apply further configurations like encoding or compression as needed:



In comparison to last time, Excel and XML are new file types

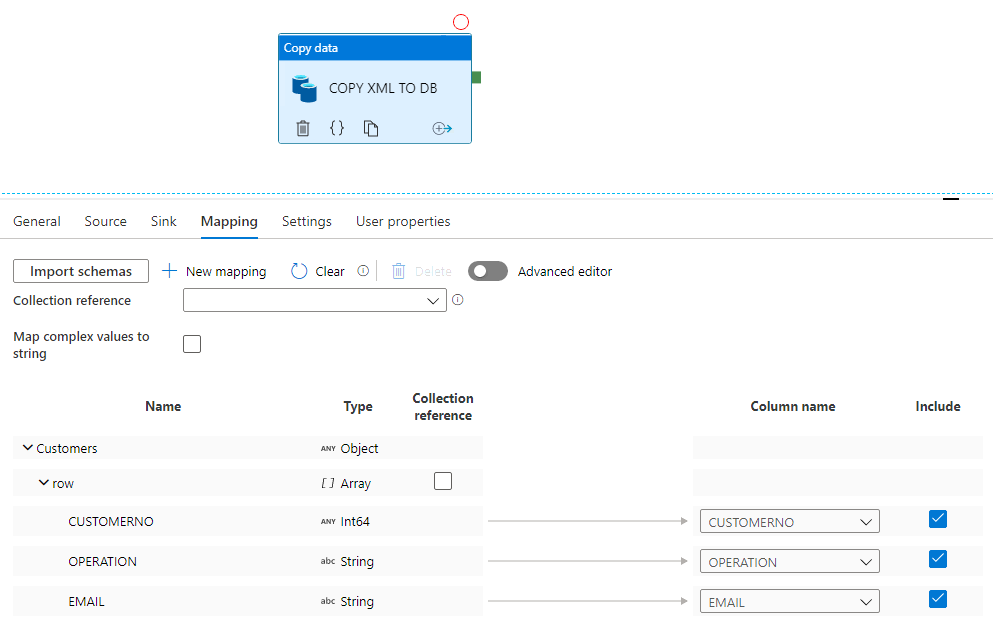
Next create a new pipeline and add a single *Copy Activity*. As source choose the XML dataset created earlier (Note that in the following screenshot we use a Generic XML Dataset in DataLake with FileName, Container, and Encoding properties, just to be able to switch files easily by parameterizing the pipeline). For a simple file like ours you don’t have to configure something else.



Generic XML Dataset with filename property

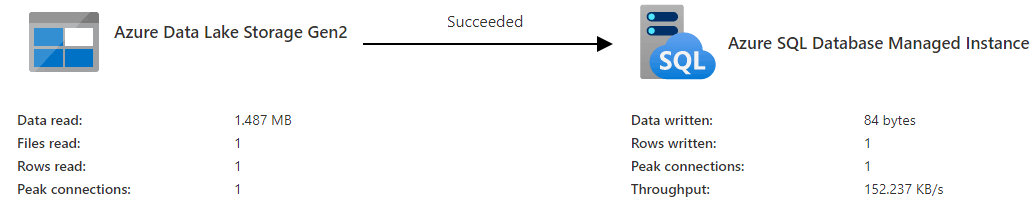
As sink choose the dataset pointing to the target table in SQL Server. In our case we use a table named *dbo.CUSTOMERS*which is structured exactly like a row in the XML file.

The *Mapping* options for the *Copy Activity* are the most relevant part. For a quick shot just hit the “**Import Schemas**” button top left and see the mapping appear automagically. The structure of the XML file is shown as tree, the column names of our target tables are also recognized and a default mapping is created. At first glance this looks quite promising — like in “could work”.



Press “Import schemas” and mapping is done automagically

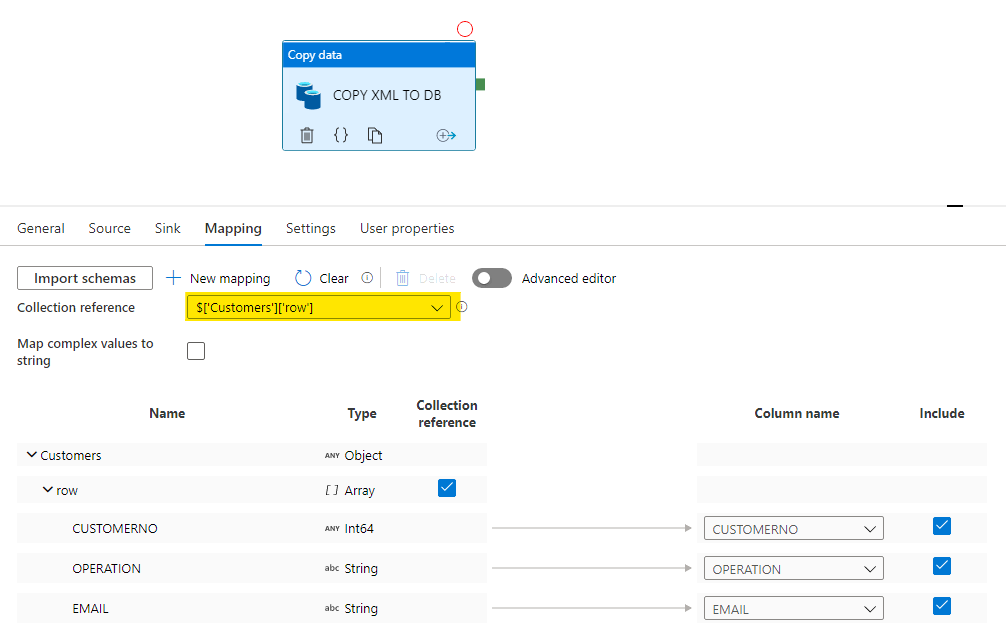
Publish the pipeline, trigger it and see as result: Only the first row is written to the database! That’s wrong, our input file contains 10.000 rows.



We expect 10.000 rows to be written

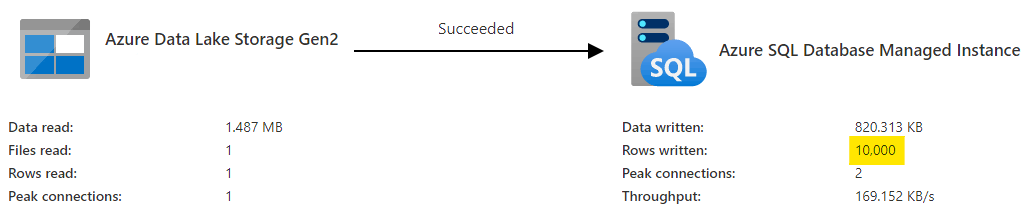
The reason is that we did not specify a **Collection Reference**, which the ADF will loop over to copy data. Easy to spot, in our sample file the collection is the *row* node. Either use the checkbox next to *row*in the mapping or use the select box next to the “Collection reference” label to set the collection reference. The syntax in the Select box might look unfamiliar and refers to a JSON path to walk through the hierarchical structure of the XML up to a nested JSON array.

Noteworthy: when you preview the XML file in the *Source*tab, you will see the XML file represented as JSON. When you are not sure about the correct JSON path in your XML, then a glance at the JSON representation of the data in preview might help.



Set the Collection Reference

Republish, rerun, result: 10.000 rows written!



With Collection Reference all rows are written

Pro tip: you can clean the target table before loading. Just add a pre-copy script in the *Sink*tab. That way the data does not pile up after several pipeline runs and makes the result easier to check:

TRUNCATE TABLE dbo.CUSTOMERS

**It tends to get complex when your file structure does**No problems so far. But XML has a lot to offer, ranging from attributes over namespaces to optional subnodes leading to very different structures in the very same file. Let’s adjust our sample file to reflect some of those features:

<Customers>  
 <row segment="corporate">  
 <CUSTOMERNO>12</CUSTOMERNO>  
 <OPERATION>INSERT</OPERATION>  
 <EMAIL>billing@microsoft.com</EMAIL>  
 </row>  
 <row segment="private">  
 <CUSTOMERNO>1015</CUSTOMERNO>  
 <OPERATION>INSERT</OPERATION>  
 <NAME>John Doe</NAME>  
 <EMAIL>john.doe@example.org</EMAIL>  
 </row>  
 ...  
</Customers>

We now have to handle corporate and private customers, divided by a *segment*attribute in the *row* node. While individual private customers have a name, corporate customers don’t. When we peek into the XML Preview in the Data Factory, we see something similar to:

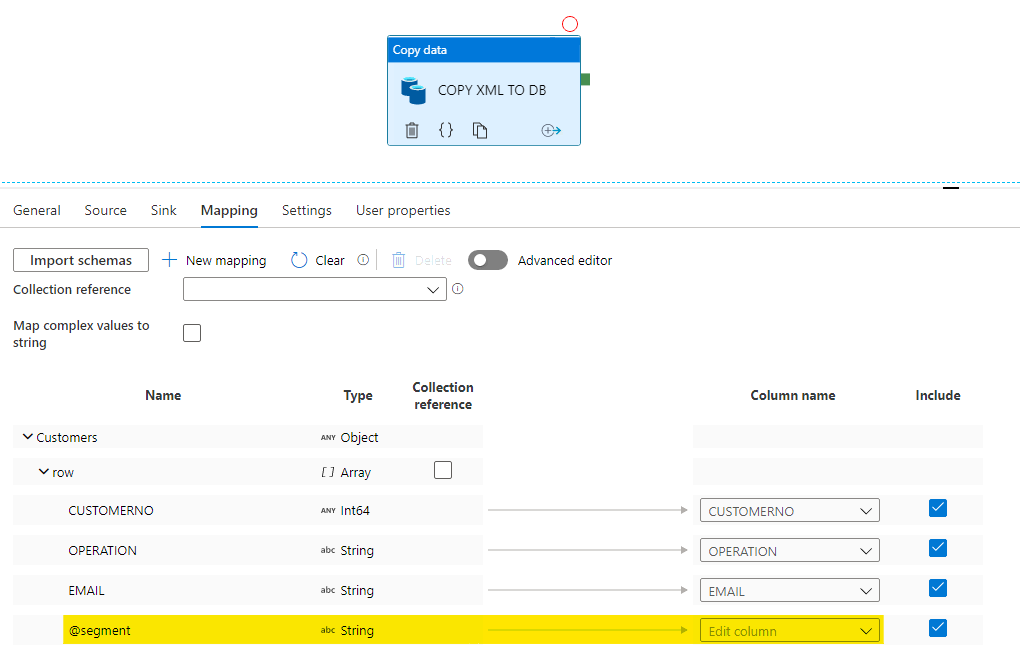


Attributes are inlined with leading “@”

To reflect these structural additions we alter our target table accordingly:

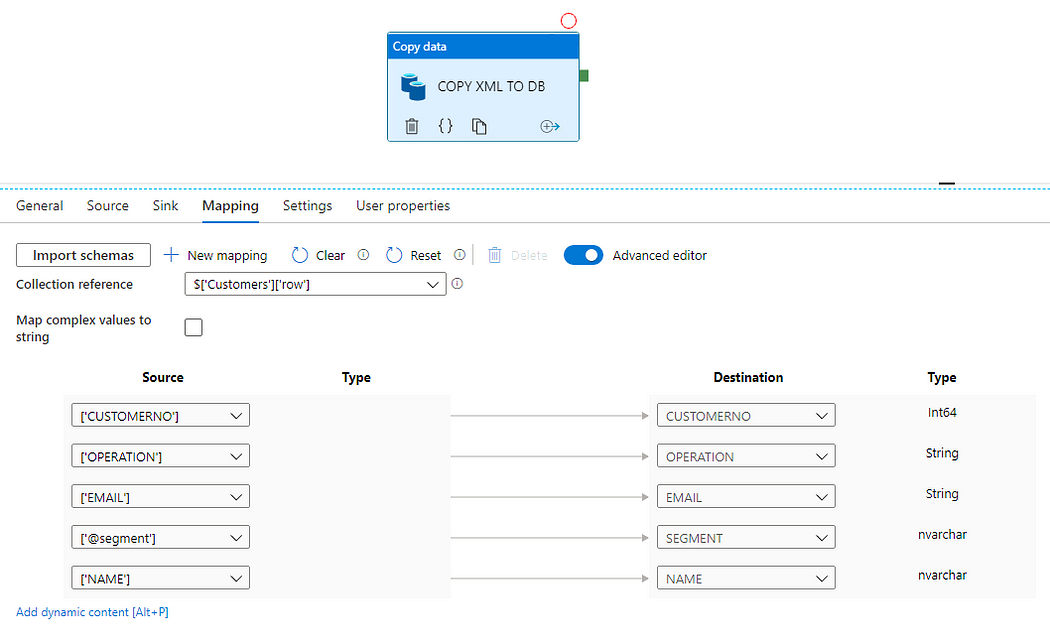
ALTER TABLE dbo.CUSTOMERS  
ADD [SEGMENT] NVARCHAR(255) NULL,  
 [NAME] NVARCHAR(255) NULL;

The automagical schema mapping fails to reflect the changes. The segment attribute is added to the mapping but not assigned to the SEGMENT column in the target table. The optional name field is missing altogether:



Segment is imported but unmapped, optional name is missing

This is where the Advanced Editor come into play, whereas advanced means you assign the mapping yourself using JSON paths for every attribute. After reassigning the Collection reference and switching into advanced mode its easy to extend the mapping by hand. Note that the NAME property in the Source is not recognized by default but has to be added using new “+ New Mapping” button.



Until now, it remains a mystery to us, why the newly added destination columns are of type *nvarchar* while Operation and Email as marked as *string*.

After the execution of the pipeline a check of the results in the database shows how robust the loading of XML attributes and optional properties is. Using the advanced mode also works for mapping substructures and reshaping the hierarchy of the data. For now expression support like concatenating multiple source columns of calculating a percentage of a values seems to be missing. Or otherwise it cannot be configured intuitively.

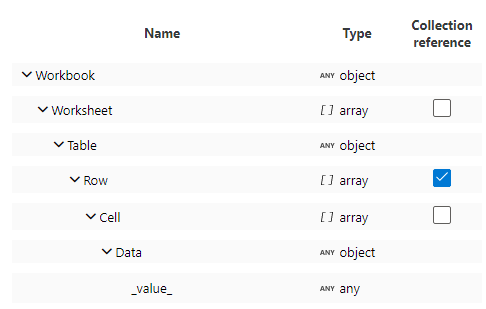
**For XML saved by Microsoft Excel, also use XML Datasets**  
Worth mentioning: Initially we looked into the extended file format support of the Azure Data Factory because some XML files generated by excel (using the .xml extension) should be loaded into our data Warehouse.

Importing those files using the Excel dataset type fails. So XML file type is the only thing that works — but it introduces a bunch of new complex challenges. First of all, the Collection Reference reflects the typical structure of Excel files divided by Worksheets, Tables, Rows and so on. It ends up to look something like this:

$['Workbook']['Worksheet'][0]['Table']['Row']

Next, Advanced Editor is a must to map the non trivial structures, adressing cell values by index like (subtree of Collection Reference):

['Cell'][1]['Data']['\_value\_']



Do you recall the Excel structure?

Note that **\_value\_** is the representation of the text value when the XML node also has attributes — which appears frequently when the Excel cell is styled in any way. Styles and formats are namespaced with “ss” referencing

xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"

Although the mapping ends up to be handmade (even changing the JSON code of the ADF pipeline — bad thing!) it works and Excel-generted XML files can be imported into the database or transformed to other file types.