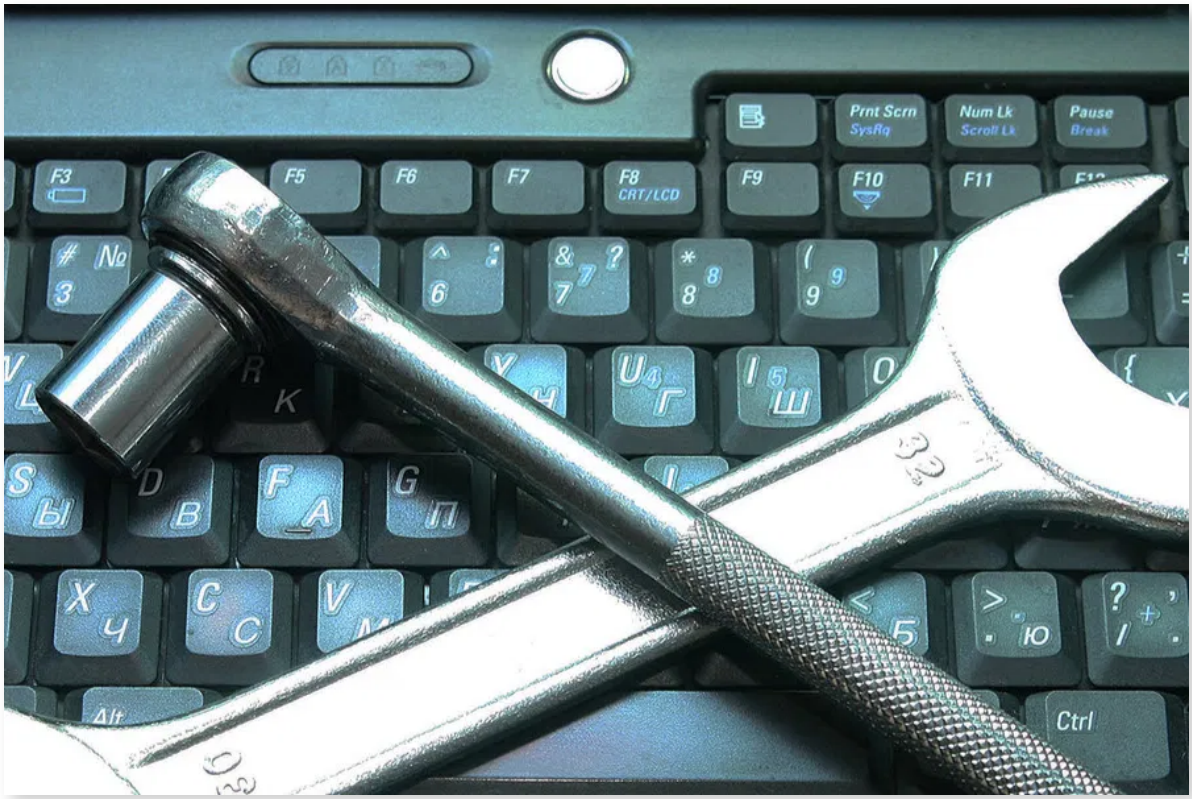
MICROSOFT

Building Modular Pipelines in Azure Data Factory using JSON data

by **Gary Brandt** on May 31st, 2020 | ~ *9 minute read*



Azure Data Factory (ADF) pipelines are powerful and can be complex. In this post, I share some lessons and practices to help make them more modular to improve reuse and manageability.

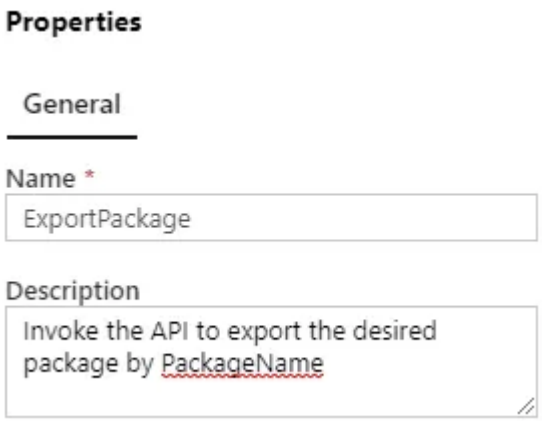
Why Modular Pipelines?

as well. This is similar to any procedure in code, the longer it gets the ability to edit, read, understand becomes harder and harder. As a result, software best practices promote refactoring procedures into smaller pieces of functionality. This helps in countless ways like testing, debugging and making incremental changes. The same approach can be taken with pipelines. However, there are some things about pipelines to take into account in order to do so:

- Break down into units of work
- Leverage dynamic with parameters
- Leverage expressions for behavior
- What about output results
- Utilized Functions and JSON data to pass messages

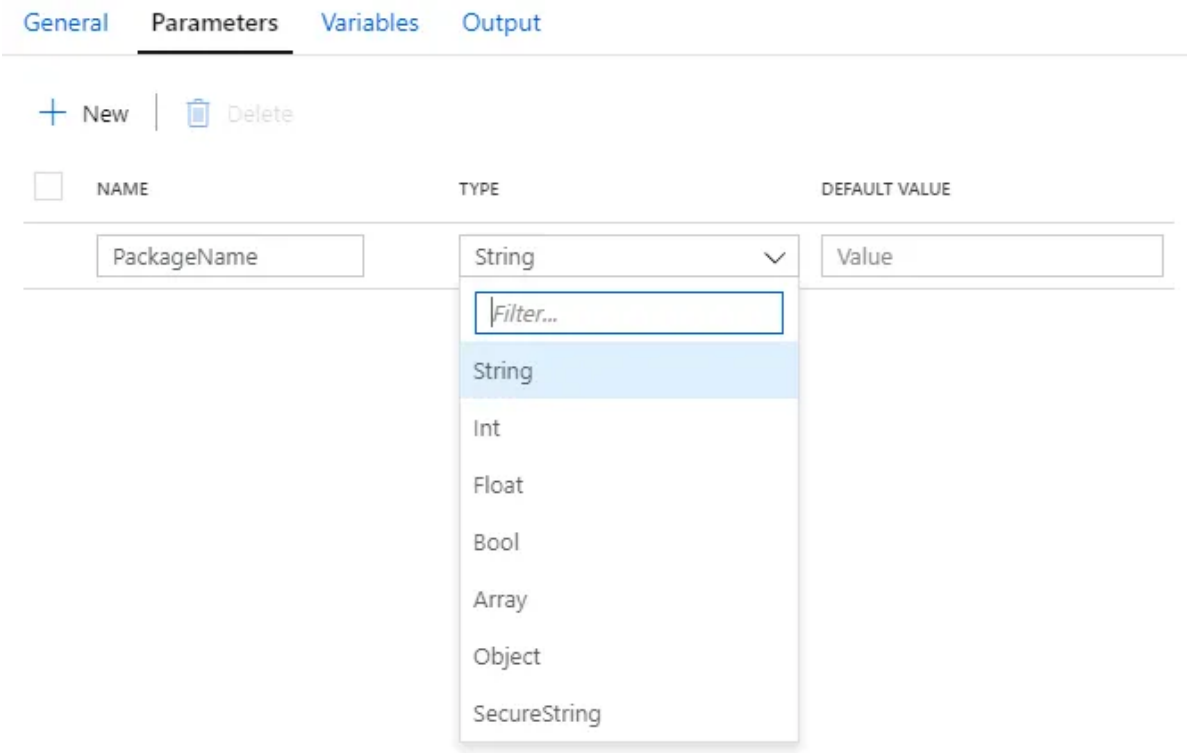
Unit of Work

Just like code, a pipeline should be focused on a specific function and be self-contained. It's logic, rules and operations will be a "black box" to users of the pipeline, requiring only knowledge of "what" is does rather than "how" it does. ADF pipelines include propertied for Name (required) and Description which can be used to provide for basic documentation of the pipeline to communicate with it's consumers.



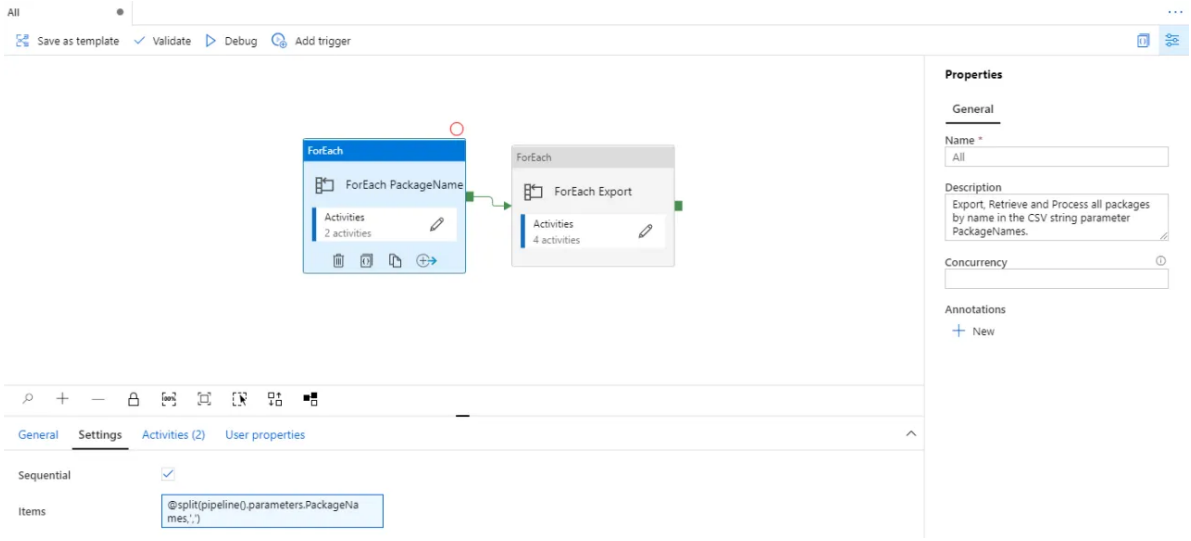
Input Parameters

Parameters can be different types: String, Int, Float, Bool, Array, Object, and SecureString. Each parameter can also include a Default Value, making a specific parameter value optional when the pipeline is run.



Expressions

Part of the rules of a pipeline can include expressions that process the parameter values provided when run. These expressions can prepare the data for activities within the pipeline. The use of expressions introduces a dependency on the content and format of parameter values beyond simple being required or optional. For example, a String parameter may be a comma-separated-value (CSV) which will be handled by an expression. For these cases, the pipeline description should detail this and/or other documentation should accompany the pipeline so that users understand how is is to be used.



In this example, the dynamic expression for Items of the ForEach PackageName activity uses the PackageNames parameter as CSV string to iterate over the values provided:

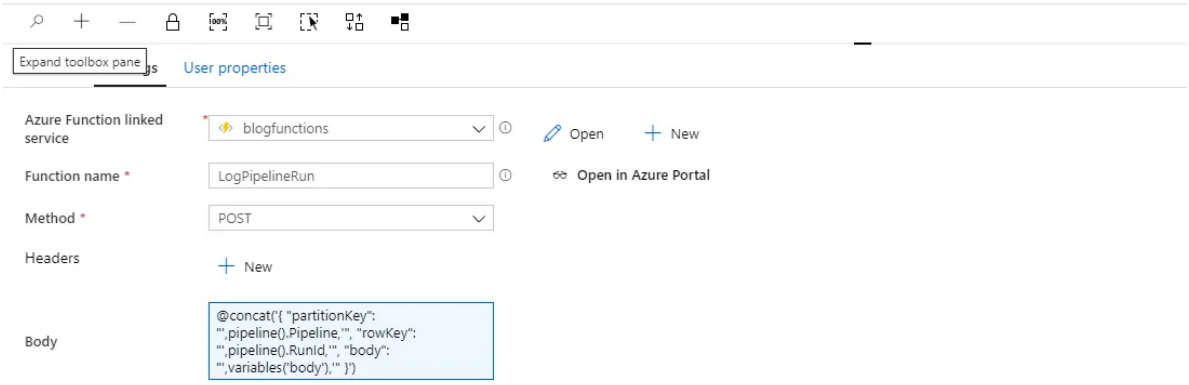
```
1. @split(pipeline().parameters.PackageNames,',')
```

There will be more samples in the **Real-World Examples** section that follows.

Output Results

Just as with input parameters, in order to be reusable, it should be obvious that a pipeline will need to return details of the activities performed so that subsequent pipelines or other activities can take place. Just like code where a method or procedure returns *void*, this is not a requirement and the pipeline results can be limited to the manipulated datasets or other artifacts of the pipeline run. However, in a modular case, it is likely the pipeline will need to convey details for further operations. One of the lessons learned is the fact that ADF pipelines do NOT provide a means to define outputs in a similar way to input parameters. Instead, the pipeline returns only one value, the *RunId*. The *RunId* is in the form of a globally unique identifier (guid). In order to associate more details of the pipeline results (e.g. output value(s)) with other operation, some form of storage of those results will be required so

I've found Azure Table Storage to be a convenient solution for this. Using a table with a partition key for the pipeline name and a row key for the *RunId*, other data can be included as needed. A pipeline that needs to provide more detailed result will insert a row. Then, activities needing those details can read the row based on the specific pipeline's *RunId* during execution.



In this example, an Azure Function activity is used to insert the record via the Body expression:

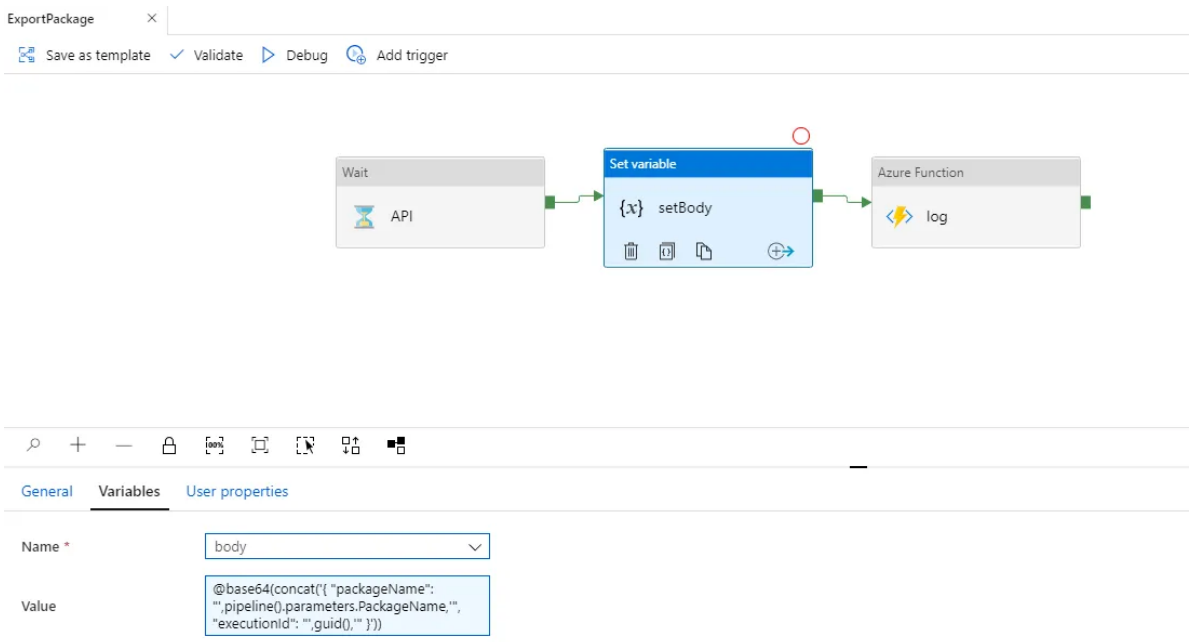
```
@concat('{
   "partitionKey": "',pipeline().Pipeline,'",
   "rowKey": "',pipeline().RunId,'",
   "body": "',variables('body'),'"
}')
```

There will be more detailed examples in the **Real-World Examples** section that follows.

Using Azure Functions and JSON data

have different output values. This shouldn't be surprising just like you would expect the input parameters to differ based on the pipeline and the functionality it implements. To address this, I leveraged JSON data in string format as the *Body* column. The JSON data can have a dynamic schema for each pipeline as needed. However, this JSON schema is now a dependency for users of the pipeline. Like shown in the CSV input parameter above, where "callers' of a pipeline need to understand the input format requirements, those same "callers" will need to understand the output *Body* schema in order to extract the details contained. ADF expressions play a key part in creating *Body* "messages" inserted to the table and parsing the same *Body* into values for other activities.

This is typically done with a Set Variable activity as was seen in the ExportPackage pipeline above. Doing so, keeps the expression for calling the Azure Function simpler and the schema details within the Set Variable activity expression.



This example expression creates a JSON string from other pipeline and/or activity values. The JSON string is base64 encoded because it will be used as the value of the JSON *Body* member of the Azure Function method.

```
4.  }'))
```

Note: the use of guid() as the API executionId value is for example purposes only and simulates the API behavior. More on that the the next section.

Real-World Examples

As part of a recent project exporting Microsoft Dynamics 365 data into a data warehouse using the Data management package REST API, it was crucial to design the pipelines so they could be dynamically executed for different scenarios and the resulting stages of the export process be coordinated.

This process includes three component pipelines:

package
- GetPackage: for the returned *ExecutionId*, after successful export completion, download the package zip file, extract the zip file to Azure Blob Storage and return the storage path for the package contents
- ProcessPackage: for each data entity in the exported package contents, upsert the data into the data warehouse

Note: in the examples below, the API calls and data warehouse activities are simulated with a Wait activity so that the pipeline can demonstrate the modular pattern without any external dependencies.

ExportPackage Pipeline

This pipeline is quite simple, just uses the PackageName parameter to invoke the D365 REST API to trigger the export process. The API returns an ExecutionId to be used to monitor the asynchronous process before the output can be retrieved. For demo purposes, the API here returns a new guid as the *ExecutionId*.

Input:

```
1. PackageName, string
```

```
    "executionId": "',guid(),'"
}'))
```

GetPackage Pipeline

This pipeline in reality is more complex than what is shown here. This shows a single API call to get the package output. In reality the package needs to poll the status of the export until complete. Once complete it makes an API call to retrieve the download URL. With the URL, the output zip file can be copied and extracted to Azure Blob Storage. Those details for the D365 API are not needed to show the modularity of the pipeline. For demo purposes, the entity count is a random integer between 1 & 5 to simulate different package contents.

Input:

```
PackageName, string
ExecutionId, string
```

Output:

```
@base64(concat('{
    "packageName": "',pipeline().parameters.PackageName,'",
    "executionId": "',guid(),'",
    "entityCount": '.string(rand(1,6)).'
```

This pipeline needs to iterate over every entity in the package output contents.

Input:

```
PackageName, string
ExecutionId, string
EntityCount, int
```

Output:

```
@concat('{
   "partitionKey": "',pipeline().Pipeline,'",
   "rowKey": "',pipeline().RunId,':',item(),'",
   "body": "',base64(concat('{
         "packageName": "',pipeline().parameters.PackageName,'",
         "executionId": "',pipeline().parameters.ExecutionId,'",
         "entityNo": ',item(),'
      }')),'"
}')
```

ProcessPackage Pipeline – ForEach Entity

The core of the pipeline is the for each Entity, which performed the upsert of the data entity into the data warehouse.

All Pipeline

This pipeline is the coordination of the three pipeline It utilizes the PackageNames CSV parameter as a batch to Export, Get & Process.

Input:

```
1. PackageNames, string (CSV)
```

Output:

```
1. none
```

All Pipeline – ForEach PackageName

For each package name in the CSV parameter, the **ExportPackage** pipeline is run. Since each **ExportPackage** pipeline logs output data, the *RunId* for each pipeline is appended to an array variable so that the results can be used for the next activity.

All Pipeline – ForEach Export

After the export for each package name in the CSV parameter is run, for every *RunId* the **GetPackage** and **ProcessPackage** pipelines are run. In this pipeline an Azure Function activity is used to get the data for the associated **ExportPackage** *RunId* and uses it to run the **GetPackage**. pipeline In the same way, another Azure Function activity is used to get the data for the associated **GetPackage** *RunId* and uses it to run the **ProcessPackage** pipeline.

Get **ExportPackage** results:

```
@concat('{
   "partitionKey": "ExportPackage",
   "rowKey": "',item(),'"
}')
```

Get **GetPackage** results:

```
@concat('{
   "partitionKey": "GetPackage",
   "rowKey": "',json(activity('Execute Get').output).pipelineRunId,'"
}')
```

Note: from the output of the **GetPackage** pipeline, the *pipelineRunId* value is needed to read the table record as the row key. The execute pipeline activity *output* value is converted to a json object in order to reference the property value.

Code

The JSON files for the example modular pipelines, as well as the Azure Function code file, are available ModularADFwithJSON.

Tags

#MicrosoftCloud Azure Azure Data Factory Dynamics 365 Microsoft

About the Author

Gary Brandt is a Senior Solutions Architect focusing on custom solutions design, development, and delivery utilizing the Microsoft platform and Azure cloud services. He has more than 20 years of development and consulting experience and has seen a lot of different technologies over the years. He is always excited about the emerging changes and to see how they will impact the work we do today and in the future.

More from this Author

Leave a Reply

← PREVIOUS POST NEXT POST →

Development

Microsoft

Follow Us