XML to tables (csv) with NiFi and Groovy

All code used in this article can be downloaded from <https://github.com/maxbback/nifi-xml/>

# Problem with XML and design for converting XML to CSV

XML is a common format used by many applications, so it will be one of the formats you need to support in your Big Data platform.

 XML is a structured format that is good for application but not so convenient if you are used to work with SQL and tables so I decided to design and write code for transforming XML to CSV.

You might thing that converting from XML to CSV is a very simple thing, but it is not XML is a hierarchical tree structured data format while CSV is a flat column based format.

A branch in XML can key value pairs and branches, branches is of type one2one or one2many

When we convert an XML the best is if we have a schema so we know  how the XML file can look like as we then has the description for how the DB table shall be defined.

My design is based on that we have a XSD schema as a base to secure that we know which table columns we need.

In below schema we have example of many of the patterns we often see in a schema, lets discuss them.

Element person has an attribute name and can exist in any number of instances and needs to be its own table.

Element person has direct element child’s;

* Full\_name is a simple element that is of type string and is a column in person table
* Child\_name is a simple element of type string that can exist in 0-5 times and can either be its own table och be 5 columns in person table “child\_name1-5”
* Child\_name\_complex is unbound and has to be its own table

Element child\_name\_complex has direct element child’s;

* Length is a simple element of type decimal and is a column in child\_name\_complex table
* Age is also a simple element and will also be a column in child\_name\_complex
* Mother is a complex type but can only exist 1 time and can then be flattened out to be columns in child\_name\_complex

[code lang="XML"]

<xs:element name="persons">

<xs:complexType>

<xs:sequence>

<xs:element name="person" maxOccurs="unbounded">

<xs:complexType>

<xs:sequence>

<xs:element name="full\_name" type="xs:string"/>

<xs:element name="child\_name" type="xs:string" minOccurs="0" maxOccurs="5"/>

<xs:element name="child\_name\_complex" minOccurs="0" maxOccurs="unbounded">

<xs:complexType>

<xs:sequence>

<xs:element name="length" type="xs:decimal" minOccurs="0" nillable="true"/>

<xs:element name="age" type="xs:decimal" minOccurs="0" nillable="true"/>

<xs:element name="mother" minOccurs="0" maxOccurs="1">

<xs:complexType>

<xs:sequence>

<xs:element name="name" type="xs:string" minOccurs="0" nillable="true"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

[/code]

## Rules for XML to tables

We need simple rules to follow when we are flattening out the XML tree to tables and columns.

1. Element with maxOccurs=”unbound” must be its own table
2. Element with maxOccurs=”1” can be flattened out to be column in its parent but child elements must be renamed to include the parent name
   1. Example mother\_name for mother element name
3. Attributes is column in its parent
4. Element of type simple that has a limited maxOccurs can be flattened out to columns, but if the value is high it’s better to generate a new table
5. Tables of child’s need to be unique this can be accomplished be including parent table name in the table name or make each table unique by adding a suffix to them.
6. If element that represent a table has 1 attribute we treat this as the primary key
7. All child table has the parent primary key as a foreign key
8. If table do not have a natural primary key we set it to current timestamp of execution for each top level table.

Sometimes XSD schema is very open which will result in many tables, if you know your data and the alternatives of its look, it is better to make a more strict schema which will dramatically lower the number of tables, if you use many  1t1 branches remove them and make them simple elements instead as you then will get shorter column names.

## Use Cases

Let’s define our expectations of what we need;

* DDL to create tables
* Hierarchical diagram in a simple form to show table relations and columns in each table
* The content of each table after parsing the XML passed on in its own flow file for each table

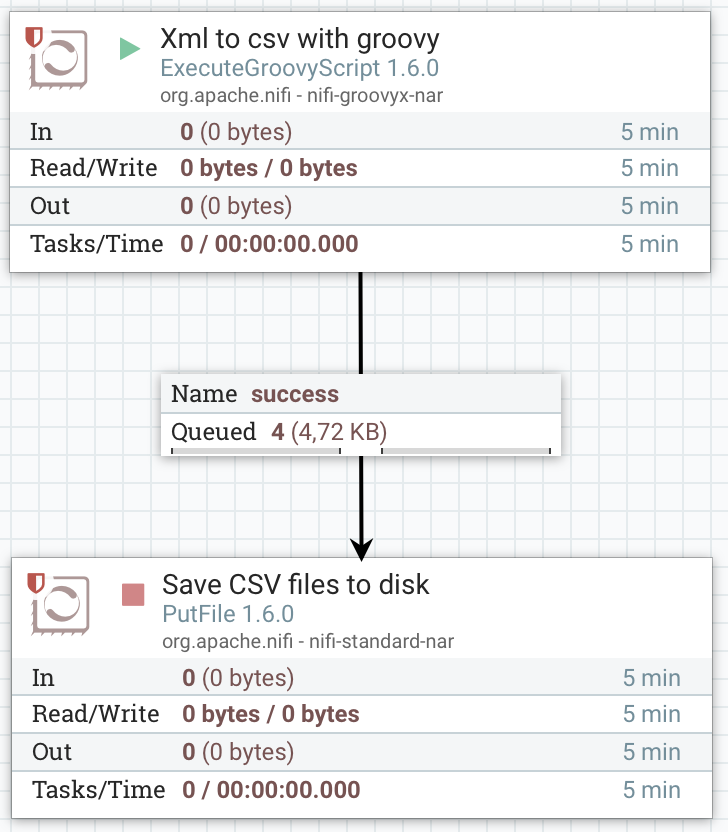
# Implementation

All code used in this article can be downloaded from <https://github.com/maxbback/nifi-xml/>

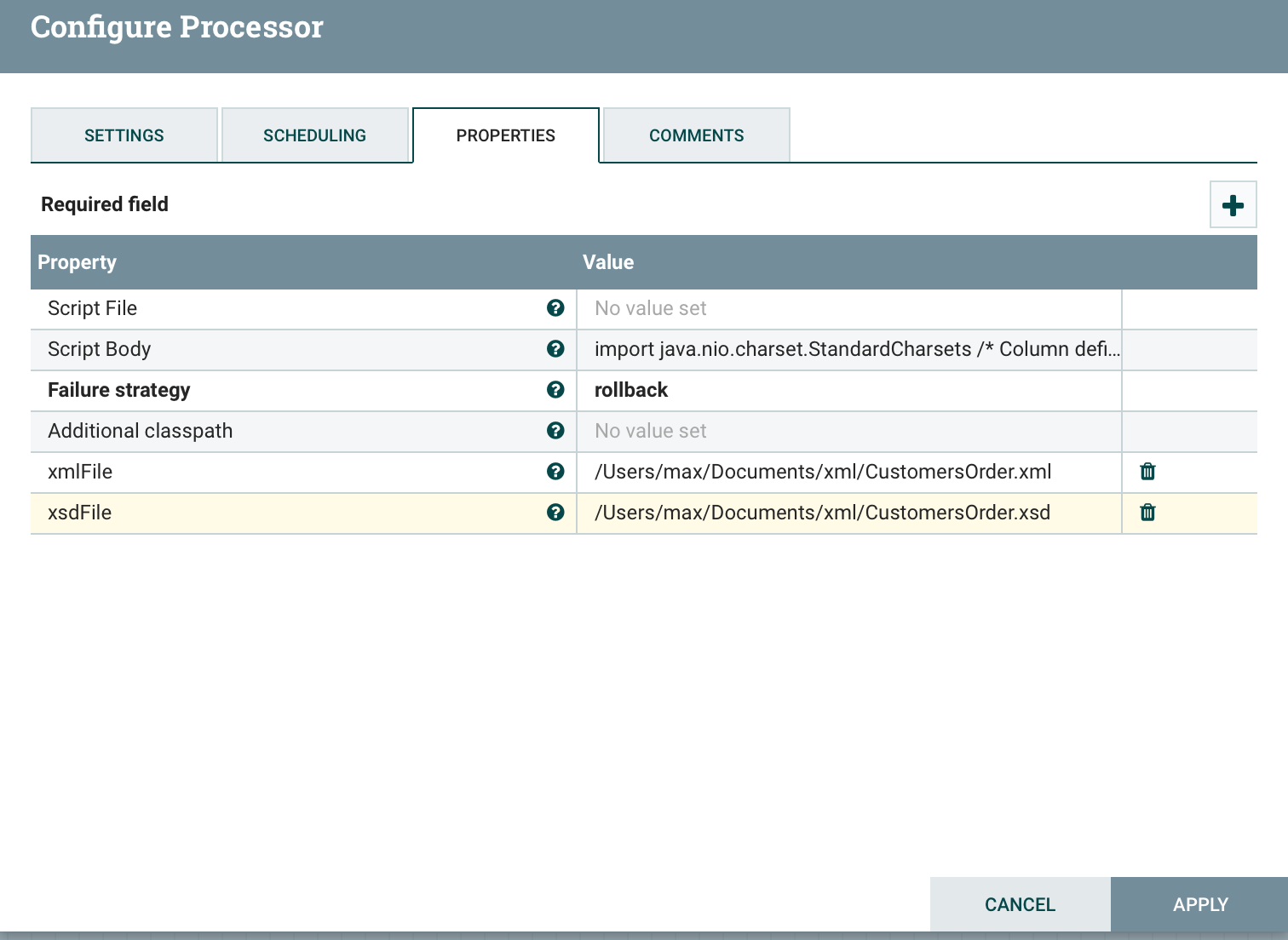
In this part I have made an implementation in NiFi with a groovy processor.

Why write the processor in Groovy and not Java? Groovy is almost as fast as native Java and the amount of code needed to do this is far less than with Java.

My small NiFi implementation contains of two steps, my groovy script that converts a XML file to one or more CSV files depending on how many tables the XML file needs for its content, the second step is saving all CSV files down to disk.



## Xml to csv with groovy configuration



I have in this example included the whole Groovy script in the processor attribute Script Body.

My script requires to attributes to be defined;

xmlFile and xsdFile, they need to point to the location of XSD and XML file to be used.

When you implement this you will likely need the files to be more dynamically identified and it can be done in a few different ways;

* have a pre-step that lists all files to be loaded and then modify this groovy script to read the flow file and collect files to process
* modify this script so it collects all XML files from a path and not just one
* pass an XML file as a flow file to this processor

All above methods needs modifications of this script, and in a coming article I will give you some code example of how this can be done.

## The result of an execution

As we can see in the process definition my XML file was converted into four tables each passed in its own flow file.

The flowfiles are dynamically created with this small code;

*// Define flowfile list that will hold each table*

*def flowFiles = [] as List<FlowFile>*

*// Create a new flowfile*

*flowFile = session.create()*

*// Add it as a write flowfile to the session*

*flowFile = session.write(flowFile, {outputStream ->*

*} as OutputStreamCallback)*

*// Add the flow file to the list of flowfiles*

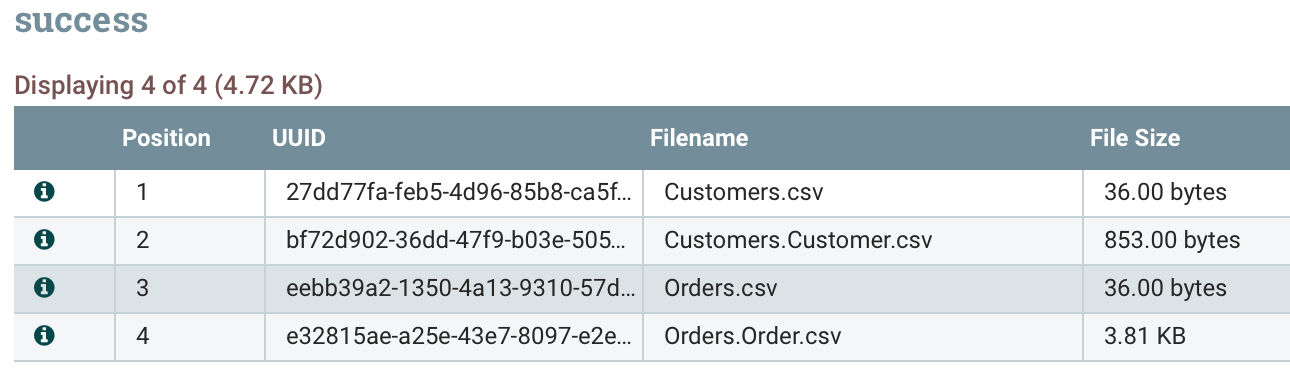
*flowFiles << flowFile*

*// When all flow files are gernated we release the list of flowfiles*

*session.transfer(flowFiles, REL\_SUCCESS)*

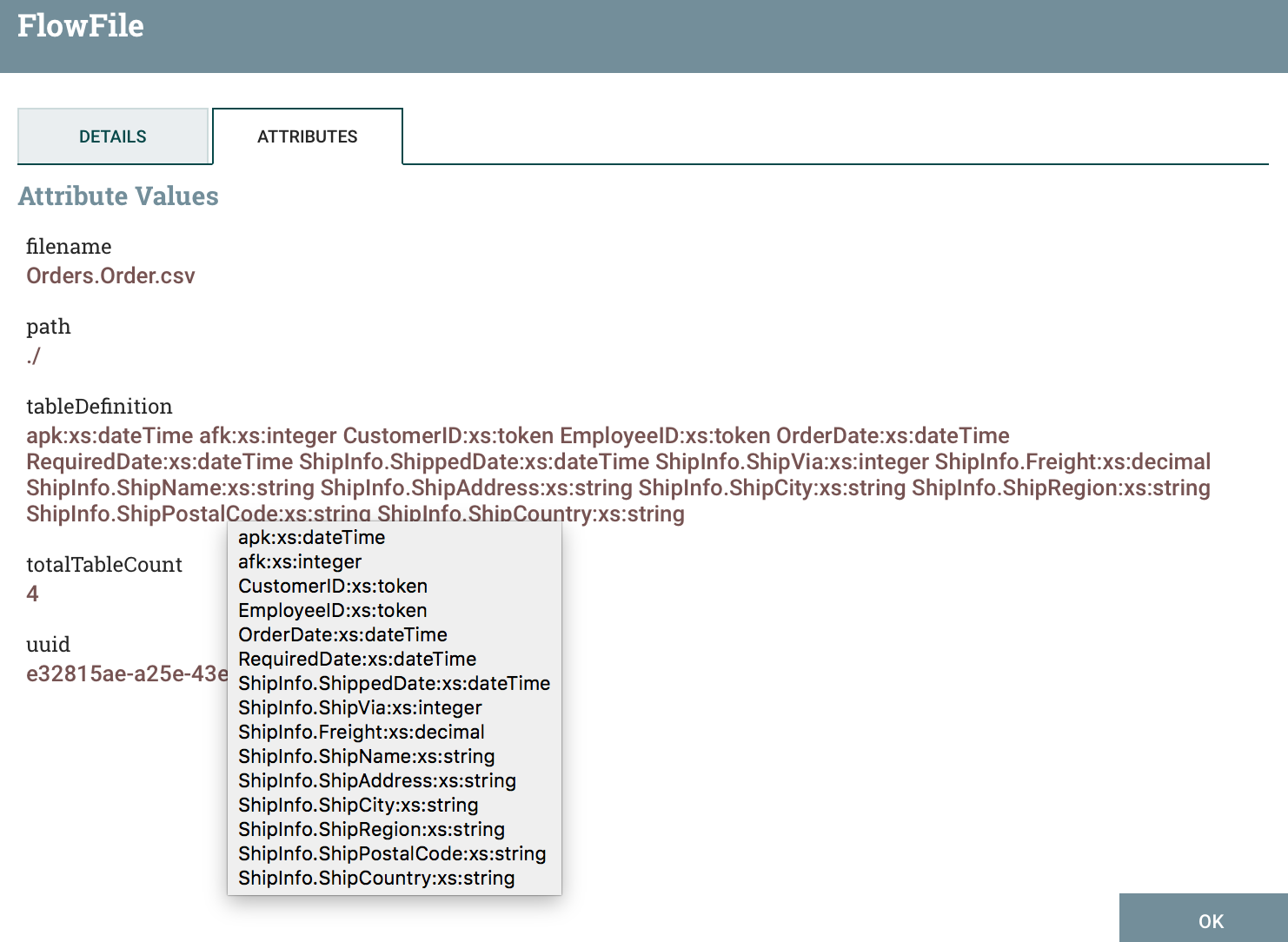
The session.create function returns a new flow file, we then define it as an output stream so we can write to it, and output stream always has to end with as OutputStreamCallback, and the last thing needed is to release all flowfiles with status success.

If we list the queue we can see the name of the output file as well as the size of each file.



If we check the details of each flow file we can see the attributes as well as the content of the file.

Let’s start with the attributes;



For each flowfile I have added 3 attributes filename, tableDefinition and totalTableCount.

The totalTableCount represents all tables found, but is not the same as number of generated table in all situations as I only generate a table if it contains data.

The XSD file can contain definition of tables that is not part of an XML file and this script focus only on content so it is not generating flow files if it do not contain content.

If we hover over tableDefinition we can see that it is a newline separated list of column name and the column type, where column type is XML styled types and need to be converted to the types supported by Hive when ingesting it.

In this article I am not covering how we dynamically create and validate hive tables, but it’s an interesting topic that I might touch on in a future article.

It’s easy to extend the attribute list with for example number of records and other things.

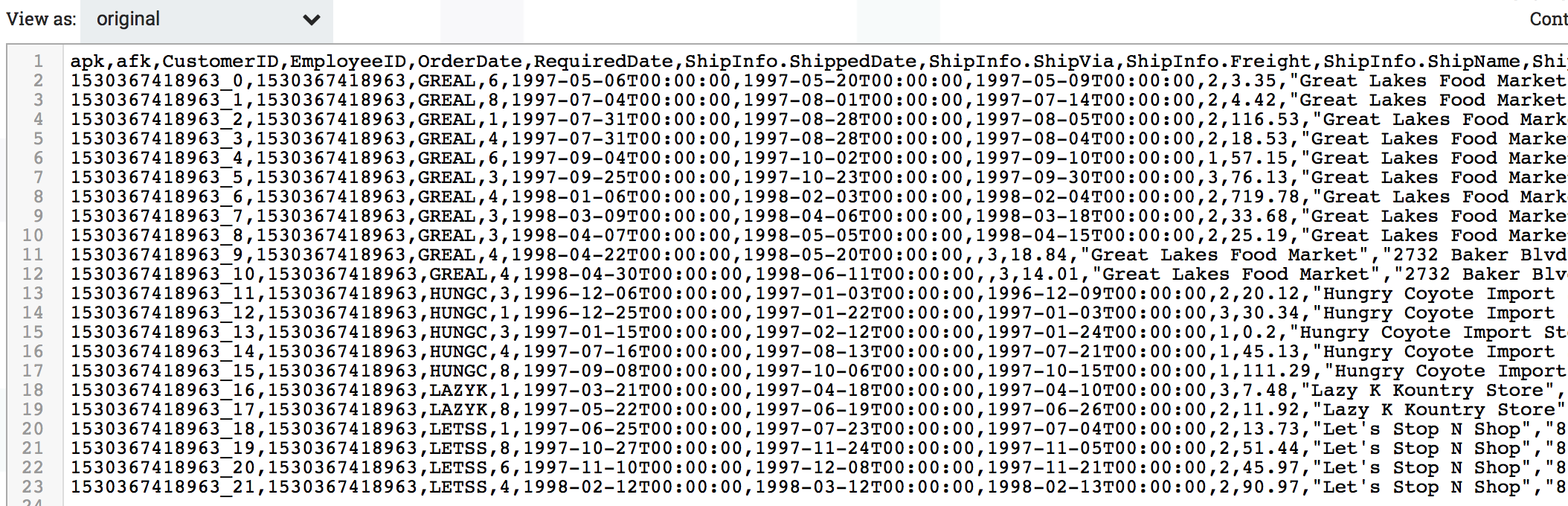
Setting and attribute is done with putAttribute see following code snippet;

*flowFile = session.putAttribute(flowFile, 'totalTableCount', totalTableCount.toString())*

If we look at the result we can view this in NiFi as it is pure text.

The first two columns are autogenerated by the script and represent primary key and foreign key for this table.

If a branch has a single attribute on the top level I have decided that this represents the primary key for this table, if non found I use a timestamp of the execution and for each record I add the record number.



The XML and XSD files are copied from Microsoft <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/sample-xml-file-customers-and-orders-linq-to-xml-2> but the XSD is modified to match the style I used in my code, which is the more standard way of defining XSD files compared to the more modern way, which breaks up the code into section and referencing them by using the type keywork, it is rather easy to extend the code to support this styling as well but it is not in the scope of this article.